L'extension tdsfrmath*

Le TEXnicien de surface†

22 juin 2009

Résumé

Cette extension veut fournir des macros à « l'utilisateur final » pour créer des documents mathématiques ayant un aspect français.

Abstract

This package provides a bunch of macros to help the "final user" to produce maths texts with a definite french look. For there is a marked aspect of localisation, I don't provide any English documentation.

Table des matières

*Ce document correspond au fichier tdsfrmath v1.3, du 2009/06/22.

†`le.texnicien.de.surface@wanadoo.fr`

1 Introduction

Le but de cette extension est de fournir des macros prêtes à l'usage à des professeurs de mathématiques des collèges, lycées — et plus si affinités — qui voudraient bien utiliser LaTeX sans trop mettre le nez dans la programmation ni devoir retenir des choses aussi barbares que `\overrightarrow` pour faire un vecteur `;-)`

De plus elle tente de donner aux mathématiques un aspect vraiment plus français. On aura par exemple « dx » au lieu de « dx » dans les intégrales et les dérivées.

`tdsfrmath.sty` s'appuie lourdement sur `amsmath` qu'il requiert. On n'aura donc pas besoin de le charger avec un `\usepackage` si l'on utilise `tdsfrmath`.

Remarque : depuis la version 1.2, `amssymb` est utilisé si l'option **`avecmathdesign`** est fausse comme c'est le cas par défaut. Si on utilise `mathdesign.sty`, on donnera la valeur *`true`* à **`avecmathdesign`** et, dans ce cas, `amssymb` ne sera pas chargé. Voir la section 2.5, page 6.

À l'origine de cette extension, je trouve un vieux fichier `.sty` que je m'étais concocté, par petits bouts, lorsque je sévissais encore dans le secondaire. Ayant appris un peu de LaTeX depuis, j'ai pensé à en améliorer les macros. J'ai aussi consacré quelques heures à la francisation de l'aspect, chose à laquelle je n'avais accordé que peu d'attention jusqu'ici car, je dois l'avouer, je m'étais beaucoup servi de LaTeX pour produire des textes mathématiques en anglais.

En tout cas, pour en rassurer certains qui pourraient considérer qu'ils ne pourraient jamais arriver à un tel niveau (*sic*) d'écriture de macros : ceci est le résultat de nombreuses heures étalées sur plus de 15 ans. Je dois par ailleurs remercier publiquement tous ceux qui sur `fr.comp.text.tex` ont répondu à mes questions, pas toujours très profondes d'ailleurs, et m'ont apporté une aide précieuse jusqu'aujourd'hui même dans l'utilisation de notre outil préféré de création de document.

2 Utilisation

Pour la première fois, plutôt que des options, j'utilise le système de clés et valeurs que permet `xkeyval.sty`. De même, j'utilise `xargs.sty` qui permet la définition de commandes admettant plusieurs arguments par défaut.

Dans le cours du texte une clé est écrite **`clé`** et une valeur *`val`*. Les clés dont les noms comportent des majuscules sont booléennes c.-à-d. que leur valeur est soit *`true`* — vrai — soit *`false`* — faux. Les clés marquées « choix » dans la table 3, page 19, permettent de choisir entre quelques valeurs prédéfinies. D'autres enfin attendent un texte avec plus ou moins de restrictions suivant ce à quoi servira le texte.

J'ai amplement (?) commenté la partie contenant le code, et on s'y reportera pour les détails d'impémentation, mais je commence ici par présenter toutes les options et toutes les macros de cette extension.

2.1 À propos des clés booléennes

Nouveauté de la version 1.3.

Les clés booléennes, comme **`taupe`** p. ex., sont des clés qui n'admettent pour valeur que `true` ou `false`. L'extension `xkeyval.sty` me permet de leur donner une

valeur *par défaut* et une valeur *d'origine* : la valeur d'origine est celle que possède la clé lorsque l'on charge `tdsfrmath.sty` sans aucune option avec `\usepackage{tdsfrmath}`. La valeur par défaut est celle que l'on obtient en mentionnant la clé sans lui donner de valeur comme avec `\usepackage[taupe]{tdsfrmath}` qui revient au même que `\usepackage[taupe = true]{tdsfrmath}` car la valeur par défaut de la clé **taupe** est *true*.

2.2 Chargement optionnel

`tdsfrmath.sty` permet de charger du code de manière optionnelle. Ce code est placé dans des fichiers d'extension `.sto`, à savoir : `taupe.sto` contenant des macros destinée plutôt à une utilisation en classe prépa ; `suite.sto` dont les macros ne traitent que des suites.

taupe

À chacun de ces fichiers correspond une clé booléenne, de même nom, dont la valeur d'origine est *false* ce qui entraine que ces fichiers ne sont pas chargés.

suite

Si on veut utiliser les macros définies dans `taupe.sto`, on appelera `tdsfrmath` par : `\usepackage[taupe=true]{tdsfrmath}` ou encore avec `\usepackage[taupe]{tdsfrmath}` depuis la version 1.3 puisque la valeur par défaut de **taupe** est *true*.

N v1.3

Le fichier `taupe.sto` contient des définitions qui dépendent de la clé **ArgArcMaj** — **arg** et **arc** avec **maj**uscule — dont la valeur d'origine est *false* ce qui entraine que les noms des fonctions circulaires et hyperboliques réciproques — comme $\operatorname{argch} x$ — sont écrites en minuscule. En donnant la valeur *true*, valeur par défaut, à la clé **ArgArcMaj**, ils prennent une majuscule — on a alors $\operatorname{Argch} x$.

ArgArcMaj

2.3 Réglage de la police calligraphique

Trois clés règlent le choix de la police calligraphique en mode mathématique. D'origine la clé booléenne **CharPoCal** — pour **Char**ger une **Po**lice **Cal**ligraphique — est *true* ce qui permet de définir la police calligraphique pour remplacer `\mathcal` qui serait celle que l'on obtiendrait si **CharPoCal** avait la valeur *false*.

CharPoCal

Lorsque **CharPoCal** vaut *true* — ce qui est également sa valeur par défaut —, il faut définir les clés **calpack** et **calcomd**.

La clé **calpack**, qui contient *mathrsfs* par défaut, prend pour valeur le nom, sans l'extension `sty`, d'un module donnant accès à une police calligraphique, p. ex., `mathrsfs` ou `eucal`.

calpack
mathrsfs

La clé **calcomd**, qui contient *mathscr* par défaut, prend pour valeur le nom d'une macro **sans** la barre oblique initiale. C'est la macro permettant de *passer* en police calligraphique. L'extension `mathrsfs.sty` contient bien la macro `\mathscr`.

calcomd
mathscr

En résumé, si l'on veut utiliser le `\mathcal` tel que proposé par LaTeX plus `amsmath.sty`, on chargera :
`\usepackage[CharPoCal=false]{tdsfrmath}`
— ce que l'on fera également pour utiliser `fourier.sty` si on veut bénéficier de la redéfinition de `\mathcal` qu'opère cette extension — et si l'on veut utiliser `eucal.sty` et sa commande `\mathcal` — eh oui ! cette extension redéfinit `\mathcal` — il faudra
`\usepackage[calpack=eucal, calcomd=mathcal]{tdsfrmath}`

on remarquera que `CharPoCal=true` n'est pas nécessaire puisque c'est la valeur par défaut.

Si nécessaire, on peut passer une option à l'extension *passée* à **calpack**, en
caloptn renseignant la clé **caloptn** comme dans, p. ex.

```
\usepackage[calpack=euscript,
            caloptn=mathcal,
            calcomd=mathcal]{tdsfrmath}
```

dans lequel on remarquera qu'il faut bien donner une valeur à **calcomd** comme je l'ai déjà écrit ci-dessus.

`Remarque` : j'ai introduit dans la version 1.2 une clé permettant d'utiliser conjointement `tdsfrmath.sty` et `mathdesign.sty`, cf. 6. Dans ce cas, les clés **CharPoCal**, **calpack**, **caloptn** et **calcomd** sont ignorées donc inutiles.

`\manus` Dans tous les cas, on accède à la police calligraphique avec la macro `\manus`, à un seul argument obligatoire, qui est définie de telle sorte que l'on puisse saisir « `et dans \manus{c} on trouve` » pour obtenir « et dans $\mathscr{C}$ on trouve ».

2.4 Réglage de la police « gras de tableau »

Par défaut, le « gras de tableau » (*blackboard bold*) est celui de LaTeX plus `amsmath.sty` c.-à-d. $\mathbb{G}$. Il en existe bien d'autres versions que l'on trouvera dans le fameux `symbols-a4.pdf` disponible généralement dans votre distribution, et donc sur votre disque, dans `texmf-doc/doc/english/comprehensive/`.

Pour permettre de redéfinir la police du gras de tableau, je fournis un mécanisme similaire à celui qui précède. On utilisera alors la macro `\grastab` pour obtenir le « gras de tableau » choisi.

CharPoGdT La clé booléenne **CharPoGdT** — pour **Char**ger une **Po**lice **G**ras **d**e **T**ableau — vaut *false* à l'origine mais a *true* pour valeur par défaut.

gdtpack En fixant **CharPoGdT** à *true*, on peut définir la clé **gdtpack** en lui donnant le nom de l'extension qui fournira le gras désiré, on peut éventuellement lui passer
gdtoptn une option avec **gdtoptn** et, toujours éventuellement, fixer **gdtcomd** avec le nom de la macro désirée — toujours sans barre oblique inverse — seulement, cette fois,
gdtcomd du fait du choix de la valeur par défaut de **gdtcomd**, on n'aura pas besoin de fixer la valeur de **gdtcomd** si la macro est encore `\mathbb`.

Voici ce que l'on écrira pour obtenir le gras de tableau tel que fournit par `dsfont.sty` avec l'option `sans` et la commande `\mathds` — tous les goûts sont dans la nature —

```
\usepackage[CharPoGdT=true, gdtpack=dsfont, gdtoptn=sans,
            gdtcomd=mathds]{tdsfrmath}
```

ou encore :

```
\usepackage[CharPoGdT, gdtpack=dsfont, gdtoptn=sans,
            gdtcomd=mathds]{tdsfrmath}
```

et `\grastab` donnera ce que l'on voit dans `symbols-a4.pdf`.

On peut également [1], lorsque la clé **CharPoGdT** vaut *true* ne pas définir la clé **gdtpack** mais définir la clé **gdtcomd** comme précédemment. Ce mécanisme permet d'utiliser p. ex. le gras « normal » pour le « gras de tableau » avec

```
\usepackage[CharPoGdT=true, gdtcomd=textbf]{tdsfrmath}
```

1. Ce changement à lui seul justifie le passage à la version 1.1. C'est à la demande générale de Maxime Chupin sur `fctt` que je procède à la redéfinition du mécanisme du choix du gras de tableau `;-)`

ou encore :
`\usepackage[CharPoGdT, gdtcomd=textbf]{tdsfrmath}`
et on aura alors, p. ex., $\mathbf{R}$ avec `\R`.

`\grastab` La macro `\grastab` prend également un seul argument obligatoire. Elle ne passe pas son argument en majuscule car certaines extensions fournissent aussi des minuscules en gras de tableau. Cependant elle assure le mode mathématique. On peut donc saisir « `et comme \grastab{M} est unifère` » pour obtenir « et comme $\mathbb{M}$ est unifère ».

2.5 Utilisation de `tdsfrmath.sty` avec `mathdesign.sty`

À la demande d'un utilisateur [2] de `mathdesign.sty` qui voulait pouvoir utiliser `tdsfrmath.sty` j'ai passé cette extension à la version 1.2 dans laquelle j'introduis un mécanisme plutôt simplissime pour assurer la cohabitation la plus harmonieuse possible entre ces deux extensions.

`avecmathdesign` La clé booléenne **`avecmathdesign`** vaut *`false`* à l'origine. Lorsqu'on lui donne, au chargement de `tdsfrmath.sty`, la valeur *`true`* — qui est sa valeur par défaut —, on n'a plus besoin de spécifier quoique ce soit concernant la police calligraphique car `tdsfrmath.sty` se repose entièrement sur `mathdesign.sty` qu'il faut charger explicitement avec `\usepackage` en lui passant les options adéquates comme l'explique le manuel — `mathdesign-doc.pdf` — de l'extension.

Cette option ne change rien au mécanisme de gestion du gras de tableau. C'est dû au fait agréable que la macro de `mathdesign.sty` qui crée le gras de tableau s'appelle également `\mathbb` ce que `tdsfrmath.sty` considère comme la valeur par défaut. Cela permet de répondre à moindre frais aux deux demandes de Maxime Chupin.

2.6 Des n-uplets, de leur saisie et de leur présentation

`nupletsep` La clé **`nupletsep`** peut prendre la valeur *`virgule`*, qui est le réglage par défaut, ou *`pointvirgule`*. Toute autre valeur provoque un avertissement et on se retrouve avec le réglage par défaut.

`virgule` Le réglage par défaut, *`virgule`* compose les n-uplets comme ceci : (a, b). Avec l'autre réglage possible *`pointvirgule`*, on aurait $(a\,;\,b)$. J'ai longtemps utilisé cette dernière [3] pour écrire des textes à destination des élèves du secondaire car on est souvent amené à utiliser des nombres décimaux et, dans ce cas, le mélange de virgule ne m'a jamais paru très heureux.

`\TdSMnuplet` La macro `\TdSMnuplet` prend un argument obligatoire qui est une liste dont les éléments sont séparés par des espaces. Avec « `\TdSMnuplet{a b c d}` » on obtient « a, b, c, d ». C'est une macro auxiliaire aussi lui ai-je donné un nom qui commence par `TdSM` mais elle peut reservir, directement ou dans la définition d'une commande dont je n'ai pas vu l'utilité, aussi je n'ai pas mis de `@` dans son nom.

`\nuplet` La macro *ordinaire* est `\nuplet`.Avec elle on obtient la présentation *classique*

2. Eh oui ! C'est encore Maxime Chupin. Si vous aussi, vous voulez voir votre nom dans cette documentation, n'hésitez pas à me faire part de vos désidératas. Sait-on jamais, vous pourriez motiver le passage à la version suivante `;-)`

3. Pour tout dire, à l'époque, mon fichier de macros ne ressemblait à celui-ci que de très loin mais on apprend avec l'age — au moins pendant un moment.

des n-uplets : `\nuplet{a b 3 c 8}` compose $(a, b, 3, c, 8)$.

La définition de `\TdSMnuplet` permet de coder `\nuplet{␣a␣b}` pour (a, b).

