

De LaTeX vers Typst pour l'enseignement

Louis-Clément LEFÈVRE

Lycée Hoche, Versailles

Louis-Clement.Lefevre@ac-versailles.fr

<https://math1b.bcpsthoche.fr>

Version du 28 octobre 2026

Le but de ce document est de raconter mon expérience avec le tout récent langage **Typst**, et de convaincre la communauté qu'il est comme il le prétend un remplaçant crédible à LaTeX, dans le contexte de production de documents d'enseignement au niveau lycée ou classes préparatoires.

A. Mon expérience avec LaTeX	1
B. Le langage Typst	2
C. Un bref tour d'horizon du langage Typst	3
C.1. Démarrage	4
C.2. Des Mathématiques	6
C.3. Les règles set et show	8
C.4. La programmation	12
D. Des fonctions avancées avec Typst	15
D.1. Templates	15
D.2. Utiliser un fichier de données CSV	17
D.3. Passer des arguments en ligne de commande	19
D.4. Export multiple	21
D.5. Quelques paquets utiles	23
E. Quelques limitations	24
E.1. Gestion très fine des espacements	24
E.2. Éléments personnalisés	25
E.3. Quelques mots sur l'intelligence artificielle	25
F. Conclusion	26

A. Mon expérience avec LaTeX

J'ai appris à utiliser LaTeX tout au long de mes études en mathématiques et m'y suis perfectionné peu à peu, notamment lors de la rédaction de rapports de stages d'une vingtaine de pages, et jusqu'à l'écriture de ma thèse de doctorat et d'articles publiés dans des revues de l'AMS (*American Mathematical Society*). J'ai passé énormément de temps à lire des « guides de bonnes pratiques » sur le langage lui-même ainsi que sur la typographie, à lire du code d'autres personnes, et à lire les documentations de paquets que j'utilise.

Quand on s'y plonge, on ne peut que constater la complexité de tout ce système : le langage TeX de base créé par Donald KNUTH, par-dessus lequel sont écrites les macros LaTeX créées par Leslie LAMPORT, puis les nombreux paquets existants qui améliorent et corrigent les macros de base tout en ajoutant de nouvelles fonctionnalités... Plus je progressais, plus je trouvais cela compliqué !

Après mon expérience de recherche, je suis devenu enseignant, pendant un an en lycée puis en classes préparatoires. Évidemment que j'allais continuer à utiliser LaTeX puisque le rendu est tellement impeccable, une fois qu'on est habitué ! Et j'aime programmer, je ne supporte de toute façon plus les logiciels comme LibreOffice, trop de boutons à cliquer partout... Je rédige tout avec l'éditeur **GNU Emacs** sur un clavier en configuration **BÉPO**.

Pour écrire des documents de cours, formulaires, feuilles d'exercices, pour les élèves, on peut se contenter du style **article** fourni de base avec LaTeX : cela peut être satisfaisant pour des élèves très matheux, à partir de L2/L3, mais pas tout à fait pour des lycéens. Sinon on en vient très vite, au minimum, à vouloir élargir la page, créer des effets sympathiques de titres et de

cadres, positionner finement des images ou des tableaux, et enfin resserrer le contenu pour tout faire tenir sur une seule page à imprimer. Si ajuster les marges et configurer des en-têtes et pieds-de-pages est effectivement « facile » (un paquet, quelques options), le reste peut vite devenir *extrêmement compliqué* avec LaTeX ! J'affirme même : il est plus facile d'écrire un article de mathématiques pures et dures pour un journal AMS, avec une classe de document déjà fournie (par exemple `amsart`) et sans trop se préoccuper de l'apparence et de la longueur finale, que d'écrire un document adapté à des élèves lycéens avec un style un petit peu personnalisé.

B. Le langage Typst

Le langage **Typst** a été créé par Laurenz MÄDJE et Martin HAUG, alors étudiants à l'Université technique de Berlin, comme projet de Master. Ils ont depuis créé leur propre société pour pouvoir se salarier et continuer à développer le projet. La version 0.1 est sortie en avril 2023 et c'est à ce moment que j'en ai entendu parler, alors que j'étais en plein dans mes réflexions sur LaTeX décrites précédemment. Et donc j'ai tout de suite voulu m'y plonger.