Bien entendu, comme d'habitude, on ne peut avoir le beurre et l'argent d'icelui. On peut coder `\nuplet{␣a\times␣b+c␣a\cap␣b}` pour obtenir $(a \times b + c, a \cap b)$ mais avec `\nuplet{␣a␣+␣b␣+␣c␣}` on aura $(a, +, b, +, c)$. Vous êtes prévenus ! `;-)`

`\EncloreExtensible` En fait, la présentation obtenue avec `\nuplet` repose sur `\EncloreExtensible` dont la syntaxe est :

`\EncloreExtensible[`⟨*md*⟩`]{`⟨*délim-gauche*⟩`}{`⟨*délim-droite*⟩`}{`⟨*texte*⟩`}`

⟨*md*⟩ vaut `1` par défaut, s'il vaut `0` les mathématiques sont composées en mode hors-texte — `\displaystyle`. Je sais bien que c'est une mauvaise pratique, que ça bousille l'interligne, que ça fiche en l'air l'uniformité du gris typo, &c. MAIS, hélas, parfois, c'est bien utile. Alors je le permets mais avec 0 qui rappelle ce qu'il faut penser d'une telle pratique `;-)`.

⟨*délim-gauche*⟩ est quelque chose qui peut être précédé de `\left` comme `(` ou `\Vert`, ⟨*délim-droite*⟩ est quelque chose qui peut être précédé de `\right` comme `)` ou `\Vert`, Si on ne veut rien de visible à gauche ou à droite, il faut que le 1er ou le 2e argument obligatoire soit un point. ⟨*texte*⟩ est ce qui sera placé entre les délimiteurs, en mode mathématique.

La macro `\EncloreExtensible` nous place en mode mathématique.

En voici un exemple un rien bête : `\EncloreExtensible{(}{\rangle}{x^{2}}` produit $(x^2\rangle$.

Le comportement des délimiteurs varient suivant qu'on est — de manière forcée par l'argument optionnel ou de manière *naturelle* parce que l'on est dans une formule composée hors-texte — en mode mathématique hors-texte ou en mode mathématique en ligne. Dans le 1er cas, les délimiteurs sont extensibles, dans le second ils ne le sont pas. On verra plus bas, page 11, le rendu des parenthèses dans la macro `\repere`.

Cette macro `\EncloreExtensible` me sert à en définir plusieurs autres que voici. Toutes ont la même syntaxe :

`\Macro[`⟨*md*⟩`]{`⟨*texte*⟩`}`

`\parent` `\accol` `\crochet` où ⟨*md*⟩ et ⟨*texte*⟩ ont le même sens que ci-dessus. Ce sont `\parent` pour obtenir des parenthèses, `\accol` pour des accolades et `\crochet` pour des ... oui, des crochets ! Voici, p. ex., `\parent{a}` qui produit (a) ; `\accol{\vide}` qui produit $\{\varnothing\}$; `\crochet{8\cdot9}` qui produit $[8 \cdot 9]$.

`\varabs` Dans le même genre, on a `\varabs` pour obtenir la valeur absolue comme ici : $|-12|$ codé `\varabs{-12}`.

`\norme` Dans la même veine, toujours, `\norme` pour écrire la norme comme suit : $\|\overrightarrow{i}\|$ codé `\norme{\vecti}`.

Revenons aux n-uplets. Les macros qui les produisent acceptent, elles aussi, toutes un argument optionnel qui force le mode hors-texte quand il vaut `0`.

`\nuplet` `\anuplet` On obtient, comme déjà vu ci-dessus, (c, d) avec `\nuplet{c␣d}` — j'insiste sur l'espace, non ? — et avec `\anuplet{c␣d}` on a $\{c, d\}$. Cette dernière doit son nom à ce qu'elle utilise des **a**ccolades.

Toutes les deux, comme je l'ai déjà signalé, peuvent traiter un nombre quelconque d'arguments séparés par des espaces comme, p. ex., $\{a, b, c, d, e, f\}$ obtenu avec `\anuplet{a b c d e f}`. Il faut toutefois remarquer que si l'on veut utiliser un macro à l'intérieur, p. ex. `\alpha`, il faudra la faire suivre ou l'entourer

d'une paire d'accolades pour préserver l'espace, sinon c'est l'erreur assurée et TEX proférera une de ces habituelles remarques absconces `;-)`

On codera donc `\nuplet{a \alpha{} \beta}` pour obtenir (a, α, β). Mais, coquèterie d'auteur, je me suis arrangé pour que l'on puisse coder directement `\nuplet{a §a §b}` pour avoir (a, α, β) lorsque l'extension `paresse.sty`, de votre serviteur, est chargée.

Avec `\nuplet{\frac{1}{2} \frac{3}{4}}` on produit $(\frac{1}{2}, \frac{3}{4})$.

Je ne suis pas allé plus loin car je pense que je couvre largement les besoins du secondaire avec tout ça. Qui voudrait obtenir une macro du même genre, pourra toujours la définir à l'aide de `\EncloreExtensible` et `\TdMnuplet` qui font le travail principal.

Cependant, je fournis la macro `\rnuplet`, prévue pour être utilisée dans le cas de l'écriture d'une fonction, p. ex. En effet, elle précède la composition du n-uplet d'une espace négative ce qui a pour effet de rapprocher la première parenthèse de ce qui précède. Comparer $f(x, y)$, obtenu avec `\(f\rnuplet{x y}\)`, à $f(x, y)$, `\(f\nuplet{x y}\)`, et à $f\ (x, y)$, `\(f\rnuplet{x y}[5]\)`.

`\rnuplet` Cette macro a pour syntaxe
`\rnuplet[`⟨*md*⟩`]{`⟨*texte*⟩`}[`⟨*écart*⟩`]`
le seul argument nouveau est ⟨*écart*⟩ qui règle l'espacement entre ce qui précède la macro et la parenthèse. Par défaut cet argument vaut `\TdSMReculParenthese` dont la valeur est -2, ⟨*écart*⟩ doit être un nombre.

Le `r` est là pour faire penser (?) à *recul*.

On pourrait donc écrire mais, bien sûr, on **ne le fera pas**, `\(f\rnuplet[0]{\frac{1}{2} 3}[10]\)` pour obtenir l'horreur : $f\ \left(\frac{1}{2}, 3\right)$.

`\TdSMReculParenthese` C'est la macro qui fixe, de manière générale, l'espace entre le texte qui précède et la parenthèse — ou délimiteur équivalent — ouvrante. On peut la redéfinir avec `\renewcommand`.

Remarque : Elle n'est pas *secrète* donc son nom ne comporte pas de `@` mais on n'est pas sensé l'utiliser toutes les trois secondes d'où les capitales. C'est la convention générale[4] de nommage des macros.

2.7 De la définition des ensembles

`SepDefEnsExt` *`true`* Je fournis la macro `\ensemble`, cf. page 15, qui permet d'écrire, p. ex., « $\{x \in \mathbb{R} \,/\, x^2 \geqslant 2\}$ » avec `\ensemble{x\in\R}{x^{2}\pgq 2}`. Le rendu en est controllé par la clé **`SepDefEnsExt`** — **sép**arateur de la **déf**inition d'un **ens**emble **ext**ensible — qui vaut *`true`* par défaut. Par ailleurs, la macro `\TdSMsepdefens` `\TdSMsepdefens` contient le séparateur et peut-être redéfinie à l'aide d'un `\renewcommand`.

Si, comme c'est le cas à l'origine et par défaut, la clé **`SepDefEnsExt`** vaut *`true`*, la définition de `\TdSMsepdefens` doit être *quelque chose* supportant l'action de `\middle` — qui est à un délimiteur central ce que `\left` et `\right` sont à ceux de gauche et droite — comme, p. ex., `\vert`. Ce qui fait que si l'on veut un séparateur qui ne supporte pas cela, comme `:`, les deux-points, il faut passer explicitement la valeur *`false`* à la clé **`SepDefEnsExt`**.

4. Il faut prendre ces conventions pour ce qu'elles sont et on n'aurait pas trop de peine à trouver des exceptions à cette *règle*, exceptions qui ne survivent que par la force de l'habitude.

2.8 Des noms des ensembles classiques

Il s'agit ici des macros qui permettent d'obtenir $\mathbb{R}$ et $\mathbb{Q}^*$ ou encore $\mathbb{C}_3[X]$.

`\TdSM@Decoration`

Cette macro *secrète* place les étoiles et signe plus ou moins, ce que j'appelle ici la décoration du nom de l'ensemble. Par défaut on a $\mathbb{R}_+^*$ mais cette disposition est controlée par la clé **ensdeco** qui peut prendre les valeurs ***ehsb***, ***ehsh***, ***sheh***, ***ebsb***, ***sbeb*** et ***ebsh***.

ensdeco
ebsh

Par défaut, on a `[ensdeco=ebsh]`. Toute autre valeur provoque un avertissement et on se retrouve avec le réglage par défaut.

La valeur par défaut ***ehsb*** place l'**é**toile en **h**aut et le **s**igne en **b**as. On pourra retenir que, quand l'ordre importe peu, on commence par l'étoile d'où ***ehsb*** et ***ebsh*** et que, sinon, l'ordre d'apparition de ***e*** et ***s*** règle la place de l'étoile $*$ et du signe.

Grace au mécanisme de `\define@choice*+` de l'extension `xkeyval`, on pourra passer les valeurs en capitales. Donc `[ensdeco=EHSB]` est une écriture valide.

`\TdSM@PlaceSigne`

Cette macro tout aussi *secrète* place le signe plus ou moins quand il est seul. Par défaut on a $\mathbb{R}^+$ mais cette disposition est controlée par la clé **placesigne** qui peut prendre les valeurs ***haut*** et ***bas***.

placesigne
haut

Par défaut, on a `[placesigne=haut]`. Toute autre valeur provoque un avertissement et on se retrouve avec le réglage par défaut.

`\EnsembleDeNombre`

N v1.3

Cette macro fait le gros boulot de composition. Elle prend 4 arguments obligatoires : le 1er donne la lettre majuscule symbolisant l'ensemble comme « R » pour $\mathbb{R}$; le comportement de la macro varie suivant que le 2e est égal à 1, *est égal à 0* [5], l'une de ces sept chaines de caractères `*`, `+`, `-`, `*+`, `+*`, `-*` et `-*` — ou *sinon suivant qu'il commence par un signe moins ou pas* [6]. C'est ce qui permet d'obtenir plus tard, p. ex., $\mathbb{Q}_+^*$ avec `\Q[+*]` —; le 3e argument est utilisé pour dénoter les ensembles de polynômes comme $\mathbb{C}_3[X]$ et dans ce cas le 2e doit commencer par un signe moins ; enfin le 4e doit être un entier qui donne le nombre de `mu` — unité de longueur spécifique au mode mathématique — qui séparent la majuscule du crochet ouvrant.

`\C`
`\N`
`\Q`
`\R`
`\Z`

Je rappelle au passage que `mu` — pour *maths unit* — est une unité de longueur définie uniquement en mode mathématique. Elle vaut 1/18 d'un `em` qui est — approximativement — la largeur d'un M dans la fonte courante.

N v1.3

Avec la version 1.3 j'opère un changement de doctrine ! Si le 2e argument n'est pas `*`, `+`, `-`, `*+`, `+*`, `-*`, `-*`, `0` ou `1`, la seule chose qui compte est de savoir s'il commence par un signe « − » ou pas. Ce qui fait que `\R[--4]` donne désormais le (douteux ?) $\mathbb{R}_{-4}[X]$. Toutefois, on a maintenant $\mathbb{R}^n$ avec `\R[n]` et $\mathbb{R}_m[X]$ avec `\R[-m]` et c'est bien cela dont j'avais besoin ces derniers temps ! On peut même laisser des blancs devant le signe moins : `\R[␣-3]` donne $\mathbb{R}_3[X]$ grâce à une réponse de Manuel Pégourié-Gonnard sur `fr.comp.text.tex`.

Comme il serait fastidieux d'avoir à taper `\EnsembleDeNombre{N}{1}{}{}` pour obtenir simplement $\mathbb{N}$, je fournis maintenant des commandes courtes auxquelles j'ai déjà fait allusion ci-dessus. Ce sont `\N` pour $\mathbb{N}$, `\Z` pour $\mathbb{Z}$, `\Q` pour $\mathbb{Q}$, `\R` pour $\mathbb{R}$, `\C` pour $\mathbb{C}$ et, enfin, si on a passé la valeur ***true*** à la clé **taupe**, `\K` pour $\mathbb{K}$.

5. Nouveauté de la version 1.3.
6. Nouveauté de la version 1.3.

Je ne fournis pas \D pour les décimaux car, d'une part, je doute finalement de l'utilité de cet ensemble et, d'autre part, je réserve cette macro pour plus tard.

J'utilise ici, avec beaucoup de satisfactions, l'extension xargs.sty afin que ces macros prennent deux arguments optionnels qui fourniront, dans l'ordre, les 2e et 3e arguments de \EnsembleDeNombre. Par défaut, le 1er argument vaut 1 et le 2e X.

Voici toutes les façons d'utiliser \R, p. ex., et ce qu'elles produisent :
- \R donne $\mathbb{R}$;
- \R[*] donne $\mathbb{R}^*$;
- \R[+] donne $\mathbb{R}^+$;
- \R[-] donne $\mathbb{R}^-$;
- \R[+*] ou \R[*+] donne $\mathbb{R}_+^*$;
- \R[-*] ou \R[*-] donne $\mathbb{R}_-^*$;
- \R[1] donne $\mathbb{R}$;
- \R[5] donne $\mathbb{R}^5$;
- \R[n] donne $\mathbb{R}^n$;
- \R[0] donne $\mathbb{R}[X]$;
- \R[-6] donne $\mathbb{R}_6[X]$;
- \R[-m] donne $\mathbb{R}_m[X]$;
- \R[-6][Y] donne $\mathbb{R}_6[Y]$.

On notera que l'on ne peut pas donner le 2e argument optionnel sans donner d'abord le premier.

Cependant, pour des raisons que l'on peut voir page 17 à propos de \suite*, je fournis quelques macros supplémentaires qui, du coup, peuvent abréger la saisie.

\R*
\N, \Z, \Q, \R et \C ont une forme étoilée qui fait la même chose que la macro avec une * pour 1er argument. On aura donc, p. ex., $\mathbb{N}^*$ avec \N* comme avec \N[*]. Il en est de même avec \K si taupe.sto est chargé.

\R+
\R-
\Q et \R ont une forme *plussée* et une forme *moinsée* qui font, respectivement, la même chose que la macro avec + et - pour 1er argument optionnel. On aura donc, p. ex., $\mathbb{Q}^+$ avec \Q+ comme avec \Q[+] et $\mathbb{R}^-$ avec \R- comme avec \R[-].

\R>
\R<
\R bénéficie de deux autres raccourcis, à savoir \R> qui produit $\mathbb{R}_+^*$ c.-à-d. la même chose que \R[+*] et \R< qui produit $\mathbb{R}_-^*$ c.-à-d. comme \R[-*]. Si l'on veut définir d'autres raccourcis du même genre on pourra regarder le code page 26.

Toutefois, si pour une raison quelconque, on voulait « $\mathbb{Q} + \mathbb{Q}$ » on devra coder \(\Q{} + \Q\). Il arrive que certaines épines aient des roses...

N v1.3
À la demande de Maxime Chupin, on dispose désormais de la macro \R/ qui donne $\overline{\mathbb{R}}$ et \R/+ qui donne $\overline{\mathbb{R}^+}$.

2.9 Des vecteurs, des bases et des repères

\definirvecteur
Cette macro permet, comme son nom l'indique presque, de définir des macros qui produisent des vecteurs. Sa syntaxe est :
\definirvecteur[⟨*bb*⟩]{⟨*a*⟩}{⟨*n*⟩}{⟨*m*⟩}

Avec \definirvecteur{⟨*a*⟩}{⟨*n*⟩}{⟨*m*⟩} on obtient une macro qui s'appelle \vecta et qui produit, en se plaçant dans le mode mathématique, la lettre a surmontée de la flèche que donne \overrightarrow avec un décalage de n mus devant et m mus derrière le texte.

L'argument optionnel ⟨*bb*⟩ permet d'obtenir le nom `\vectbb` ce qui est indispensable quand le 1er argument obligatoire est lui-même une macro comme p. ex. `\imath`.

Cette macro fait appel à `\TdSM@fairevecteur` décrite page 29.

`\redefinirvecteur` Elle teste l'existence d'une macro du même nom et produit une erreur s'il en existe déjà une. Si l'on veut redéfinir une commande comme `\vecti`, on utilisera `\redefinirvecteur` qui a la même syntaxe que sa grande sœur et qui, elle, produira une erreur si on tente de redéfinir un vecteur qui ne l'est pas encore.

`\vecti` `\vectj` `\vectk` `\vectu` `\vectv` Grâce à `\definirvecteur`, je définis quelques vecteurs courants et utiles : `\vecti` $\overrightarrow{\imath}$; `\vectj` $\overrightarrow{\jmath}$; `\vectk` $\overrightarrow{k}$; `\vectu` $\overrightarrow{u}$ et enfin `\vectv` $\overrightarrow{v}$.

On pourra comparer la composition que permet d'obtenir `\TdSM@fairevecteur`, à l'aide des 2e et 3e arguments qui définissent un nombre de `mus`, avec ce que donne une composition directe comme ici : $\overrightarrow{\imath}$ obtenu avec `\vecti` et $\overrightarrow{\imath}$ produit par `\(\overrightarrow\imath\)`.

`\vecteur` Cette macro peut être suivie par une étoile. Elle prend un argument optionnel, valant `1` par défaut, qui détermine l'espace placé devant le texte sous la flèche.

Elle prend un argument obligatoire qui donne le ⟨*texte*⟩ qui sera placé sous la flèche du vecteur. Avec la version sans étoile, le texte est composé *normalement* en mode mathématique comme dans $\overrightarrow{AB}$ produit par `\vecteur{AB}`. Avec la version étoilée le texte est en caractères romains, ou, plus exactement, est composé dans la police en vigueur pour l'argument de `\text` de l'extension `amstext.sty`, chargée ici par l'intermédiaire de `amsmath.sty`. On a donc $\overrightarrow{\text{CD}}$ avec `\vecteur*{CD}`.

Enfin, le troisième argument, optionnel, règle l'espace supplémentaire, toujours en nombre de `mus`, qui suit le texte. Comparez $\overrightarrow{AB}$ produit par `\vecteur{AB}` avec $\overrightarrow{\vphantom{AB}AB}$ produit par `\vecteur[10]{AB}`, $\overrightarrow{AB\vphantom{AB}}$ produit par `\vecteur{AB}[20]` et $\overrightarrow{\vphantom{AB}AB\vphantom{AB}}$ produit par `\vecteur[10]{AB}[20]`.

`\V` Ce n'est qu'un raccourci de `\vecteur`. Il a donc la même syntaxe :
`\V*[`⟨*espace-avant*⟩`]{`⟨*texte*⟩`}[`⟨*espace-après*⟩`]`
où ⟨*espace-avant*⟩ et ⟨*espace-après*⟩ sont des nombres.

`\base` La macro `\base` admet un seul argument, optionnel, qui ne doit prendre que les valeurs `1`, `2` — valeur par défaut — ou `3`. On obtient alors $(\overrightarrow{\imath})$ avec `\base[1]`, $(\overrightarrow{\imath}, \overrightarrow{\jmath})$ avec `\base` ou `\base[2]`, $(\overrightarrow{\imath}, \overrightarrow{\jmath}, \overrightarrow{k})$ avec `\base[3]`.

`\repere` La macro `\repere` fournit un repère à la française. Elle est construite sur `\base` et son 1er argument optionnel a le même rôle que celui de `\base`. Le 2e argument optionnel de `\repere` définit le centre du repère, c'est `O` par défaut.

On a donc $(O, \overrightarrow{\imath}, \overrightarrow{\jmath})$ ou $(O, \overrightarrow{\imath}, \overrightarrow{\jmath}, \overrightarrow{k})$ ou $(O', \overrightarrow{\imath}, \overrightarrow{\jmath})$ avec `\repere` ou `\repere[3]` ou `\repere[2][O']`. On a même $(O, \overrightarrow{\imath})$ avec `\repere[1]`.

Je rappelle qu'il faut le 1er argument optionnel si l'on veut préciser le 2e.

Voyons maintenant le rendu des repères dans une formule hors-texte :

$$(O, \overrightarrow{\imath}) \quad (O, \overrightarrow{\imath}, \overrightarrow{\jmath}) \quad \left(O, \overrightarrow{\imath}, \overrightarrow{\jmath}, \overrightarrow{k}\right)$$

Viennent maintenant des macros qui servent essentiellement d'abréviations.

\rog \ron \rond

Tout d'abord ce qu'il faut pour écrire « repère orthogonal $(O, \vec{\imath}, \vec{\jmath})$ » avec `\rog` puis « repère orthonormal $(O, \vec{\imath}, \vec{\jmath})$ » avec `\ron` et enfin « repère orthonormal direct $(O, \vec{\imath}, \vec{\jmath})$ » avec `\rond`.

Ces trois commandes acceptent les mêmes arguments que `\repere` ce qui fait que l'on peut obtenir « repère orthonormal direct $(A, \vec{\imath}, \vec{\jmath}, \vec{k})$ » avec `\rond[3][A]`. On **ne** doit **pas** les utiliser en mode mathématiques.

\repcom \roncom \rondcom

Je fais de même avec les repères pour le plan complexe, où, en général, on utilise $\vec{u}$ et $\vec{v}$ pour la base. On a donc « $(O, \vec{u}, \vec{v})$ » avec `\repcom`, « repère orthonormal $(O, \vec{u}, \vec{v})$ » avec `\roncom` et enfin « repère orthonormal direct $(O, \vec{u}, \vec{v})$ » avec `\rondcom`.

Je fournis de quoi écrire les repères à la mode du collège [7] mais je ne traite que le cas d'un repère du plan.

\Repere \Repere*

Cette macro a une forme étoilée. Sans étoile, on obtient « (O, I, J) » et avec l'étoile — c.-à-d. avec `\Repere*` — c'est « (O, I, J) ».

\Rog \Ron \Rond

Viennent ensuite des abréviations, construites sur le même modèle que les précédentes : `\Rog` pour « repère orthogonal (O, I, J) », `\Ron` pour « repère orthonormal (O, I, J) » et enfin `\Rond` pour « repère orthonormal direct (O, I, J) ». Elles ont toutes une forme étoilée qui permet d'obtenir les lettres « droites » — avec les mêmes remarques qu'à propos de `\vecteur*`, cf. page 11. On a donc, p. ex., « repère orthonormal direct (O, I, J) » avec `\Rond*`.

2.10 L'exponentielle

\E

La macro `\E` permet d'obtenir un « e » droit quelque soit l'environnement : « *Le nombre* e *vaut approximativement* 2,7 » codé `\emph{Le nombre \E vaut approximativement \(\np{2,7}\).}` grace à `\textup` mais il n'est en *romain* que si l'environnement est en romain : « *Le nombre* e *vaut approximativement* 2,7 » où j'ai utilisé `\textsl` pour obtenir des caractères sans empattements.

\eu

La macro `\eu` prend un argument obligatoire qui sera placé en exposant. On saisit `\eu{2x+3}` pour obtenir e^{2x+3}. Une fois encore, grace à `\ensuremath`, on n'a pas besoin de passer explicitement en mode mathématique.

2.11 Le nombre i

\I

Je définis `\I` pour qu'elle donne un « i » droit qui est ce que l'on devrait utiliser en français pour noter « la racine carrée de -1 » — pour parler comme les anciens.

On a donc « le nombre i qui vérifie $\mathrm{i}^2 = -1$ » avec le code « `le nombre \I qui vérifie \(\I^{2}=-1\)` ».

Les remarques faites ci-dessus à propos de `\E` s'appliquent également à `\I`.

2.12 Intégrales

$$\int_a^b f(x)\,\mathrm{d}x \qquad \text{plutôt que ça :} \qquad \int_a^b f(x)dx$$

7. Enfin, c'est comme cela que j'y pensais du temps où j'enseignais en lycée. Est-ce bien encore le cas ?

\FixeReculIntegrande
\FixeAvanceDx

Ces deux macros prennent un **nombre** pour unique argument obligatoire. Elles permettent de *fixer* le nombre de `mu`s dont l'intégrande sera rapproché du signe somme et celui dont l'intégrande et le dx seront séparés.

\D

Je fournis la macro `\D` avec un `\providecommand` car elle est déjà définie par `kpfonts.sty` de Christophe Caignaert, avec le même effet mais par un autre tour. Cela permet d'utiliser `kpfonts.sty` et `tdsfrmath.sty` sans craindre un conflit de nom.

\intgen

C'est la macro la plus générale pour écrire une intégrale. Sa syntaxe est :
`\intgen[`⟨*md*⟩`][`⟨*recul*⟩`]{`⟨*inf*⟩`} {`⟨*sup*⟩`}{`⟨*intégrande*⟩`}`
où ⟨*md*⟩ est le mode dans lequel sera composé la formule, par défaut le mode mathématique courant, valeur `1`, avec `0` on est en mode hors-texte — je ne fais pas de rappel sur ce qu'il faut penser de cette manœuvre `;-)`

⟨*recul*⟩ vaut par défaut `6mu` ou la valeur fixée par `\FixeReculIntegrande`, sinon ce doit être un nombre de `mu`s — explicitement on écrira `[1][-8mu]`, et je rappelle que le 2e argument optionnel ne peut être donné que si le 1er est donné également. ⟨*inf*⟩ et ⟨*sup*⟩ sont les bornes inférieure et supérieure de l'intégrale, ⟨*intégrande*⟩ est — surprise ! — l'intégrande.

On l'utilise lorsque l'intégrande et le dx sont *mélangés* comme dans

$$\int_2^5 \frac{\mathrm{d}x}{\ln x}$$

codé avec `\[\intgen{2}{5}{\dfrac{\D x}{\ln x}}\]`.

\integrer

Vient la macro pour le cas où l'intégrande est séparé de dx. Sa syntaxe est :
`\integrer[`⟨*md*⟩`][`⟨*recul*⟩`]{`⟨*inf*⟩`} {`⟨*sup*⟩`}{`⟨*integrande*⟩`}{`⟨*var*⟩`} [`⟨*avance*⟩`]`.

On retrouve les arguments de `\intgen` mais on trouve un argument obligatoire supplémentaire ⟨*var*⟩, qui est le symbole de la variable, et un argument optionnel final ⟨*avance*⟩ qui règle la distance entre l'intégrande et le `\D` ; ⟨*avance*⟩ est soumis aux mêmes règles que le ⟨*recul*⟩. Par défaut ⟨*avance*⟩ vaut `4mu`.

On code `\[\integrer{0}{\pi}{\cos 2x}{x}\]` pour avoir

$$\int_0^\pi \cos 2x \,\mathrm{d}x$$

\integrale

La macro suivante est construite sur `\integrer` mais est conçue pour être un raccourci de `\integrer{a}{b}{f(x)}{x}` avec `\integrale{a}{b}{f}{x}`.

À l'exception du 3e argument obligatoire qui est un *symbole* de fonction — comme f, g &c — tous ses arguments sont ceux de `\integrer`.

\intabfx

Enfin, raccourci du raccourci `\intabfx` remplace `\integrale{a}{b}{f}{x}` et compose $\int_a^b f(x)\,\mathrm{d}x$ dans le cours du texte et

$$\int_a^b f(x)\,\mathrm{d}x$$

en hors-texte.

2.13 Au bonheur du mathématicien, bazar

Je regroupe ici plusieurs macros qui me facilitent la vie dans la saisie des mathématiques. J'y fais une utilisation intense de `\ensuremath` et `\xspace`.

2.13.1 De l'infini

\plusinf
\moinsinf

J'ai mis très longtemps à retenir le nom de `\infty`, aussi je me suis fait `\plusinf`, $+\infty$, et `\moinsinf`, $-\infty$. J'espère que leurs seuls noms me dispense d'en dire plus sauf qu'il me faut préciser qu'elles assurent le mode mathématique et s'occupe de l'espace derrière ce qui permet d'écrire `et en \moinsinf on trouve` pour composer « et en $-\infty$ on trouve ».

2.13.2 Des intervalles de $\mathbb{R}$

\interff
\interoo
\interof
\interfo

On peut écrire les différents intervalles avec les macros `\interff`, `\interoo`, `\interof` et `\interfo`. Leur syntaxe commune est `\int...[`⟨*md*⟩`][`⟨*avant*⟩`]{`⟨*a b*⟩`} [`⟨*après*⟩`]`. On retiendra que `\inter` est mis pour `intervalle` puis que la première lettre donne le *sens* du crochet gauche et la suivante celui du crochet droit avec `f` pour *fermé* et `o` pour *ouvert*.

Le premier argument ⟨*md*⟩ est optionnel est règle le mode mathématique, il vaut `1` par défaut. Le 2e ⟨*avant*⟩, optionnel, vaut `0` par défaut et donne le nombre de `mu`s qui sépare le délimiteur ouvrant du texte. Le 4e et dernier ⟨*après*⟩, qui vaut `0` par défaut, est également optionnel. Il définit, en nombre de `mu`s, la distance qui sépare le texte intérieur du délimiteur fermant.