Comme LaTeX, le but de Typst est d'abord d'être un système de composition de documents orienté vers la publication scientifique, permettant de rédiger du contenu, de programmer, d'inclure des mathématiques, des images, de la bibliographie. Mais si un certain nombre d'idées sont évidemment inspirées de LaTeX, le but n'est pas d'améliorer celui-ci ni de chercher à être compatible : on repart du tout début, avec toutes les technologies les plus modernes (le compilateur est écrit en langage **Rust** et tout l'écosystème en est inspiré) et des idées nouvelles solidement réfléchies. Les nouvelles versions contiennent encore fréquemment des *breaking changes* si une amélioration apparaît. Le langage est open-source, chacun peut installer et tester le compilateur chez lui. En plus existe une **application en ligne** (similaire à Overleaf pour LaTeX : je connais mal car j'aime compiler sur mon propre ordinateur), dont les fonctionnalités les plus avancées sont payantes et financent la société Typst et ses salariés.

Voici résumées quelques spécificités du langage Typst :

- **Tout ce qui est fréquent doit être réalisable facilement.** Les mises en formes usuelles (gras, italique, titres, listes) se font à l'aide d'un balisage léger inspiré par Markdown ; l'encodage par défaut des documents est UTF-8 et la page est au format A4. Pas besoin de charger d'étranges paquets pour démarrer ! De nombreuses configurations de styles de texte, numérotation des titres, mise en forme des tableaux, se font très directement sans sorcellerie façon TeX.
- Le compilateur peut fonctionner en mode **compilation incrémentale** en ne recompilant que partiellement le document depuis la dernière modification. Avec l'outil en ligne de commande, cela signifie qu'on le lance en mode *watch* et qu'à chaque nouvelle sauvegarde du document, il recompile bien plus rapidement ; quant à l'application Web, elle affiche le résultat en direct au fur et à mesure ! Le compilateur peut aussi fonctionner comme une librairie Rust. Il est infiniment plus rapide et léger que les moteurs pdfTeX et LuaTeX, et fournit des messages d'erreurs précis et explicites. Cela intéresse énormément ceux qui ont besoin de produire des documents PDF en grande quantité de façon automatisée. Enfin, le compilateur peut exporter dans différents formats, de nombreuses versions du standard PDF, et aussi en HTML, une fonctionnalité annoncée dès le départ et toujours considérée comme expérimentale (même si déjà largement fonctionnelle).
- Le langage de programmation intégré est de type **fonctionnel pur** : les notions de *fonctions* et d'*expressions* priment sur la notion d'instruction, et une fonction ne peut pas modifier des variables globales hors de sa portée. C'est d'ailleurs ce principe qui permet la recompilation aussi rapide que nous venons de décrire (en un sens, cela garantit que le reste du document ne peut pas trop changer en dehors de la zone où sont faites les modifications). Si à première vue un document Typst ressemble à du langage Markdown avec des morceaux de code insérés, l'interaction entre code et contenu est bien plus profonde que cela : d'une part le contenu peut contenir du code à tout moment, mais d'autre part le contenu est un type de donnée (parmi beaucoup d'autres) créé et manipulé par du code. C'est ce qui permet de configurer globalement l'aspect de certains éléments (titres, listes, etc).

- Tout doit être prêt pour la **publication scientifique**. Un mode mathématiques (inspiré de LaTeX, mais là aussi largement ré-imaginé), insertion d'images, numérotation et légende des figures, références internes, notes de bas de page, coloration syntaxique du code. Un gros travail a été apporté à la gestion des citations et des bibliographies, un sujet dont je ne parlerai pas car je ne l'utilise pas pour l'enseignement. Les auteurs de Typst ont comme objectif explicite de percer dans le milieu de la publication scientifique en offrant une alternative à LaTeX, et le projet fait son chemin, voir par exemple [cet exposé sur YouTube](#) d'un administrateur arXiv lors du Typst Meetup qui manifeste son intérêt et explique les obstacles actuels. Les auteurs refusent de travailler eux-mêmes sur un export LaTeX depuis le compilateur : le but est de définitivement remplacer LaTeX.
- Un système de chargement de **paquets et d'extensions**, disponibles sur le [Typst Universe](#). De très nombreux paquets ont rapidement été créés par la communauté, certains encore assez expérimentaux. Nous verrons que la création de templates est en fait *tellement facile* qu'on n'est pas obligé d'aller télécharger des dizaines de paquets : on peut se contenter d'en lire le code et de s'en inspirer ou de les adapter.
- Un support de toutes les **dernières technologies modernes** (unicode, polices OpenType, images SVG, environnement Rust et Cargo, fichiers de configuration TOML, récupération automatique des paquets via un dépôt en ligne, ...) qui donnent l'impression d'un saut technologique incroyable depuis les pdfLaTeX, langage C, compilation avec Makefile, au point de regarder ces outils comme nettement dépassés.