Le 3e argument ⟨*a b*⟩ est obligatoire, il fournit le texte à placer entre les délimiteurs. L'espace sépare les deux valeurs extrêmes de l'intervalle. On code donc `\interff{12␣37/5}` pour obtenir [12, 37/5] mais il faudra coder `\interoo {\moinsinf{}␣\plusinf}` pour avoir $]-\infty, +\infty[$.

Le séparateur des valeurs extrêmes de l'intervalle est soumis à la clé **`nupletsep`**.

2.13.3 La réserve du bazar, miscellanées

\mdfrac
\mfrac

Deux macros pour fainéant donc pour moi `;-)` : `\mdfrac` et `\mfrac` permettent de saisir les fractions comme si on utilisait `\(\dfrac{...}{...}\)` et `\(\frac{...}{...}\)` respectivement. On pourra donc coder `\mfrac{1}{2}` pour obtenir $\frac{1}{2}$.

\cnp

Il fut une époque où, en France, on ne notait pas le nombre de combinaisons comme dans le monde anglo-saxon, d'où `\cnp`. La tradition s'est perdue mais la macro est restée pour fournir la notation *nouvelle vague*. Avec `\cnp{n}{p}` on a $\binom{n}{p}$. Là encore il n'est pas nécessaire de passer explicitement en mode mathématique.

\dans
\donne

Deux abréviations pour écrire les définitions de fonctions. Je trouve que `\dans` est plus court et plus facile à retenir que `\longrightarrow` et qu'il en est de même de `\donne` vis-à-vis de `\longmapsto`. De fait `\(f\): \(\R \dans \R\); \(x \donne 2x\)` compose : « $f : \mathbb{R} \longrightarrow \mathbb{R}\,;\, x \longmapsto 2x$ » [8].

\vide

Je préfère $\varnothing$ à $\emptyset$ [9] et par paresse encore, je me suis fait une `\vide` qui permet de saisir `\vide et autre` pour obtenir « $\varnothing$ et autre ». Merci `\ensuremath` *et* `\xspace`.

\ppq
\pgq

Je veux ceci $0 \leqslant 1$ et $2 \geqslant 1$. Comme `\leqslant` et `\geqslant`, c'est pas beau, je me suis fait `\ppq` — **p**lus **p**etit **q**ue — et `\pgq` — **g**rand.

8. Début seconde, peut-être `;-)`

9. Celui-là je m'en sers tellement que j'avais oublié son nom. Ce n'est pas `\nothing` mais `\emptyset` pour faciliter le travail de la mémoire.

\ensemble La macro \ensemble a deux arguments obligatoires, elle sert à écrire la définition d'un ensemble comme « $\{x \in \mathbb{R} / f(x) \geqslant \frac{1}{2}\}$ » obtenu avec `\ensemble {x\in\R}{f(x)\pgq \frac{1}{2}}` et dont l'aspect est

$$\left\{ x \in \mathbb{R} \middle/ f(x) \geqslant \frac{1}{2} \right\}$$

en hors-texte, du fait de la présence de \middle, lorsque la clé booléenne **SepDefEnsExt** vaut *true* comme c'est le cas par défaut.

Sa syntaxe complète est :
`\ensemble[`⟨*avant*⟩`]{`⟨*1er texte*⟩`}{`⟨*2e texte*⟩`}[`⟨*après*⟩`]`
où ⟨*avant*⟩ et ⟨*après*⟩ doivent être des nombres. Leur valeur par défaut est 3. Ces arguments optionnels règlent la distance avant et après le symbole de séparation, en nombre de mus.

2.14 Pour les taupes, taupins et taupines

Les macros de cette section ne sont définies que si l'on a passé la valeur *true* à la clé **taupe**. Je ne pense pas que ces macros soient utiles avant le supérieur. Cependant aucun mécanisme n'est prévu pour s'assurer de la destination du document final ;-)

\K La macro \K donne $\mathbb{K}$, le corps bien connu, alias de $\mathbb{R}$ ou $\mathbb{C}$.

\prodscal Cette macro permet d'écrire le produit scalaire comme on le trouve assez souvent dans les bouquins pour taupins. Elle ne prend qu'un seul argument obligatoire qui est une liste dont les éléments sont séparés par des espaces. Les remarques formulées à propos de \nuplet s'appliquent donc ici.

Avec `\prodscal{u v}` on obtient « $\langle u, v \rangle$ » et avec `\prodscal{\vectu{} \vectv}` ou `\prodscal{{\vectu} \vectv}` on obtient « $\langle \overrightarrow{u}, \overrightarrow{v} \rangle$ ».

Je redéfinis quelques macros classiques pour leur donner un aspect français comme on le trouve encore souvent.

Ce sont les fonctions taupiques usuelles \sinh, \cosh, \tanh auxquelles j'ajoute \cot parce que j'ai dû en avoir besoin un jour.

On aura donc, p. ex., « $\operatorname{ch} x$ » en codant `\(\cosh x\)`.

Je crée les macros \argsh, \argch et \argth pour les fonctions hyperboliques réciproques. Par défaut elles ont l'aspect suivant : $\operatorname{argch} x$, $\operatorname{argsh} y$ et $\operatorname{argth} z$.

Si on a passé la valeur *true* à la clé **ArgArcMaj** alors je définis les macros \argsh, \argch et \argth pour qu'elles soient écrites avec une majuscule comme $\operatorname{Argsh} x$. Dans ce cas, je redéfinis également les macros \arccos, \arcsin et \arctan pour qu'elles aient le même aspect.

Table 1 – Macros redéfinies dans `taupe.sto`

\sinh	$\operatorname{sh} x$	\cosh	$\operatorname{ch} x$
\tanh	$\operatorname{th} x$	\cot	$\operatorname{cotan} y$

Pour noter le noyau et l'image avec une majuscule, je fournis \Ker — $\operatorname{Ker} f$ avec `\(\Ker f\)` à comparer à $\ker f$ avec `\(\ker f\)` — et \Img qui donne $\operatorname{Im} f$ avec `\(\Img f\)` — \Im est déjà prise pour noter la partie imaginaire d'un complexe.

TABLE 2 – Macros dont l'aspect dépend de la clé **ArgArcMaj** — aspect par défaut

`\arccos`	$\arccos x$	`\arcsin`	$\arcsin x$
`\arctan`	$\arctan x$	`\argsh`	$\operatorname{argsh} x$
`\argch`	$\operatorname{argch} x$	`\argth`	$\operatorname{argth} x$

`\tendversen` Pour écrire $f(x) \underset{0}{\longrightarrow} +\infty$, je fournis `\tendversen{`⟨*en*⟩`}` à utiliser **en mode mathématique**. J'ai codé `\(f(x)\tendversen{0} \plusinf\)` l'exemple ci-avant.

`\devlim` Je fournis `\devlim[`⟨*en*⟩`]{`⟨*ordre*⟩`}` à utiliser en mode mathématique pour obtenir $\mathrm{DL}_4(0)$ avec `\devlim{4}` et $\mathrm{DL}_7(1)$ avec `\devlim[1]{7}`. On remarque donc que ⟨*en*⟩ vaut `0` par défaut. N'est-ce pas étrange ?

`\parties` Pour écrire $\mathscr{P}(E)$, je fournis `\parties` utilisable en mode texte. Sa syntaxe est `\parties[`⟨*n*⟩`]{`⟨*ensemble*⟩`}` où *n* est un nombre de `mu`s qui permet de régler la distance entre $\mathscr{P}$ et la parenthèse ouvrante, *n* vaut `-2` par défaut ; *ensemble* est le nom de l'ensemble dont on considère l'ensemble des parties, étonnant, non ?

`\drv` Pour écrire « $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ », je fournis `\drv{`⟨*fonction*⟩`}{`⟨*var*⟩`}` utilisable en mode texte. J'ai codé `\drv{f(x)}{x}` l'exemple ci-dessus.

N v1.3 Cette macro admet désormais — depuis la version 1.3 — un argument optionnel avec `\drv[3]{f(x)}{x}` on obtient « $\frac{\mathrm{d}^3 f(x)}{\mathrm{d}x^3}$ ».

`\ddrv` `\ddrv` est à `\drv`, ce que `\dfrac` est à `\frac` et donc « `et \ddrv{f(x)}{x} vaut` » compose « et $\dfrac{\mathrm{d}f(x)}{\mathrm{d}x}$ vaut », en bousillant l'interligne comme prévu ! Elle admet également un argument optionnel qui joue le même rôle que celui de `\drv`.

`\interent` Avec `\interent{3 12}` on obtient $[\![3,\, 12]\!]$. Cette macro a pour syntaxe complète : `\interent[`⟨*md*⟩`][`⟨*avant*⟩`]{`⟨*n m*⟩`} [`⟨*après*⟩`]`, les arguments jouant le même rôle que ceux des macros pour intervalles, cf. page 14.

Sa définition utilise `\llbracket` et `\rrbracket` fournis par `stmaryrd.sty` chargé quand **taupe** vaut *true*.

`\interzn` Cette macro permet d'obtenir $[\![0,\, n]\!]$ avec `\interzn`. Sa syntaxe est dérivée de celle de `\interent` et prend les mêmes arguments *optionnels* avec la même signification. Sa syntaxe est donc `\interzn[`⟨*md*⟩`][`⟨*avant*⟩`][`⟨*après*⟩`]`.

Je consacre quelques lignes à la macro `\derpart` qui permet d'obtenir — et je passe en mode mathématique hors-texte pour l'occasion —

$$\frac{\partial^6 f(x,\, y,\, z)}{\partial x^2\, \partial y^3\, \partial z}$$

avec le code `\[\derpart{f\rnuplet{x y z}}{xxyyyz}\]`.

`\TdSMDerPartSepar` Cette macro contient ce qui sépare, p. ex., un ∂x^2 du ∂y qui le suit. Par défaut, elle est définie comme étant égale à `\,` ce qui, à mon sens, améliore le rendu. Mais on peut la redéfinir avec un coup de `\renewcommand`.

`\derpart` Comme on vient de le voir cette macro permet d'obtenir l'écriture de la dérivée partielle. Sa syntaxe est :
`\derpart{`⟨*dessus*⟩`}{`⟨*dessous*⟩`}`

où ⟨*dessus*⟩ est le texte qui sera composé à coté du ∂ au numérateur et ⟨*dessous*⟩ est une liste de lexèmes — à priori des lettres mais on peut y placer une macro comme `\alpha` en la faisant suivre d'un espace — qui formeront le dénominateur.

Cette macro assure le passage en mode mathématique si nécessaire.

Allez, encore un petit exemple, `\[\derpart{f}{xyz␣␣zz\alpha\alpha␣x} \]` compose

$$\frac{\partial^8 f}{\partial x\,\partial y\,\partial z^3\,\partial\alpha^2\,\partial x}$$

et on voit que l'on peut se permettre de placer des espaces inutiles `;-)`

Remarque : pour les utilisateurs de `paresse.sty`. On ne peut pas utiliser `§a` comme raccourci de `\alpha` dans le 2e argument de `\derpart`. Ça serait analysé comme `§` puis `a` ce qui n'est peut-être pas tout à fait ce que l'on veut.

2.15 Des suites pour le secondaire

Lorsque l'on passe la valeur *`true`* à la clé `suite`, on charge le fichier `suite.sto` qui donne accès à quelques macros concernant les suites.

N v1.3 Depuis la version 1.3, on obtient le chargement de `suite.sto` en écrivant simplement `\usepackage[suite]{tdsfrmath}`.

`\suite` La première macro `\suite` a pour syntaxe `\suite[`⟨*texte*⟩`]` et la valeur par défaut de ⟨*texte*⟩ est `u`. Elle assure le mode mathématique et on peut donc coder `\suite` pour avoir « (u_n) ».

`\suite*` La version étoilée a pour syntaxe `\suite*[`⟨*deco*⟩`][`⟨*texte*⟩`]` où ⟨*texte*⟩ a la même fonction que dans la version sans étoile et où ⟨*deco*⟩ vaut `\N` c.-à-d. $\mathbb{N}$ par défaut. On a donc « $(u_n)_{\mathbb{N}}$ » avec `\suite*`, « $(u_n)_{\mathbb{N}^*}$ » avec `\suite*[\N*]` et « $(w_n)_{\mathbb{N}}$ » avec `\suite*[\N][w]`.

LaTeX ne supporte pas les arguments optionnels imbriqués, les parenthèses dans `\suite*[{\N[*]}]` sont absolument indispensables, c'est pourquoi j'ai défini les macros étoilées, plussées et moinsées présentées en page 10.

`\suitar` La commande `\suitar` — `ar` pour *arithmétique* — a pour syntaxe
`\suitar[`⟨*texte*⟩`]{`⟨*raison*⟩`}[`⟨*rang*⟩`]{`⟨*prem*⟩`} [`⟨*entre*⟩`]`
où la valeur par défaut de ⟨*texte*⟩ est encore `u`, ⟨*raison*⟩ donne la valeur de la raison de la suite et ⟨*prem*⟩ la valeur du premier terme dont ⟨*rang*⟩ est le rang. Enfin ⟨*entre*⟩, qui vaut `{}` par défaut, compose le texte entre la suite et sa description.

En codant `\suitar{3}{5}`, on compose « (u_n) la suite arithmétique de raison $r = 3$ et de premier terme $u_0 = 5$ » et, avec `\suitar[w]{3}{5}`, on obtient « (w_n) la suite arithmétique de raison $r = 3$ et de premier terme $w_0 = 5$ », enfin, avec `\suitar{3}[1]{5}`, on obtient « (u_n) la suite arithmétique de raison $r = 3$ et de premier terme $u_1 = 5$ ».

Avec `\suitar{3}{5}[␣est]` on compose « (u_n) est la suite arithmétique de raison $r = 3$ et de premier terme $u_0 = 5$ », on fera attention à laisser un blanc devant le texte ici. Avec `\suitar{3}{5}[,]` on compose « (u_n), la suite arithmétique de raison $r = 3$ et de premier terme $u_0 = 5$ ».

`\suitgeo` La commande `\suitgeo` a la même syntaxe que `\suitar` mais cette fois elle concerne les suites géométriques. En codant `\suitgeo{3}{5}`, on compose « (u_n) la suite géométrique de raison $q = 3$ et de premier terme $u_0 = 5$ » et, avec `\suitgeo[w]{3}{5}`, on obtient « (w_n) la suite géométrique de raison $q = 3$ et de

premier terme $w_0 = 5$ », enfin, avec `\suitgeo[w]{3}[2]{5}`, on obtient « (w_n) la suite géométrique de raison $q = 3$ et de premier terme $w_2 = 5$ ».

`\suitar*` Les versions étoilées, `\suitar*` et `\suitgeo*` ont la syntaxe suivante :
`\suitar*[`⟨*deco*⟩`][`⟨*texte*⟩`]{`⟨*raison*⟩`}[`⟨*rang*⟩`] {`⟨*prem*⟩`}[`⟨*entre*⟩`]`
où on retrouve l'argument optionnel ⟨*deco*⟩ de `\suite*` avec la même signification, cf. page 17.