Dès sa parution en avril 2023, le langage Typst a attiré de nombreux bidouilleurs en tout genre lassés de TeX, qui n'ont cessé d'en explorer les possibilités. Et qui ont très vite été convaincus qu'ils pouvaient recréer et améliorer leur *worflokw* LaTeX complexe en quelques heures de travail. Les versions jusqu'à la 0.10 sont sorties espacées de quelques semaines, de nombreuses fonctionnalités manquantes ont rapidement été intégrées. Chaque fois que quelqu'un disait « d'accord, Typst c'est bien, mais il manque quand même [...] » la fonctionnalité en question apparaissait dans la version suivante ! Et dès la version 0.11 (mars 2024) le langage était franchement utilisable pour produire du contenu en PDF de haute qualité. Au jour où j'écris cet article, la version la plus récente de Typst est 0.15 (juin 2026), apportant des nouveautés fort attendues que je décrirai plus loin. Il n'est pas prévu immédiatement de sortir une version 1.0 stable : les nouvelles versions n'hésitent pas à casser la compatibilité si c'est pour introduire des idées meilleures, c'est le jeu pour l'instant.

Pour démarrer, j'ai utilisé Typst pour des documents d'enseignement « personnels » (brouillons d'exercices, mise en forme de tableaux de notes d'élèves) et « de peu d'enjeu » (documents non imprimés pour les élèves : programmes de colle, TP informatique). Après ce rodage, j'ai franchi le pas et ai décidé de convertir l'intégralité de mes documents y compris feuilles d'exercices et cours distribués imprimés aux élèves, et mon objectif pour l'année scolaire 2026–2027 est de bannir définitivement LaTeX.

C. Un bref tour d'horizon du langage Typst

Je préviens : cette section n'est pas conçue comme un cours ni même un tutoriel de Typst, mais plutôt comme un guide de démarrage ! De part sa toute jeunesse, le langage Typst est encore réservé à ceux qui ont des connaissances en informatique et en programmation. Ces personnes-là n'auront pas de mal à installer le compilateur eux-mêmes en suivant les instructions sur [GitHub](#) (via une installation de Rust, ou bien un fichier binaire autonome) ; je suis de mon côté bien incapable de dépanner quelqu'un sous Windows. L'application en ligne est disponible librement pour faire des tests, il suffit de créer un compte (gratuitement). Et enfin, ceux qui apprennent Typst actuellement sont tous déjà passés par une expérience de LaTeX. Je me contenterai donc d'illustrer et surtout de motiver, comme l'indique le titre de cet article, la transition de LaTeX vers Typst.

Pour apprendre, il est de toute façon nécessaire de parcourir la [documentation](#), contenant à la fois des guides, et la liste de tous les éléments disponibles et leur options ; et d'expérimenter soi-même.

C.1. Démarrage

C.1.i. Markup

Tout d'abord, un simple texte est déjà un document Typst valide !

```
Mon tout premier document créé avec Typst !
```

```
Mon tout premier document créé avec Typst !
```

On peut l'enregistrer dans un fichier `fichier.typ` et le compiler avec

```
$ typst compile fichier.typ
```

La page est par défaut au format A4, l'encodage par défaut est UTF-8 : tout marche bien comme on voudrait. La police utilisée par défaut s'appelle Libertinus Serif, en taille 11pt.

Les mises en forme les plus usuelles s'obtiennent par un balisage léger (« *type Markdown* »). Écrire du texte entre des astérisques `*` pour le mettre en gras, entre des underscore `_` pour le souligner.

```
Mon deuxième document_ créé avec le magnifique  
*langage Typst*.
```

```
Mon deuxième document créé avec le magnifique langage  
Typst.
```

Comme avec LaTeX, les caractères espaces superflus n'ont pas de signification particulière, et les changements de paragraphe sont indiqués par deux sauts de ligne consécutifs. Le caractère `\` produit un saut de ligne, et sert aussi à échapper les autres caractères : `\*` pour produire un astérisque seul, `\_` pour un underscore.

Poursuivons :

- Les titres s'obtiennent en précédant leur ligne par un caractère égal `=`, ou plusieurs `==`, `===`, le nombre de caractères correspondant au niveau du titre.
- Les listes à puces (non numérotées) s'obtiennent en démarrant la ligne par un tiret `-`. Tout ce qui est ensuite indenté au même niveau fait partie de l'item ; éventuellement les listes peuvent être emboîtées.
- Les énumérations s'obtiennent de même en démarrant la ligne par `+` et sont automatiquement numérotées.

```
= Un document Typst
+ Premier élément de la liste.
+ Sous-item.
+ Encore.
+ Deuxième élément.
+ Troisième élément.

= Suite de mon document

== Première sous-partie
- Liste
- Sous-liste
- Sous-liste
- Fin de la liste

== Deuxième sous-partie
Fin !
```

Un document Typst

1. Premier élément de la liste.
 1. Sous-item.
 2. Encore.
2. Deuxième élément.
3. Troisième élément.

Suite de mon document

Première sous-partie

- Liste
 - Sous-liste
 - Sous-liste
- Fin de la liste

Deuxième sous-partie

Fin !

Le texte brut (*raw*) écrit dans une police à largeur fixe (*monospace*), peut être écrit en français entre ce que nous appelons des accents graves (en anglais ce sont plutôt des *backticks*) : ``Typst`` produit Typst. Il est aussi possible d'utiliser comme délimiteur trois accents graves `````, suivis immédiatement du nom du langage : la coloration syntaxique est une fonction intégrée nativement ! En sautant une ligne après ces trois accents graves, on peut aussi formater le code en mode bloc.

Créons un fichier `listes.py` et définissons une fonction Python `python tous_positifs(L)` qui prend en argument une liste `L` et teste si tous ses éléments sont positifs.

```
python
def tous_positifs(L):
    n = len(L)
    for i in range(n):
        if L[i] < 0:
            return False
    return True
```

Créons un fichier `listes.py` et définissons une fonction Python `tous_positifs(L)` qui prend en argument une liste `L` et teste si tous ses éléments sont positifs.

```
def tous_positifs(L):
    n = len(L)
    for i in range(n):
        if L[i] < 0:
            return False
    return True
```

C.1.ii. Des fonctions éléments

De nombreux autres éléments, qui n'ont pas de *syntaxe dédiée* (un markup comme ci-dessus), peuvent être créés en appelant des fonctions. Le caractère `#` permet de passer en mode code. Voici quelques autres fonctions utiles :

- `link(adresse, texte)` : produit un lien cliquable,
- `underline(texte)` : texte souligné,
- `highlight(texte)` : texte surligné,
- `lorem(n)` : produit *n* mots du faux-texte *lorem ipsum*.

À tout moment, le caractère `//` marque un commentaire jusqu'au bout de la ligne.

Essayons ces éléments :

```
Lisez la
#link("https://typst.app/docs/")[documentation] !
C'est #underline[très important]. Vous comprendrez
alors ce que fait #highlight[cette commande] :
`lorem(20)`

// combler par du texte
#lorem(20)
```

Lisez la documentation ! C'est très important. Vous comprendrez alors ce que fait **cette commande** : `lorem(20)`

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

(le lien n'apparaît plus cliquable lorsque le PDF est inséré ci-contre ; et par défaut, il n'a pas de mise en forme particulière)

Quelques mots sur les syntaxes : les arguments d'une fonction sont normalement passés entre parenthèses, et les crochets [...] servent à délimiter des *blocs de contenu* ; les chaînes de caractères sont automatiquement converties en contenu lorsqu'on demande à les insérer dans le document. De nombreuses fonctions prennent comme argument des paramètres optionnels puis un ou plusieurs blocs de contenus à formater. Pour éviter un certain emboîtement des parenthèses, ceux-ci peuvent être écrits *juste après* la fonction, collés. Ainsi toutes les syntaxes ci-dessous donnent exactement le même résultat pour un titre (**heading**) de niveau 1 :

```
// mode markup
= Titre
// appel de fonction
#heading([Titre])
// idem, mais argument de type chaîne de caractères
#heading("Titre")
// en fait le plus pratique
#heading[titre]
```

De même, une liste numérotée peut être créée dans du code par deux syntaxes équivalentes :

```
// fonction à plusieurs arguments
#enum([premier item], [deuxième item], [troisième item])
// blocs de contenu hors de l'appel de fonction
#enum[premier item][deuxième item][troisième item]
```

Les éléments rencontrés précédemment correspondent tous à des fonctions : **strong** pour le texte mis en gras, **emph** pour le texte en italique (pour des raisons de séparation entre la sémantique et la mise en forme, il convient effectivement d'appeler ceci des mises en valeurs *fortes* et *légères* :

le fait que le texte apparaisse en gras ou italique n'est qu'un choix et peut être modifié, alors qu'il existe d'autres moyens de forcer le texte en gras ou italique), **heading** pour les titres, **raw** pour le texte brut, **enum** et **list** pour les listes numérotées ou non.