On a donc « $(v_n)_{\mathbb{N}^*}$ la suite arithmétique de raison $r = 3$ et de premier terme $v_1 = 9$ » avec le code `\suitar*[\N*][v]{3}[1]{9}`. On prendra garde au fait que la macro ne cherche pas à assurer la cohérence entre l'ensemble des indices et le rang du premier terme `;-)`

`suitedeco` Ce que je viens de décrire est le comportement par défaut de ces macros `\suite`, `\suitar`, `\suitgeo`, `\suite*`, `\suitar*` et `\suitgeo*`, comportement
`false` obtenu lorsque la clé `suitedeco` a la valeur *`false`*, sa valeur d'origine. Lorsque l'on passe la valeur *`true`*, sa valeur par défaut, à la clé `suitedeco` le comportement des macros avec et sans étoile est inversé.

Quitte à être un peu lourd, avec `suitedeco=false` on a « (u_n) » avec `\suite` et « $(u_n)_{\mathbb{N}}$ » avec `\suite*`. Avec `suitedeco=true` — ou simplement `suitedeco` — on a « $(u_n)_{\mathbb{N}}$ » avec `\suite` et « (u_n) » avec `\suite*`.

On fera attention que, si l'on a donné explicitement les premiers arguments optionnels de `\suitar`, p. ex., dans le cas où `suitedeco=true` on ne pourra pas tout bonnement repasser à `suitedeco=false` sans remplacer les formes sans étoiles par des formes étoilées et vice-versa en faisant, de plus, attention au 2$^{\text{e}}$ argument optionnel donnant le « nom » de la suite. Bref, on choisira une fois pour toute la forme de base et on s'y tiendra !

3 Récapitulatif

3.1 Extensions chargées

L'appel de `tdsfrmath` avec `\usepackage` entraine le chargement des extensions suivantes : `ifthen`, `xkeyval`, `amsmath`, `amssymb` — si `avecmathdesign` a la valeur *`false`* —, `xspace`, `xargs`, `suffix`, `xstring` et, si la clé `taupe` a la valeur *`true`*, `stmaryrd` .

Il n'est donc pas nécessaire de les appeler avec `\usepackage` dans le préambule d'un document chargeant `tdsfrmath`.

3.2 Options

Dans la table 3, page 19, je note « texte TEX » pour dire que la valeur passée à la clé doit être une chaine de *lettres* au sens de TEX, c.-à-d. les minuscules et majuscules non accentuées de l'ASCII comme on le trouve pour le nom des macros. Le « texte » tout court est ce qui sert à écrire les noms des extensions, on ne devrait donc pas y trouver de caractères *bizarres* mais on peut y voir des chiffres.

Table 3 – les clés de `tdsfrmath.sty`

clé	type	valeur		référence
		`d'origine`	`par défaut`	voir page
avecmathdesign	booléen	`false`	`true`	6
taupe	booléen	`false`	`true`	4
ArgArcMaj	booléen	`false`	`true`	4
suite	booléen	`false`	`true`	4
suitedeco	booléen	`false`	`true`	18
SepDefEnsExt	booléen	`true`	`true`	8
CharPoCal	booléen	`true`	`true`	4
calpack	texte	`mathrsfs`		4
calcomd	texte TEX	`mathscr`		4
caloptn	texte TEX	`***`		5
CharPoGdT	booléen	`false`	`true`	5
gdtpack	texte	`***`		5
gdtcomd	texte TEX	`***`		5
gdtoptn	texte TEX	`***`		5
placesigne	choix	`haut`		9
ensdeco	choix	`ebsh`		9
nupletsep	choix	`virgule`		6

4 Le code

4.1 Options et macros de service

Je commence par charger une extension bien utile tout de suite.

```
1 \RequirePackage{ifthen}
```

et une autre qui ne l'est pas moins.

```
2 \RequirePackage{xkeyval}
```

N v1.3 Depuis la version 1.3, je charge aussi

```
3 \RequirePackage{xstring}
```

`tdsfrmath.sty` est la première extension où je fais usage de `xkeyval.sty` et de sa gestion de clés pour les options de l'extension.

4.1.1 Séparateur des n-uplets

`\TdSM@separateur` Cette macro, déterminée par la clé **`nupletsep`**, fournit le séparateur utilisé dans l'écriture des n-uplets comme dans (a, b).

J'en profite pour signaler que toutes les macros « secrètes » de cette extension commence par `\TdSM@`.

Pour que, plus bas, `\ExecuteOptionsX` ait le bon gout de considérer les options de l'extension, il faut utiliser le 2e argument de `\define@choicekey` en lui passant le nom complet du fichier contenant l'extension.

Pour des raisons de tranquilité je regroupe toutes les clés définies au niveau de l'extension dans le *trousseau* — la documentation de `xkeyval` parle simplement d'un préfixe — `TdSM`, c'est la raison du 1er argument.

Le 3e argument contient le nom de la clé. Le 4e contient deux macros qui prendront pour la première la valeur passée à la clef si elle est valide et pour la deuxième un nombre valant −1 si la valeur est invalide sinon le numéro de la valeur dans la liste — aux valeurs séparées par des virgules — constituant le 5e argument. Les valeurs sont numérotées en partant de 0.

Le 6e argument contient du code qui est exécuté quand la valeur passée à la clé est valide. Ici, je me sers de `\nr` dans le `\ifcase` pour définir `\TdSM@separateur` congrument au choix de l'utilisateur : le 0e cas est celui de la *virgule*, &c.

Le 7e argument contient le code exécuté quand la valeur est invalide. Ici ça consiste en un avertissement — erreur non fatale — inscrit à la console et dans le `.log`.

Grace au `+` on peut utiliser un 7e argument. Grace au `*`, la vérification de la validité de la valeur passée à la clé est faite après que l'entrée et la liste ont été passée en minuscules. L'utilisateur peut donc saisir `[nupletsep=Virgule]` sans troubler l'extension.

À la sortie du traitement, on met `\val` à `\relax` pour éviter de le rendre inutilisable alors que l'on n'en a plus besoin. On fera la même chose avec `\nr` après l'exécution des options.

```
\newcommand{\TdSM@separateur}{\string,\,}
\define@choicekey*+[TdSM]{tdsfrmath.sty}{nupletsep}[\val\nr]%
  {virgule,pointvirgule}{%
    \ifcase\nr\relax
    \renewcommand{\TdSM@separateur}{\string,\,}\or
    \renewcommand{\TdSM@separateur}{\,\string;\,}\fi
  }{%
    \PackageWarningNoLine{tdsfrmath}{la clef <<nupletsep>> ne connait pas
      <<\val>>\MessageBreak <<nupletsep=virgule>> en vigueur}
  }
\let\val\relax
```

4.1.2 De l'aspect des noms des ensembles classiques

`\TdSM@Decoration` Cette macro place la décoration. Elle est réglée par **ensdeco**. C'est par la déclaration de cette clé que je commence. La macro `\TdSM@DecoChoix` sert à conserver la trace de la valeur passée à la clé. On s'en servira au moment de définir la macro `\TdSM@Decoration`

```
\define@choicekey*+[TdSM]{tdsfrmath.sty}{ensdeco}[\TdSM@DecoChoix\nr]%
  {ehsb,ebsb,ehsh,ebsh,sbeb,sheh}{%
    \PackageInfo{tdsfrmath}{Vous avez choisi \TdSM@DecoChoix\@gobble}
  }{%
    \PackageWarningNoLine{tdsfrmath}{la clef <<ensdeco>> ne connait pas
      <<\TdSM@DecoChoix>>\MessageBreak <<ensdeco=ehsb>> en vigueur}%
    \def\TdSM@DecoChoix{ehsb}}
```

`\TdSM@PlaceSigne` Cette macro place le signe seul. Elle est réglée par **placesigne**. La macro `\TdSM@PlaSiChoix` sert à conserver la trace de la valeur passée à la clé. On s'en servira au moment de définir la macro `\TdSM@PlaceSigne`

```
\define@choicekey*+[TdSM]{tdsfrmath.sty}{placesigne}[\TdSM@PlaSiChoix\nr]%
  {haut,bas}{%
    \PackageInfo{tdsfrmath}{Vous avez choisi \TdSM@PlaSiChoix\@gobble}
  }{%
```

```
    \PackageWarningNoLine{tdsfrmath}{la clef <<placesigne>> ne connait pas
      <<\TdSM@PlaSiChoix>>\MessageBreak <<placesigne=haut>> en vigueur}%
    \def\TdSM@PlaSiChoix{haut}}
```

4.1.3 Du choix de la police calligraphique

N v1.3

Le mécanisme de `xkeyval` crée la macro `\ifTdSM@CharPoCal` et ses sœurs `\TdSM@CharPoCaltrue` et `\TdSM@CharPoCalfalse`. Le 5e argument contient — depuis la version 1.3 — la valeur par défaut `true` ce qui permet à l'utilisateur d'obtenir le même comportement de `tdsfrmath.sty` avec `\usepackage[CharPoCal]{tdsfrmath}` qu'avec `\usepackage[CharPoCal = true]{tdsfrmath}`. Il en sera de même avec toutes les clés booléennes de cette extension.

On ne confondra pas cette valeur par défaut avec la valeur que la clé possède à l'exécution des options, autrement dit, la valeur de la clé lorsque l'extension est appelée avec un simple `\usepackage{tdsfrmath}`. Je parlerai alors de valeur *d'origine*.

```
\define@boolkey+[TdSM]{tdsfrmath.sty}[TdSM@]{CharPoCal}[true]{}%
  {\PackageWarningNoLine{tdsfrmath}{CharPoCal attend la valeur
      << true >> ou << false >>}}
```

Avec `\define@cmdkey[TdSM]{tdsfrmath.sty}[TdSM@] {calpack}[]{}` je définis la clé **`calpack`** et conjointement une macro `\TdSM@calpack` qui contiendra la valeur passée à la clé. Le préfixe de la commande est le contenu du 3e argument. La clé est *attachée au trousseau* `TdSM` grace au 1er argument et, une fois encore, la famille est obligatoire pour le fonctionnement ultérieur du `\ExecuteOptionsX` et `\ProcessOptionsX`, c'est l'objet du 2e argument.

Le dernier pourrait contenir du code mais je m'occupe de la valeur passée à la clé plus bas, *à la main* !

```
\define@cmdkey[TdSM]{tdsfrmath.sty}[TdSM@]{calpack}[]{}
\define@cmdkey[TdSM]{tdsfrmath.sty}[TdSM@]{calcomd}[]{}
\define@cmdkey[TdSM]{tdsfrmath.sty}[TdSM@]{caloptn}[]{}
```

4.1.4 Du choix du gras de tableau

```
\define@boolkey[TdSM]{tdsfrmath.sty}[TdSM@]{CharPoGdT}[true]{}
```

```
\define@cmdkey[TdSM]{tdsfrmath.sty}[TdSM@]{gdtpack}[]{}
\define@cmdkey[TdSM]{tdsfrmath.sty}[TdSM@]{gdtcomd}[]{}
\define@cmdkey[TdSM]{tdsfrmath.sty}[TdSM@]{gdtoptn}[]{}
```

4.1.5 Un peu plus pour les taupes

Avec la clé booléenne **`taupe`**, on charge le fichier `taupe.sto`. Cela permet de garder l'extension assez mince pour une utilisation dans le secondaire.

Si la valeur passée à la clé est `true` alors `\ifTdSM@taupe` est définie comme valant `true` ce qui entrainera plus bas le chargement de l'extension `stmaryrd.sty` qui fournit des symboles mathématiques dont celui dont je me sers pour écrire les intervalles de $\mathbb{N}$.

```
\define@boolkey[TdSM]{tdsfrmath.sty}[TdSM@]{taupe}[true]{}
```

Clé fixant l'aspect des fonctions circulaires et hyperboliques réciproques.

```
\define@boolkey[TdSM]{tdsfrmath.sty}[TdSM@]{ArgArcMaj}[true]{}
```

4.1.6 Des macros pour les suites

Avec la clé booléenne **suite**, on charge le fichier `suite.sto`.

```
\define@boolkey[TdSM]{tdsfrmath.sty}[TdSM@]{suite}[true]{}
\define@boolkey[TdSM]{tdsfrmath.sty}[TdSM@]{suitedeco}[true]{}
```

4.1.7 Séparateur de définition dans les ensembles

La clé booléenne **SepDefEnsExt**.

```
\define@boolkey[TdSM]{tdsfrmath.sty}[TdSM@]{SepDefEnsExt}[true]{}
```

4.1.8 Pour les utilisateurs de `mathdesign.sty`

La clé booléenne **avecmathdesign**.

```
\define@boolkey[TdSM]{tdsfrmath.sty}[TdSM@]{avecmathdesign}[true]{}
```

4.1.9 Exécutons les options

Il est temps d'exécuter les options par défaut. Puis de s'occuper de celles fournies par l'utilisateur. Pour finir, on relache une macro qui n'a plus d'usage.

```
\ExecuteOptionsX[TdSM]{%
  avecmathdesign=false,%
  taupe=false,%
  ArgArcMaj=false,%
  suite=false,%
  suitedeco=false,%
  nupletsep=virgule,%
  SepDefEnsExt=true,%
  placesigne=haut,%
  ensdeco=ehsb,%
  CharPoCal=true,calpack=mathrsfs,calcomd=mathscr,caloptn=***,%
  CharPoGdT=false,gdtpack=***,gdtcomd=***,gdtoptn=***}
\ProcessOptionsX[TdSM]\relax
\let\nr\relax
```

On charge maintenant les extensions nécessaires à cette extension. La moins connue, car la plus récente, est peut-être `xargs.sty` que l'on verra à l'œuvre plusieurs fois pour définir des macros acceptant plusieurs arguments optionnels.

```
\RequirePackage{amsmath}
```

On ne charge `amssymb.sty` que si on déclare ne pas utiliser `mathdesign.sty`, c'est le comportement par défaut.

```
\ifTdSM@avecmathdesign\else\RequirePackage{amssymb}\fi
\RequirePackage{xspace}
\RequirePackage{xargs}
\RequirePackage{suffix}
```

Si on a passé `[taupe=true]`, on charge `stmaryrd.sty`

```
\ifTdSM@taupe
\RequirePackage{stmaryrd}
```

puis, à la fin de l'extension, on inclut le fichier `taupe.sto` si on le trouve, sinon on grommèle.

```
\AtEndOfPackage{%
  \InputIfFileExists{taupe.sto}{%
    \PackageInfo{tdsfrmath}{fichier taupe.sto inclus\@gobble}}{%
    \PackageWarningNoLine{tdsfrmath}{fichier taupe.sto introuvable}}}
\fi
```