Enfin, étudions un instant l'élément **text**. Celui-ci formate le texte passé en argument et possède aussi de nombreux paramètres optionnels :

- **size** : la taille de la police de caractères, absolue (en point **pt**) ou relative (**em**, ou pourcentage).
- **fill** : la couleur du texte. Les couleurs disponibles en Typst ne sont pas des chaînes de caractères (écrites entre guillemets), mais des objets à part entière, avec des méthodes et des opérations.
- **weight** : épaisseur ("**bold**" pour gras).

Nous y reviendrons avec les règles **set** et **show**.

```
Jouons avec le
#text(weight: "bold", fill: red)[texte]
et ses propriétés.

#text(size: 1.5em)[Pour comprendre, il faut
expérimenter par *soi-même* !]

#text(fill: blue, size: 8pt)[#lorem(20)]
```

Jouons avec le **texte** et ses propriétés.

Pour comprendre, il faut expérimenter
par **soi-même** !

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

C.2. Des Mathématiques

C.2.i. Principes

Le mode mathématiques est introduit à l'aide de symboles dollar **\$...\$**. Sans laisser d'espace entre le caractère et le texte qui suit, la formule est en mode en-ligne. Si on laisse un espace **\$... \$**, elle est alors en mode bloc.

Brièvement, les règles du mode mathématiques sont les suivantes :

- Les lettres seules **x**, **y** sont des noms de variables.
- Les mots composés sont des symboles ou des fonctions : **forall**, **alpha**, **exp**, **sum**, etc.
- Les parenthèses peuvent à la fois être affichées comme tel, ou servent à former des groupes pour les fonctions, indices, exposants et fractions. Par exemple, **sum_(k=1)^n k^2** est interprété comme $\sum_{k=1}^n k^2$ et **(x + y) / 2** est interprété comme $\frac{x+y}{2}$ où la barre / forme automatiquement les fractions (sinon, il faut l'échapper avec ****). Les syntaxes **x^n** et **x^(n)** donnent donc exactement la même chose x^n , mais pour écrire x^{n+1} il faut des parenthèses **x^(n+1)** ; et pour écrire $x^{(n+1)}$ il faut donc écrire **x^((n+1))**. En plus, les parenthèses et autres délimiteurs sont automatiquement agrandis et viennent parfois avec leur propre fonction : **abs(x / y)** forme une valeur absolue $\left|\frac{x}{y}\right|$ avec les bons espacements et les bonnes tailles des délimiteurs.

Inutile de commenter plus, autant regarder directement un exemple.

```
Soit une fonction $f$ supposée $cal(C)^oo$ sur un
intervalle $I subset RR$.
Soit $a in I$, soit $n in NN$.
La *formule de Taylor avec reste intégral* à l'ordre
$n$ nous dit :
$ forall x in I, quad
f(x) = sum_(k=0)^n (f^(k))(a) / k! (x-a)^k
+ integral_a^x (f^((n+1))(t)) / n! (x-t) dif t $
```

Soit une fonction f supposée C^∞ sur un intervalle $I \subset \mathbb{R}$. Soit $a \in I$, soit $n \in \mathbb{N}$. La **formule de Taylor avec reste intégral** à l'ordre n nous dit :

$$\forall x \in I, \quad f(x) = \sum_{k=0}^n \frac{f^{(k)}(a)}{k!} (x-a)^k + \int_a^x \frac{f^{(n+1)}(t)}{n!} (x-t) dt$$

Une bonne partie de la « beauté » de cette mise en forme ne provient en fait pas de Typst lui-même, mais de la police de caractères utilisée, New Computer Modern, qui est une version retravaillé de la police classique Computer Modern créée par Donald KNUTH pour le TeX d'origine.

Dans cet exemple, le symbole infini ∞ s'appelle tout simplement **oo** et le symbole **dif** produit le d droit avec les bons espacements. La fonction **cal()** appelle les lettres calligraphiques ; éventuellement la fonction **bb()** appelle les lettres *blackboard*, mais les lettres usuelles $\mathbb{N}$, $\mathbb{R}$ ont déjà un nom **NN**, **RR**.

C.2.ii. Nommage des symboles

Tous les symboles ont été renommés par rapport à LaTeX. Si le $\forall$ s'appelle bien `forall`, le $\sum$ s'appelle bien `sum`, les lettres grecques ont gardé leur nom, mais l'intégrale $\int$ s'appelle maintenant `integral`. En fait, beaucoup de symboles correspondent maintenant à une « catégorie » contenant des sous-symboles et des modificateurs. Par exemple, le symbole d'appartenance $\in$ s'appelle `in` et son contraire $\notin$ s'appelle `in.not`. Le symbole de flèche simple $\rightarrow$ s'appelle `arrow.r` (*right*), la flèche dans l'autre sens $\leftarrow$ s'appelle `arrow.l` (*left*), et les variantes longues $\longrightarrow$, $\longleftarrow$ s'appellent `arrow.r.long` et `arrow.l.long`. Les variantes doubles $\Rightarrow$, $\Leftarrow$ (implications) sont obtenues en rajoutant le modificateur `double` : `arrow.r.double`, `arrow.l.double`, peu importe l'ordre (`arrow.double.r` donne le même symbole), et existent aussi en version longue $\Longrightarrow$, $\Longleftarrow$, et les variantes à barre $\mapsto$ sont obtenues avec le modificateur `bar`, en version longue $\mapsto$. Nous venons donc d'apprendre d'un seul coup à écrire plusieurs dizaines de flèches, et tout cela est en fait beaucoup plus logique et facile à retenir !

Il ne s'agit pas seulement d'un renommage, mais d'une nouvelle stratégie pour classer et accéder aux symboles. Beaucoup d'autres flèches et délimiteurs ont des variantes `l`, `r` ; beaucoup de relations ont des variantes `not` (par exemple `eq.not` pour $\neq$).

En plus de ce renommage, un certain nombre de symboles admettent un « markup mathématiques » permettant de les entrer plus facilement. La flèche simple $\rightarrow$ est obtenue par la séquence `->`, la flèche d'implication $\Rightarrow$ par `=>`, et la variante à barre $\mapsto$ par `|->`. La liste précise est disponible dans la [documentation](#). Cela rend un code Typst déjà particulièrement lisible !

En fait, j'ai appris à entrer directement les symboles mathématiques unicode, soit au clavier BÉPO soit avec Emacs, et j'utilise de moins en moins les noms en Typst. Un code source peut ressembler à ceci :

```
La fonction f est continue en a si et seulement si :  
\forall \epsilon > 0, \text{space} \exists \alpha > 0, \text{space} \forall x \in \mathcal{D}_f \text{quad } (|f(x) - f(a)| < \epsilon)
```

C.2.iii. Alignements et affinements

En mode mathématiques, le symbole `\` permet de passer à la ligne, et les symboles `&` fonctionnent à peu près comme avec LaTeX pour insérer des points d'alignement des équations. Le texte en français peut être inséré simplement entre guillemets. Signalons en plus que le mode mathématiques peut à tout moment passer en mode contenu avec un caractère `#` et que le contenu lui-même peut contenir des mathématiques ! Cela permet de créer des alignements complexes, voir cet exemple.

```
Soit n in  $\mathbb{N}^*$ , supposons la propriété cal(P)(n) vraie. Alors :  
\sum_{k=1}^{n+1} k^2 &= \sum_{k=1}^n k^2 + (n+1)^2 \quad \&= \frac{n(n+1)(2n+1)}{6} + (n+1)^2  
&\quad \text{"par"} \text{cal(P)(n)} \quad \&= \boxed{\frac{(n+1)(n+2)(2n+3)}{6}}  
\quad \&\quad \text{"en factorisant"}
```

Soit $n \in \mathbb{N}^*$, supposons la propriété $\mathcal{P}(n)$ vraie. Alors :

$$\begin{aligned} \sum_{k=1}^{n+1} k^2 &= \sum_{k=1}^n k^2 + (n+1)^2 \\ &= \frac{n(n+1)(2n+1)}{6} + (n+1)^2 \quad (\text{par } \mathcal{P}(n)) \\ &= \boxed{\frac{(n+1)(n+2)(2n+3)}{6}} \quad (\text{en factorisant}) \end{aligned}$$

Le mode mathématiques permet des ajustements typographiques fins. La fonction `op()` permet de formater son argument comme un opérateur, avec les bons espacements : `$op("Argth")(x)$` produit bien $\text{Argth}(x)$.