On traite la clé **`suite`** de même :

```
\ifTdSM@suite
\AtEndOfPackage{%
  \InputIfFileExists{suite.sto}{%
    \PackageInfo{tdsfrmath}{fichier suite.sto inclus\@gobble}}{%
    \PackageWarningNoLine{tdsfrmath}{fichier suite.sto introuvable}}}
\fi
```

On commence par définir `\TdSM@MathCalPol` dans le cas où **`avecmathdesign`** est *`true`*

```
\ifTdSM@avecmathdesign
\def\TdSM@MathCalPol{\mathscr}
\PackageInfo{tdsfrmath}{On compte sur mathdesign !\MessageBreak
  La police calligraphique est << mathscr >> \@gobble}
\else
```

dans le cas contraire on traite la clé **`CharPoCal`**. Si on a `CharPoCal=true`, on s'occupe de la police calligraphique

```
\ifTdSM@CharPoCal
  \PackageInfo{tdsfrmath}{La clef CharPoCal est vraie \@gobble}
```

on s'inquiète de l'existence de l'extension réclamée, on la charge, avec l'éventuelle option, si on la trouve

```
  \IfFileExists{\TdSM@calpack.sty}{%
    \ifthenelse{\equal{\TdSM@caloptn}{***}}%
      {\RequirePackage{\TdSM@calpack}}
      {\RequirePackage[\TdSM@caloptn]{\TdSM@calpack}}
    \def\TdSM@MathCalPol{\csname\TdSM@calcomd\endcsname}
```

et, une fois chargée l'extension, on teste l'existence de la commande demandée

```
    \@ifundefined{TdSM@MathCalPol}%
```

on grommèle si la macro est inconnue

```
    {\PackageWarningNoLine{tdsfrmath}{La macro
        << \TdSM@calcomd\space >> n'est pas connue\MessageBreak
        par l'extension \TdSM@calpack\space!\MessageBreak Revoyez
        la valeur de la clef << calcomd >> SVP}}%
```

ou on signale que tout s'est passé correctement.

```
    {\PackageInfo{tdsfrmath}{La police calligraphique est obtenue via
        << \TdSM@calcomd\space >>\@gobble}}}%
```

Vient le cas ou l'extension n'est pas présente : on grommèle et on définit la police calligraphique par défaut.

```
  {\PackageWarningNoLine{tdsfrmath}
    {Extension \TdSM@calpack\space pas vue sur la
      machine.\MessageBreak
      La police calligraphique est << mathcal >>}
    \def\TdSM@MathCalPol{\mathcal}}
```

Cela dit, il faut bien définir `\TdSM@MathCalPol` dans le cas où on la veut sortie de boite c.-à-d. égale à `\mathcal`.

```
\else
  \PackageInfo{tdsfrmath}{La clef CharPoCal est fausse.\MessageBreak
    La police calligraphique est << mathcal >> \@gobble}
  \def\TdSM@MathCalPol{\mathcal}
\fi
\fi
```

Si on a `CharPoGdT=true`, on s'occupe de la police du gras de tableau en employant la même méthode — et le même code `;-)` — avec la différence qu'il faut tenir compte du fait que, par défaut, `\TdSM@gdtcomd` contient `***` et que l'on peut — depuis la version 1.1 — définir une commande sans charger une extension supplémentaire.

On commence avec deux macros auxiliaires pour raccourcir le code suivant.

```
\newcommand\TdSM@DefinirGrasDefaut{%
  \def\TdSM@MathGdTPol{\mathbb}%
  \PackageInfo{tdsfrmath}{Gras de tableau obtenu par mathbb\@gobble}}
\newcommand\TdSM@SiDefaut[1]{\ifthenelse{\equal{#1}{***}}}
```

On profite lachement du fait que la commande par défaut de `mathdesign.sty` pour obtenir du gras de tableau est aussi `\mathbb` ce qui fait que l'on n'a pas à traiter différemment ici le cas avec `mathdesign.sty` du cas sans ladite extension.

On teste la clé booléenne `CharPoGdT`

```
\ifTdSM@CharPoGdT
\PackageInfo{tdsfrmath}{La clef << CharPoGdT >> est vraie\@gobble}
\TdSM@SiDefaut{\TdSM@gdtpack}
```

Si on n'a pas passé de valeur à la clef `gdtpack` on regarde ce qui l'en est pour la clef `gdtcomd` :

```
{\TdSM@SiDefaut{\TdSM@gdtcomd}
```

et si cette dernière clé n'est pas définie on demande à l'utilisateur de bien vouloir faire des choix cohérents !

```
  {\PackageWarningNoLine{tdsfrmath}
    {Je ne comprends pas ce que vous voulez !\MessageBreak
      Vous demandez une autre police de gras de tableau\MessageBreak
      sans donner ni extension (clef gdtpack)\MessageBreak
      ni commande (clef gdtcomd).\MessageBreak
      Revoyez la documentation SVP}
    \TdSM@DefinirGrasDefaut}
```

Sinon, on définit `\TdSM@MathGdTPol`

```
  {\def\TdSM@MathGdTPol{\csname\TdSM@gdtcomd\endcsname}
```

et on teste la disponibilité de cette commande.

```
    \@ifundefined{TdSM@MathGdTPol}%
```

Si elle n'est pas définie, on rouspète et on prend la valeur par défaut

```
   {\PackageWarningNoLine{tdsfrmath}
     {La macro << \TdSM@gdtcomd\space >> n'est pas connue !\MessageBreak
       Revoyez la valeur de la clef << gdtcomd >> SVP}
     \TdSM@DefinirGrasDefaut}%
```

sinon on informe, dans le `.log`, du choix effectué.

```
   {\PackageInfo{tdsfrmath}
```

```
    {Le gras de tableau est obtenu via << \TdSM@gdtcomd\space
      >>\@gobble}}}}
```

On passe au cas où la clé `gdtpack` a reçu une valeur

```
{\IfFileExists{\TdSM@gdtpack.sty}
```

On teste la présence de l'extension sur le système. Si le système est présent, on s'occupe de la clé `gdtoptn`.

```
  {\TdSM@SiDefaut{\TdSM@gdtoptn}
```

Si elle a la valeur par défaut, on charge l'extension sans option

```
    {\RequirePackage{\TdSM@gdtpack}}
```

sinon on passe l'option à l'extension.

```
    {\RequirePackage[\TdSM@gdtoptn]{\TdSM@gdtpack}}
```

On regarde la clé `gtdcomd`

```
    \TdSM@SiDefaut{\TdSM@gdtcomd}
```

si elle n'a pas reçu de valeur, on retombe dans le cas par défaut

```
    {\TdSM@DefinirGrasDefaut}
```

sinon on s'assure de la disponibilité de la commande demandée comme ci-dessus.

```
    {\def\TdSM@MathGdTPol{\csname\TdSM@gdtcomd\endcsname}
      \@ifundefined{TdSM@MathGdTPol}
      {\PackageWarningNoLine{tdsfrmath}
        {La macro << \TdSM@gdtcomd\space >> n'est pas connue\MessageBreak
          par l'extension \TdSM@gdtpack\space!\MessageBreak
          Revoyez la valeur de la clef << gdtcomd >> SVP.}}
     {\PackageInfo{tdsfrmath}
       {Le gras de tableau est obtenu via << \TdSM@gdtcomd\space
         >>\@gobble}}}}
```

On traite le cas où le fichier de style requis est introuvable. Dans ce cas on revient à la définition par défaut.

```
  {\PackageWarningNoLine{tdsfrmath}
    {Extension \TdSM@gdtpack\space pas vue sur la machine}
    \TdSM@DefinirGrasDefaut}}
```

On en a fini avec la première branche du si — cas où la clé `CharPoGdT` est vraie — et on passe à la 2e branche.

```
\else
\PackageInfo{tdsfrmath}{La clef << CharPoGdT >> est fausse\@gobble}
\TdSM@DefinirGrasDefaut
\fi
```

Les options étant traitées, `\TdSM@DecoChoix` contient la valeur passée à la clé **`ensdeco`**. On peut définir `\TdSM@Decoration` congrument.

```
\ifthenelse{\equal{\TdSM@DecoChoix}{ehsb}}%
  {\newcommand\TdSM@Decoration[2]{^{#1}_{#2}}}{%
\ifthenelse{\equal{\TdSM@DecoChoix}{sheh}}%
  {\newcommand\TdSM@Decoration[2]{^{#2#1}}}{%
\ifthenelse{\equal{\TdSM@DecoChoix}{ehsb}}%
  {\newcommand\TdSM@Decoration[2]{^{#1}_{#2}}}{%
\ifthenelse{\equal{\TdSM@DecoChoix}{ebsb}}%
  {\newcommand\TdSM@Decoration[2]{_{#1#2}}}{%
\ifthenelse{\equal{\TdSM@DecoChoix}{sbeb}}%
  {\newcommand\TdSM@Decoration[2]{_{#2#1}}}{%
```

```
\ifthenelse{\equal{\TdSM@DecoChoix}{ebsh}}%
  {\newcommand\TdSM@Decoration[2]{^{#2}_{#1}}}{%
}}}}}
\let\TdSM@DecoChoix=\relax
```

De même pour `\TdSM@PlaceSigne` :

```
\ifthenelse{\equal{\TdSM@PlaSiChoix}{haut}}%
  {\newcommand\TdSM@PlaceSigne[1]{^{#1}}}%
  {\newcommand\TdSM@PlaceSigne[1]{_{#1}}}
\let\TdSM@PlaSiChoix=\relax
```

4.2 Les noms des ensembles

Pour pouvoir écrire `\R` afin d'obtenir $\mathbb{R}$, il faut quelques macros auxiliaires par lesquelles je commence.

`\EnsembleDeNombre` La suivante fait le boulot de composition.

```
\newcommandx\EnsembleDeNombre[4]{%
  \ensuremath{%
    \grastab{#1}{%
    \edef\TdSM@arg{\@firstofone#2}%
    \ifthenelse{\equal{\TdSM@arg}{1}}{}{%
    \ifthenelse{\equal{\TdSM@arg}{0}}{\mspace{#4mu}\left[#3\right]}{%
    \ifthenelse{\equal{\TdSM@arg}{*}}{^{\ast}}{%
    \ifthenelse{\equal{\TdSM@arg}{+}}{\TdSM@PlaceSigne{+}}{%
    \ifthenelse{\equal{\TdSM@arg}{-}}{\TdSM@PlaceSigne{-}}{%
    \ifthenelse{\equal{\TdSM@arg}{+*}\or\equal{\TdSM@arg}{*+}}{%
      \TdSM@Decoration{\ast}{+}}{%
    \ifthenelse{\equal{\TdSM@arg}{-*}\or\equal{\TdSM@arg}{*-}}{%
      \TdSM@Decoration{\ast}{-}}{%
    \IfBeginWith{\TdSM@arg}{-}{%
      _{\StrBehind{\TdSM@arg}{-}}\mspace{#4mu}\left[#3\right]}{%
      ^{\TdSM@arg}}}}}}}}}\xspace}

\newcommandx\N[2][1=1,2=X]%
     {\EnsembleDeNombre{N}{#1}{#2}{\TdSMReculParenthese}}
\newcommandx\Z[2][1=1,2=X]%
     {\EnsembleDeNombre{Z}{#1}{#2}{\TdSMReculParenthese}}
\newcommandx\Q[2][1=1,2=X]%
     {\EnsembleDeNombre{Q}{#1}{#2}{\TdSMReculParenthese}}
\newcommandx\R[2][1=1,2=X]%
     {\EnsembleDeNombre{R}{#1}{#2}{\TdSMReculParenthese}}
\newcommandx\C[2][1=1,2=X]%
     {\EnsembleDeNombre{C}{#1}{#2}{\TdSMReculParenthese}}
```

David Kastrup ayant écrit récemment que la commande LaTeX `\@ifstar` qui teste la présence d'une étoile `*` derrière une commande n'était pas des plus recommandables, j'utilise son extension `suffix.sty` pour définir les commandes étoilées et, tant que j'y suis, les commandes plussées, moinsées et autres sur le même modèle.

```
\WithSuffix\newcommand\N*{\N[*]}
\WithSuffix\newcommand\Z*{\Z[*]}
\WithSuffix\newcommand\Q*{\Q[*]}
\WithSuffix\newcommand\R*{\R[*]}
```

```
\WithSuffix\newcommand\C*{\C[*]}
\WithSuffix\newcommand\Q+{\Q[+]}
\WithSuffix\newcommand\R+{\R[+]}
\WithSuffix\newcommand\Q-{\Q[-]}
\WithSuffix\newcommand\R-{\R[-]}
\WithSuffix\newcommand\R>{\R[+*]}
\WithSuffix\newcommand\R<{\R[-*]}
```

\R/

```
\DeclareRobustCommand{\TdSM@Rcomplet}{\ensuremath{\overline{\R}}}
\WithSuffix\newcommand\TdSM@Rcomplet+{\ensuremath{\overline{\R+}}}
\WithSuffix\newcommand\R/{\TdSM@Rcomplet}
```

Commençons par nous présenter puisque c'est la première fois que l'on rencontre ce fichier auxiliaire :

```
\ProvidesFile{taupe.sto}%
[\filedate\space\fileversion\space Pour tdsfrmath -- option taupe]
```

```
\newcommandx\K[2][1=1,2=X]%
     {\EnsembleDeNombre{K}{#1}{#2}{\TdSMReculParenthese}}
\WithSuffix\newcommand\K*{\K[*]}
```

4.3 Couples, paires, triplets etc.

Il s'agit maintenant de définir les macros qui permettent d'obtenir, p. ex., (a, b, c) avec `\nuplet{a␣b␣c}`. On aura noté que la délimitation des arguments est obtenue avec des espaces.

Là encore, je commence par une macro auxiliaire. Elle a pour tâche de récupérer une liste d'arguments délimités par des espaces et de fournir une liste de ces mêmes arguments séparés par le séparateur fixé par **`nupletsep`**. Elle est basée sur une macro que Manuel Pégourié-Gonnard, l'auteur de `xargs.sty`, m'a, très aimablement, fourni en réponse à une mienne question sur `fr.comp.text.tex`.

La seule « astuce » est l'utilisation de `#1#2␣#3`, ou l'équivalent ensuite, pour récupérer ce que je veux. Comme `#1` est immédiatement suivi de `#2`, TEX le considère comme non-délimité et, quand on utilise la macro, il prendra le premier lexème *différent d'un espace*. Le `#2` se chargera de récupérer alors tout ce qui suit jusqu'à l'espace suivant. Et s'il n'y a rien à récupérer il restera vide. Avec cette manœuvre, je permets de coder `\nuplet{␣a␣b}` pour obtenir (a, b).

```
\newcommand\TdSMnuplet[1]{\TdSM@nuplet #1 \@nil}
```

```
\def\TdSM@nuplet #1#2 #3{%
  \ifx\@nil #3%
  #1#2%
  \else
  #1#2\TdSM@separateur%
  \TdSM@nupletaux #3\fi}
```

```
\def\TdSM@nupletaux#1\fi{
  \fi\TdSM@nuplet#1}
```

\EncloreExtensible

Pour *passer* le mode à l'intérieur de la macro, en vue de prendre la bonne définition des délimiteurs, j'ai besoin d'un booléen.

```
\newboolean{TdSM@horstexterequis}
```

```
\setboolean{TdSM@horstexterequis}{false}
\newcommandx{\EncloreExtensible}[4][1=1]{%
  \ifthenelse{#1=0}%
  {\setboolean{TdSM@horstexterequis}{true}}%
  {\setboolean{TdSM@horstexterequis}{false}}%
```

J'ai réglé le cas de l'argument optionnel. Il faut voir si on ne serait pas en mode mathématique (`\ifmmode`) interne (`\ifinner`) c.-à-d. mathématique en ligne, ou, au contraire en mode mathématique hors-texte où il faut faire quelque chose :

```
  \ifmmode\ifinner\else
    \setboolean{TdSM@horstexterequis}{true}%
  \fi\else\fi
```

On ouvre un groupe et on s'assure d'être en mode mathématique et on agit en accord avec la valeur du booléen `TdSM@horstexterequis` qui contient le renseignement nécessaire :

```
  {\ensuremath{%
      \ifthenelse{\boolean{TdSM@horstexterequis}}%
      {\displaystyle\def\TdS@v@nt{\left #2}\def\TdS@pr@s{\right #3}}%
```

En hors-texte — naturel ou forcé — on a des délimiteurs extensibles,

```
      {\ifthenelse{\equal{#2}{.}}%
        {\def\TdS@v@nt{\relax}}%
        {\def\TdS@v@nt{\mathopen#2}}%
       \ifthenelse{\equal{#3}{.}}%
         {\def\TdS@pr@s{\relax}}
         {\def\TdS@pr@s{\mathclose#3}}}%
```

il n'en est rien en mode en ligne mais il faut tenir compte du délimiteur fantome donné par le point.

Et, pour finir, on compose le texte attendu.

```
      \TdS@v@nt #4 \TdS@pr@s}}
```

Avec `\EncloreExtensible`, je définis maintenant plusieurs macros usuelles.

```
\newcommandx{\parent}[2][1=1]{\EncloreExtensible[#1]{(}{)}{#2}}
\newcommandx{\accol}[2][1=1]{\EncloreExtensible[#1]{\{}{\}}{#2}}
\newcommandx{\crochet}[2][1=1]{\EncloreExtensible[#1]{[}{]}{#2}}
\newcommandx{\varabs}[2][1=1]{%
  \EncloreExtensible[#1]{\lvert}{\rvert}{#2}}
\newcommandx{\norme}[2][1=1]{\EncloreExtensible[#1]{\lVert}{\rVert}{#2}}
\newcommand{\nuplet}[2][1]{\parent[#1]{\TdSMnuplet{#2}}}
\newcommand{\anuplet}[2][1]{\accol[#1]{\TdSMnuplet{#2}}}
```

`\TdSMReculParenthese`

```
\newcommand\TdSMReculParenthese{-2}
```

`\rnuplet`

```
\newcommandx\rnuplet[3][1=1,3=\TdSMReculParenthese]{%
  \mspace{#3mu}\nuplet[#1]{#2}}
```

4.4 Vecteurs, bases et repères

`\definirvecteur`

```
\newcommand{\definirvecteur}[4][***]{%
```

```
  \ifthenelse{\equal{#1}{***}}%
  {\@ifundefined{vect#2}%
    {\expandafter\def\csname vect#2\endcsname{%
        \TdSM@fairevecteur{#2}{#3}{#4}\xspace}}%
    {\PackageError{tdsfrmath}%
      {Erreur il y a un vecteur de nom << #2 >>}%
      {Vous pouvez utiliser la macro << redefinivecteur >>
       \MessageBreak
       si c'est bien le nom que vous vouliez}}}%
  {\@ifundefined{vect#1}%
    {\expandafter\def\csname vect#1\endcsname{%
        \TdSM@fairevecteur{#2}{#3}{#4}\xspace}}%
    {\PackageError{tdsfrmath}%
      {Erreur il y a un vecteur de nom << #1 >>}%
      {Vous pouvez utiliser la macro << redefinivecteur >>
       \MessageBreak
       si c'est bien le nom que vous vouliez}}}}
\newcommand{\redefinirvecteur}[4][***]{%
  \ifthenelse{\equal{#1}{***}}%
  {\@ifundefined{vect#2}%
    {\PackageError{tdsfrmath}%
      {Erreur il n'y a pas de vecteur de nom << #2 >>}%
      {Vous pouvez utiliser la macro << definivecteur >>
       \MessageBreak
       si c'est bien le nom que vous vouliez}}
    {\expandafter\let\csname vect#1\endcsname=\relax%
     \definirvecteur[#1]{#2}{#3}{#4}}}%
  {\@ifundefined{vect#1}%
    {\PackageError{tdsfrmath}%
      {Erreur il n'y a pas de vecteur de nom << #1 >>}%
      {Vous pouvez utiliser la macro << definivecteur >>
       \MessageBreak
       si c'est bien le nom que vous vouliez}}
    {\expandafter\let\csname vect#1\endcsname=\relax%
     \definirvecteur[#1]{#2}{#3}{#4}}}}

\definirvecteur[i]{\imath}{-1}{3}
\definirvecteur[j]{\jmath}{0}{5}
\definirvecteur{k}{-1}{1}
\definirvecteur{u}{0}{3}
\definirvecteur{v}{0}{3}
```

La macro suivante fait le travail de composition.

```
\newcommand\TdSM@fairevecteur[3]{%
  \ensuremath{\overrightarrow{\mspace{#2mu}{#1}\mspace{#3mu}}}}
```

\vecteur J'utilise `suffix.sty` pour définir la commande `\vecteur*` après avoir défini `\vecteur`.

`\vecteur` et `\vecteur*` bénéficient eux aussi de `\newcommandx` qui permet l'utilisation de plusieurs arguments optionnels.

```
\newcommandx\vecteur[3][1=1,3=5]{\TdSM@fairevecteur{#2}{#1}{#3}}
```

Nous voyons que le code suivant est placé sous l'égide de Parmentier : réutilisation des restes `;-)`

```
\WithSuffix\newcommandx\vecteur*[3][1=1,3=5]{\vecteur{\text{#2}}}
```

\V

```
\newcommand{\V}{\vecteur}
```

La macro suivante gère les erreurs dans la macro d'après. C'est en français puisque cette extension est au bon gout de chez nous ;-) La seule difficulté c'est de ne pas mettre de lettre accentuée dans le message sinon ça donne des hyéroglyphes — enfin, presque — lors de l'affichage tant dans le fichier `.log` qu'à l'écran. Petit exercice de style !

```
\newcommand\TdSM@ErreurArgBase{%
  \PackageError{tdsfrmath}{Argument optionnel hors limites}{%
    L'argument optionnel vaut 2 par defaut\MessageBreak
    mais ne prend que les valeurs 1, 2 ou 3.}}
```

\base La macro `\base` admet un seul argument, s'il est nul ou s'il est plus grand que 3 la macro se plaint à l'aide de la macro précédente.

```
\newcommand{\base}[1][2]{%
  \ifcase #1\TdSM@ErreurArgBase\or
  \nuplet{\vecti}\or
  \nuplet{\vecti{} \vectj}\or
  \nuplet{\vecti{} \vectj{} \vectk}\or
  \TdSM@ErreurArgBase\fi\xspace}
```

\repere Reprend en le modifiant le code de `\base`.

```
\newcommandx{\repere}[2][1=2,2=O]{%
  \ifcase #1\TdSM@ErreurArgBase\or
  \nuplet{#2 \vecti}\or
  \nuplet{#2 \vecti{} \vectj}\or
  \nuplet{#2 \vecti{} \vectj{} \vectk}\or
  \TdSM@ErreurArgBase\fi\xspace}
```

Viennent maintenant des macros qui servent essentiellement d'abréviations. Comme je ne sais quel sera le codage choisi pour le document final, je place les accents à la TEX, ce qui assure la *portabilité*.

```
\newcommand{\rog}{rep\'ere orthogonal \repere}
\newcommand{\ron}{rep\'ere orthonormal \repere}
\newcommand{\rond}{rep\'ere orthonormal direct \repere}
\newcommand{\repcom}{\nuplet{O \vectu{} \vectv}\xspace}
\newcommand{\roncom}{rep\'ere orthonormal \repcom}
\newcommand{\rondcom}{rep\'ere orthonormal direct \repcom}
```

\Repere Pour la forme étoilée je reprends `\WithSuffix`.

```
\newcommand{\Repere}{\nuplet{O I J}}
\WithSuffix\newcommand\Repere*{\nuplet{\text{O} \text{I} \text{J}}}

\newcommand{\Rog}{rep\'ere orthogonal \Repere}
\newcommand{\Ron}{rep\'ere orthonormal \Repere}
\newcommand{\Rond}{rep\'ere orthonormal direct \Repere}
```

4.5 L'exponentielle

\E On peut penser que un `\DeclareMathOperator` pourrait suffire ici mais avec la définition choisie on peut utiliser la même macro dans le texte autant qu'en mode

mathématique, sans passer explicitement en mode math, cela me semble suffisant pour justifier l'effort de codage — si tant est que ce soit un effort ;-)

```
\newcommand*\E{\ensuremath{\textup{e}}\xspace}
```

\eu et la fonction

```
\newcommand*\eu[1]{\ensuremath{\E^{#1}}}
```

4.6 Le nombre i

\I Même opération qu'avec \E.

```
\newcommand\I{\ensuremath{\textup{i}}\xspace}
```

4.7 Intégrales

Je commence par définir des registres de dimensions mathématiques — dont on ne peut pas dire qu'on en abuse dans les différentes extensions de LaTeX — pour régler des distances à l'intérieur des intégrales. Leurs noms me semblent assez parlants.

```
\newmuskip\TdSM@reculintegrande
\newmuskip\TdSM@avancedx
\TdSM@reculintegrande=6mu
\TdSM@avancedx=4mu
```

Je donne à l'utilisateur de quoi les modifier globalement.

```
\newcommand\FixeReculIntegrande[1]{\TdSM@reculintegrande=#1mu}
\newcommand\FixeAvanceDx[1]{\TdSM@avancedx=#1mu}
```

Je fournis la macro \D avec un \providecommand cf. page 13 pour la raison.

```
\providecommand*\D{\textup{d}}
```

\intgen Vient ensuite la macro la plus générale.

```
\newcommandx{\intgen}[5][1=1,2=\the\TdSM@reculintegrande]{%
  \ensuremath{%
  \ifnum #1=0\displaystyle\fi
  \int_{#3}^{#4}\mspace{-#2}{#5}}}
```

\integrer

```
\newcommandx{\integrer}[7]%
[1=1,2=\the\TdSM@reculintegrande,7=\the\TdSM@avancedx]{%
  \ensuremath{%
  \ifnum #1=0\displaystyle\fi
  \int_{#3}^{#4}{\mspace{-#2}{#5}\mspace{#7}\D{#6}}}}

\newcommandx{\integrale}[7]%
[1=1,2=\the\TdSM@reculintegrande,7=\the\TdSM@avancedx]{%
  \integrer[#1][#2]{#3}{#4}{#5(#6)}{#6}[#7]}
\newcommandx{\intabfx}[3]%
[1=1,2=\the\TdSM@reculintegrande,3=\the\TdSM@avancedx]{%
  \integrale[#1][#2]{a}{b}{f}{x}[#3]\xspace}
```

4.8 Au bonheur du mathématicien, bazar

```
\newcommand{\plusinf}{\ensuremath{+\infty}\xspace}
\newcommand{\moinsinf}{\ensuremath{-\infty}\xspace}
```

La macro qui suit est une macro de service, elle place le texte à l'intérieur des délimiteurs des intervalles et assimilés.

```
\newcommand{\TdSM@dedans}[3]{\mspace{#1mu}\TdSMnuplet{#2}\mspace{#3mu}}
```

Je m'en sers dans les quatre macros suivantes.

\interff La version précédente des macros `\inter..` était erronée car la valeur par défaut du premier argument, optionnel, était fixée à `0`. Il faut `1` car, par défaut, on ne veut pas forcer le style hors texte.

```
\newcommandx{\interff}[4][1=1,2=1,4=0]%
  {\crochet[#1]{\TdSM@dedans{#2}{#3}{#4}}}

\newcommandx{\interoo}[4][1=1,2=1,4=0]{%
  \EncloreExtensible[#1]{]}{]}{\TdSM@dedans{#2}{#3}{#4}}}
\newcommandx{\interof}[4][1=1,2=1,4=0]{%
  \EncloreExtensible[#1]{]}{[}{\TdSM@dedans{#2}{#3}{#4}}}
\newcommandx{\interfo}[4][1=1,2=1,4=0]{%
  \EncloreExtensible[#1]{[}{]}{\TdSM@dedans{#2}{#3}{#4}}}
\newcommandx{\intferab}[2][1=1,2=0]{\interff[#1]{a b}[#2]}
```

Et maintenant quelque chose de complètement différent :

```
\newcommand{\mdfrac}[2]{\ensuremath{\dfrac{#1}{#2}}}
\newcommand{\mfrac}[2]{\ensuremath{\frac{#1}{#2}}}
```

Les fameuses polices :

```
\newcommand*{\manus}[1]{%
  \ensuremath{\TdSM@MathCalPol{\MakeUppercase{#1}}}}
\newcommand*{\grastab}[1]{%
  \ensuremath{\TdSM@MathGdTPol{#1}}}
```

Des abréviations qu'elles sont utiles :

```
\newcommand*{\cnp}[2]{\ensuremath{\binom{#1}{#2}}}
\newcommand*{\dans}{\longrightarrow}
\newcommand*{\donne}{\longmapsto}
\newcommand*{\vide}{\ensuremath{\varnothing}\xspace}
\newcommand*{\ppq}{\ensuremath{\leqslant}}
\newcommand*{\pgq}{\ensuremath{\geqslant}}
```

et pour écrire les définitions des ensembles

```
\newcommand*{\TdSMsepdefens}{/}
\ifTdSM@SepDefEnsExt
\newcommandx*{\ensemble}[4][1=3,4=3]{%
  \accol{{#2}%
    \mspace{#1mu}%
    \ifthenelse{\boolean{TdSM@horstexterequis}}{\middle}{}%
    \TdSMsepdefens
    \mspace{#4mu}%
    {#3}}}
\else
\newcommandx*{\ensemble}[4]{%
  \accol{{#2}\mspace{#1mu}\TdSMsepdefens\mspace{#4mu}{#3}}}
\fi
```

4.9 Le fichier `taupe.sto`

Ce qui suit n'est chargé, et donc défini, que si l'on a passé la valeur *true* à la clé **taupe**.

```
\newcommandx{\prodscal}[4][1=1,2=1,4=1]{%
  \EncloreExtensible[#1]{\langle}{\rangle}{\TdSM@dedans{#2}{#3}{#4}}}
```

On pourra réutiliser l'astuce : pour redéfinir une macro créée avec des moyens LaTeXiens, il faut d'abord la rendre équivalente — par `\let` — à `\relax`. La redéfinition devient alors possible avec les moyens LaTeXiens classiques.