Les caractères ont tous, comme avec TeX, une classe qui définit finement leurs espacements (symboles ordinaires, opérations binaires, relations binaires, délimiteur ouvrant ou fermant, etc), et la modification de cette classe est accessible avec la fonction `class()`. Observer par exemple les mauvais espacements dans la mise en forme d'intervalles semi-ouverts en français : $[0, 1[\cup]2, 3]$ (le caractère `[` est toujours interprété comme un délimiteur ouvrant, alors qu'ici on l'utilise aussi en fermant), entré naïvement par `$[0, 1[ \union ]2, 3]$`. La fonction `lr()` (*left right*) permet de faire correspondre un délimiteur ouvrant avec son délimiteur fermant, avec les bons espaces, ainsi `$lr([0, 1[ \union ]2, 3])$` produit bien $[0, 1[\cup]2, 3]$. Pour

que l'éditeur de texte détecte correctement la paire de symboles, et la coloration syntaxique, je les entre en fait par leur nom complet `bracket.l`, `bracket.r`.

De nombreuses autres options de présentation peuvent être obtenues en appliquant des règles `set` et `show` sur les objets `math.equation`.

C.2.iv. Conclusion : les trois modes

Terminons par la remarque : en tant que langage, Typst a réellement trois modes différents, dans lesquels les règles d'analyse de la syntaxe peuvent être différentes :

- Le mode markup, celui dans lequel démarre l'analyse d'un fichier.
- Le mode mathématiques, introduit par un dollar `$`.
- Le mode code (voir plus loin), introduit par une dièse `#`.

Dans le mode code, les crochets `[, ]` servent à délimiter des blocs de contenu (qui sont alors en mode markup), et les accolades `{, }` servent à délimiter des blocs de code. Mais dans le mode mathématiques, ces symboles n'ont pas de signification particulière et sont seulement des délimiteurs (il n'y a donc pas besoin de les échapper avec `\[, \{`). Le symbole `*` est une multiplication dans le mode code, un marqueur de texte *strong* dans le mode markup (qu'il faut échapper si on veut un simple astérisque), et en mathématiques il n'a pas d'autre signification que le symbole astérisque.

Tous ces modes peuvent être emboîtés les uns dans les autres à volonté, et il n'y a jamais d'ambiguïté, car la syntaxe du langage est bien définie (l'interpréteur construit un *arbre de syntaxe abstrait* à partir de tout le document). Ainsi les mathématiques peuvent contenir du contenu et de la mise en forme, et le code peut renvoyer du contenu ou des mathématiques ou les deux.

C.3. Les règles `set` et `show`

C.3.i. Règles `set`

Comme nous l'avons vu, tous les éléments de contenu (titres, listes, liens, etc) accessibles en Typst sont en fait créés par des fonctions, y compris le texte.

En première approche, une **règle `set`** permet de configurer globalement les propriétés de ces éléments.

Prenons par exemple les propriétés du texte, qui est créé par la fonction `text`. La syntaxe `#set text(fill: red)` va configurer la couleur du texte à rouge, à partir du moment où elle est appelée. On souhaite aussi fréquemment configurer la taille du texte (`size`), et la police de caractère (`font`). La police New Computer Modern, livrée avec Typst, est une version retravaillée de la police utilisée par défaut par TeX.

```
#set text(size: 10pt, font: "New Computer Modern")
Nous retrouvons maintenant un aspect similaire à
LaTeX !

#set text(fill: red)
À partir d'ici, tout le texte est en rouge.
```

Nous retrouvons maintenant un aspect similaire à LaTeX !
À partir d'ici, tout le texte est en rouge.

Il est important de comprendre que la règle `set` ne fait effet qu'à l'intérieur de son bloc d'instructions et ne peut pas porter à l'extérieur. Cela fait partie de l'aspect « fonctionnel pur » du langage. On peut l'observer en définissant un bloc de contenu `#[...]` :

```
#set text(font: "New Computer Modern")
La police utilisée est la même qu'avec LaTeX.

#[
  #set text(fill: red)
  Ici le texte est en rouge.
]

Ce texte n'est plus affectée par cette dernière règle
`set`.
```

La police utilisée est la même qu'avec LaTeX.
Ici le texte est en rouge.
Ce texte n'est plus affectée par cette dernière règle `set`.