Ici, cependant, pour me faciliter la tâche, j'écris une macro auxiliaire `\TdSM@ReDeclareMathOperator` qui fait bien ce que je demande avec toutefois un léger désavantage : elle ne vérifie pas que la macro « redéfinie » existe bien au préalable. Du fait de ce fonctionnement pas très canonique, elle restera cachée `;-)`

```
\newcommand{\TdSM@ReDeclareMathOperator}{%
  \@ifstar{\@redeclmathop m}{\@redeclmathop o}}
\long\def\@redeclmathop#1#2#3{%
  \let#2=\relax%
  \DeclareRobustCommand{#2}{\qopname\newmcodes@#1{#3}}}
```

et je m'en sers

```
\TdSM@ReDeclareMathOperator{\cosh}{ch}
\TdSM@ReDeclareMathOperator{\sinh}{sh}
\TdSM@ReDeclareMathOperator{\tanh}{th}
\TdSM@ReDeclareMathOperator{\cot}{cotan}
```

et ensuite uniquement si **ArgArcMaj** vaut *true*

```
\ifTdSM@ArgArcMaj
\TdSM@ReDeclareMathOperator{\arccos}{Arccos}
\TdSM@ReDeclareMathOperator{\arcsin}{Arcsin}
\TdSM@ReDeclareMathOperator{\arctan}{Arctan}
\fi
```

et j'ajoute de nouvelles définitions.

```
\ifTdSM@ArgArcMaj
\DeclareMathOperator{\argch}{Argch}
\DeclareMathOperator{\argsh}{Argsh}
\DeclareMathOperator{\argth}{Argth}
\else
\DeclareMathOperator{\argch}{argch}
\DeclareMathOperator{\argsh}{argsh}
\DeclareMathOperator{\argth}{argth}
\fi

\DeclareMathOperator{\Ker}{Ker}
\DeclareMathOperator{\Img}{Im}

\newcommand*{\tendversen}[1]{\xrightarrow[#1]\relax}

\newcommand*{\devlim}[2][0]{\ensuremath{\text{DL}_{#2}{\parent{#1}}}}

\newcommand*{\parties}[2][-2]{%
  \ensuremath{\manus{p}\mspace{#1mu}\parent{{#2}}}}
```

\drv

```
\newcommand{\drv}[3][***]{\ensuremath{%
    \frac{\D\ifthenelse{\equal{#1}{***}}{}{^{#1}}#2}%
```

```
    {\D#3\ifthenelse{\equal{#1}{***}}{}{^{#1}}}}}
\newcommand{\ddrv}[3][***]{\ensuremath{\displaystyle\drv[#1]{#2}{#3}}}

\newcommandx{\interent}[4][1=1,2=2,4=2]{%
  \EncloreExtensible[#1]{\llbracket}{\rrbracket}%
                          {\TdSM@dedans{#2}{#3}{#4}}}
\newcommandx{\interzn}[3][1=1,2=2,3=2]{\interent{0 n}\xspace}
```

4.10 Dérivées partielles

`\TdSMDerPartSepar` Cette macro contient ce qui sépare, p. ex., un ∂x^2 du ∂y qui le suit. Par défaut, elle est définie comme étant égale à `\,` ce qui, à mon sens, améliore le rendu. Mais on peut la redéfinir avec un coup de `\renewcommand`.

```
\newcommand\TdSMDerPartSepar{\,}
```

`\derpart` Je vais maintenant tenter d'expliquer clairement la construction de la macro `\derpart`.

Attention : je n'ai aucune prétention à fournir ici un code optimisé. Je l'ai testé, il fait ce que je veux qu'il fasse et « sam suffi ».

Remarque : ce code est basé une fois encore sur celui fourni par Manuel Pégourié-Gonnard pour traiter les noms de fichiers.

Pour commencer, on notera que la définition de `\derpart` est entourée d'une double paire de parenthèses `{{...}}`. Cela assure que les macros définies à l'intérieur resteront inconnues au niveau du document et que les compteurs utilisés retrouveront leur état antérieur au sortir de la macro. Bref on utilise la capacité de TEX à « localiser » définitions et modifications de compteurs et macros.

J'utilise `\count@` et `\count0` parce qu'ils existent déjà et que ça m'évite d'en créer deux exprès. On pourra se reporter à la documentation sur le noyau de LaTeX 2_ε, `source2e.pdf` disponible avec toutes les bonnes distributions de TEX.

`\count0` contient le nombre de lettres identiques successives trouvées à l'itération considérée. `\count@` vaut à la fin le nombre total de caractères contenus dans la chaine constituant le 2e argument de la macro `\derpart`.

La macro `\TdSM@sentinelle` a la valeur `@@` et la garde tout le temps. La macro `\TdSM@precedent` commence l'aventure avec la même valeur. Cela permet de tester le début de la lecture du 2e argument que j'appelerai désormais *la chaine*.

Au début du jeu, les macros `\TdSM@DenomAux` et `\TdSM@Denom` sont vides. À la fin `\TdSM@Denom` contient le dénominateur de la dérivée partielle, passé en 2e argument de `\frac`. La macro `\TdSM@DenomAux`, quant à elle, permet de construire `\TdSM@Denom` en me facilitant la tâche à la fin de la chaine.

```
\newcommand\derpart[2]{{%
    \count@=1
    \def\TdSM@sentinelle{@@}%
    \def\TdSM@precedent{@@}%
    \def\TdSM@DenomAux{}%
    \def\TdSM@Denom{}%
```

`\TdSM@Puissance` place un exposant — il viendra derrière le caractère — si cet exposant est supérieur à 1.

```
    \def\TdSM@Puissance{%
      \ifnum\count0>1 ^{\the\count0}\fi}%
```

`\TdSM@FaireDenom` sert à créer le dénominateur. Comme sa définition est donnée à l'intérieur d'une définition — celle de `\derpart` — il faut doubler les *dièses* `#` dénotant les arguments. J'utilise `\edef` pour forcer le développement immédiat de `\TdSM@Denom` dans la définition même de `\TdSM@Denom`. Supposons, pour faire simple, que `\TdSM@Denom` contienne `tralala` au moment où TEX arrive sur `\TdSM@FaireDenom{truc}`. Après l'exécution de cette commande, `\TdSM@Denom` se développe en `tralala \mathchar"140 truc`.

En utilisant un simple `\def` à la place du `\edef`, on partait dans une de ces boucles infinies dont TEX se sort en invoquant la finitude de sa mémoire.

```
\def\TdSM@FaireDenom##1{%
  \edef\TdSM@Denom{\TdSM@Denom \partial ##1}}
```

Voici maintenant la macro `\TdSM@derpartaux` qui fait la plus grosse partie du boulot.

```
\def\TdSM@derpartaux##1##2{%
  \ifx\@nil##2%
```

Voici ce qui arrive lorsque le 2e argument vaut `\@nil` c.-à-d. lorsque l'on arrive à la fin de la chaine : on place la valeur du 1er argument `##1` dans `\TdSM@actuel`.

```
    \def\TdSM@actuel{##1}%
```

On regarde s'il a la même valeur que le précédent, placé dans `\TdSM@precedent`.

```
    \ifx\TdSM@actuel\TdSM@precedent
```

Si oui, on incrémente le compteur 0 et on doit tenir compte de la puissance

```
      \advance\count0 by 1
      \TdSM@FaireDenom{\TdSM@precedent\TdSM@Puissance}%
    \else
```

sinon on regarde si la valeur précédente est égale à la sentinelle

```
      \ifx\TdSM@precedent\TdSM@sentinelle
```

auquel cas `\TdSM@actuel` contient l'unique lettre de la chaine et on peut écrire le dénominateur.

```
        \TdSM@FaireDenom{\TdSM@actuel}%
      \else
```

Si ce n'est pas le cas il faut placer la lettre précédente avec la bonne puissance puis placer l'actuelle :

```
        \TdSM@FaireDenom{%
          \TdSM@precedent\TdSM@Puissance
          \TdSMDerPartSepar\partial\TdSM@actuel}%
      \fi
    \fi
```

Dans le cas où le 2e argument ne vaut pas `\@nil` c'est que l'on n'est pas encore au bout.

```
  \else
  \def\TdSM@actuel{##1}%
```

On tient compte du cas où le `\TdSM@precedent` est égal à la sentinelle, auquel cas on met le compteur à 0.

```
  \ifx\TdSM@precedent\TdSM@sentinelle
    \count0=1
  \else
```

Sinon, soit la lettre actuelle est égale à la précédente et on incrémente le compteur 0

```
        \ifx\TdSM@actuel\TdSM@precedent
           \advance\count0 by 1
        \else
```

soit elle ne l'est pas et on place dans le dénominateur la lettre précédente à la bonne puissance et on remet le compteur 0 à zéro.

```
          \TdSM@FaireDenom{%
            \TdSM@precedent\TdSM@Puissance\TdSMDerPartSepar}%
          \count0=1
        \fi
      \fi
```

Enfin, on incrémente le compteur `@` général, on place la lettre actuelle dans `\TdSM@precedent` et on appelle la macro `\TdSM@derpart@continue` sur le 2^{e} argument. L'astuce ici est que le `\fi` ferme le premier `\if` et que, dans le même temps, il délimite l'argument de la macro `\TdSM@derpart@continue`

```
      \advance\count@ by 1
      \let\TdSM@precedent\TdSM@actuel%
      \TdSM@derpart@continue##2%
    \fi}%
```

comme on le voit dans sa définition qui consiste à replacer un `\fi` et rappeler `\TdSM@derpartaux` sur ce qui suit :

```
\def\TdSM@derpart@continue##1\fi{\fi\TdSM@derpartaux##1}%
```

Maintenant que `\TdSM@derpartaux` est définie, on s'en sert sur le 2^{e} argument de `\derpart` en plaçant le `\@nil` qui signale la fin de la chaine puis on compose la fraction en assurant le mode mathématique.

```
\TdSM@derpartaux#2\@nil%
\ensuremath{\frac%
              {\partial\ifnum\count@>1^{\the\count@}\fi #1}%
              {\TdSM@Denom}}%
}}

\ProvidesFile{suite.sto}%
[\filedate\space\fileversion\space Pour tdsfrmath -- option suite]
```

`\suite`

```
\ifTdSM@suitedeco
\newcommandx\suite[2][1=\N,2=u]{\ensuremath{\parent{{#2}_{n}}_{#1}}\xspace}
\newcommandx\suitar[6][1=\N,2=u,4=0,6={}]{%
  \suite[#1][#2]#6 la suite arithm\'etique de raison %
  $r = #3$ et de premier terme $#2_{#4}=#5$}
\newcommandx\suitgeo[6][1=\N,2=u,4=0,6={}]{%
  \suite*[#1][#2]#6 la suite g\'eom\'etrique de raison %
  $q = #3$ et de premier terme $#2_{#4}=#5$}
\WithSuffix\newcommand\suite*[1][u]{\parent{{#1}_{n}}\xspace}
\WithSuffix\newcommandx\suitar*[5][1=u,3=0,5={}]{%
  \suite*[#1]#5 la suite arithm\'etique de raison %
  $r = #2$ et de premier terme $#1_{#3}=#4$}
\WithSuffix\newcommandx\suitgeo*[5][1=u,3=0,5={}]{%
  \suite*[#1]#5 la suite g\'eom\'etrique de raison %
  $q = #2$ et de premier terme $#1_{#3}=#4$}
\else
```

```
\newcommand\suite[1][u]{\parent{{#1}_{n}}\xspace}
\newcommandx\suitar[5][1=u,3=0,5={}]{%
  \suite[#1]#5 la suite arithm\'etique de raison %
  $r = #2$ et de premier terme $#1_{#3}=#4$}
\newcommandx\suitgeo[5][1=u,3=0,5={}]{%
  \suite[#1]#5 la suite g\'eom\'etrique de raison %
  $q = #2$ et de premier terme $#1_{#3}=#4$}
\WithSuffix\newcommandx\suite*[2][1=\N,2=u]{%
  \ensuremath{\suite[#2]_{#1}}\xspace}
\WithSuffix\newcommandx\suitar*[6][1=\N,2=u,4=0,6={}]{%
  \suite*[#1][#2]#6 la suite arithm\'etique de raison %
  $r = #3$ et de premier terme $#2_{#4}=#5$}
\WithSuffix\newcommandx\suitgeo*[6][1=\N,2=u,4=0,6={}]{%
  \suite*[#1][#2]#6 la suite g\'eom\'etrique de raison %
  $q = #3$ et de premier terme $#2_{#4}=#5$}
\fi
```

Et ici se termine l'extension `tdsfrmath`.
Scripsit TdS.

Remerciements

Je tiens à remercier Maxime Chupin pour l'aide qu'il m'a apportée pour améliorer `tdsfrmath.sty`.

Je remercie également Manuel Pégourié-Gonnard pour les diverses réponses intelligentes et utiles à mes questions qui ne le furent pas toujours et Christian Tellechea dont j'utilise le `xstring.sty` et dont je n'oublie pas qu'il m'a envoyé du code qui fera son apparition dans la prochaine version.

Liste des tableaux

Changements

v1

Général : 1re version publique. . . . 1

v1.1

Général : Changement du mécanisme de définition du gras de tableau pour permettre l'utilisation d'un gras « normal ». . . 1

Correction de l'avertissement éronné en cas de chargement d'une police pour le gras de tableau. 1

Correction de l'oubli de la clé CharPoGdT dans l'exemple de chargement de dsfont 1

v1.2

Général : Création de la clé `avecmathdesign` pour permettre l'utilisation de mathde-

Index

Les nombres en italique ou en bleu renvoient à la page où l'entrée est décrite; les nombres soulignés renvoient à la ligne de code de la définition; les nombres en caractères romains renvoient à la ligne de code où l'entrée est utilisée.

K

M

N

O

P

Q

R

S

T

V

Z