Un autre paramètre important à configurer concerne les paragraphes, qui correspondent à la fonction `par`. Elle possède les options suivantes :

- `justify` : justifier le texte, indiquer `true` ou `false`,
- `leading` : espacement entre les lignes,
- `spacing` : espacement entre les paragraphes,
- `first-line-indent` : indentation de la première ligne, voir la documentation.

La documentation précise (*Guide for LaTeX users*) que la valeur `0.55em` pour `leading` et `spacing` permet de retrouver l'aspect d'un paragraphe avec LaTeX.

```
#set text(
  size: 10pt,
  font: "New Computer Modern",
)
#set par(
  justify: true,
  leading: 0.55em,
  spacing: 0.55em,
  first-line-indent: 1.8em,
)
Dans ce document nous retrouvons un aspect similaire à
ce dont nous avons l'habitude avec LaTeX. Le premier
paragraphe n'est pas indenté, mais les suivants le
sont.

En effet : #lorem(40)

#lorem(50)
```

Dans ce document nous retrouvons un aspect similaire à ce dont nous avons l'habitude avec LaTeX. Le premier paragraphe n'est pas indenté, mais les suivants le sont.

En effet : Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aequi doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aequi doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut.

Pour le présent article, les valeurs sont, avec en plus les adaptations de typographie française :

```
#set text(
  size: 11pt,
  font: "New Computer Modern",
  lang: "fr",
  region: "fr",
)
#set par(
  justify: true,
  leading: 0.55em,
  spacing: 1em,
)
```

Testez diverses valeurs... Dans tous les cas il faut au moins ou bien un espace marqué entre les paragraphes, ou bien une indentation des premières lignes.

Intéressons-nous maintenant aux titres de sections. Un paramètre intéressant et qu'on désire rapidement configurer est l'aspect de la numérotation des titres, dans l'option `numbering`. On peut le désactiver globalement avec la règle `#set heading(numbering: none)`, ou créer un seul titre non numéroté avec la fonction `#heading(numbering: none)[titre]`. Mais Typst introduit aussi une idée nouvelle et très simple (*ce qui est fréquent doit être réalisable facilement*), le *numbering pattern*, formé d'une chaîne de caractère contenant des symboles `1`, `I`, `i`, `a`, `A`, et des caractères de séparation entre, indiquant un motif d'affichage du compteur des titres (ainsi le symbole `1` correspond aux chiffres arabes, `I` aux chiffres romains majuscule, `a` à la numérotation par des lettres minuscules, etc). On peut donc le configurer pour les titres :

```
#set heading(numbering: none)
= Première partie
= Deuxième partie

#set heading(numbering: "A-i-1.")
= Titre
== Sous-titre
=== Sous-sous-titre
== Autre sous-titre
== Dernier sous-titre
```

Première partie

Deuxième partie

A. Titre

A-i. Sous-titre

A-i-1. Sous-sous-titre

A-ii. Autre sous-titre

A-iii. Dernier sous-titre

En toute généralité, **numbering** est une fonction prenant en argument une chaîne de caractères (le motif) et une liste de nombres, et qui la formate selon le motif ; et à la place d'une chaîne de caractères, on peut aussi lui donner une fonction quelconque qui va recevoir les nombres et produire un contenu, ce qui permet des variations très créatives !

Cela fonctionne aussi pour les listes. Éventuellement, l'option **full: true** permet de donner à chaque item la numérotation complète. Une autre option configurable par une règle **set** est l'espace entre le numéro et le texte (**body-indent**).

<pre>#set enum(full: true, numbering: "1.a.i)", body-indent: 1em,) + Une liste avec un premier élément + Un sous-élément. + Un sous-sous élément. + Retour en arrière. + Un dernier élément.</pre>	<pre>1) Une liste avec un premier élément 1.a) Un sous-élément. 1.a.i) Un sous-sous élément. 1.b) Retour en arrière. 2) Un dernier élément.</pre>
---	---

D'autres paramètres usuels à utiliser dans des règles **set** :

- L'élément **page()** permet de contrôler divers paramètres de la page :
 - **paper** : la taille du papier, par défaut **"a4paper"**,
 - **width** : la largeur de la page,
 - **height** : la hauteur, **auto** pour une page qui grandit à l'infini,
 - **margin** : la taille de la marge, donnée éventuellement par un dictionnaire, (**x: 2.5cm**, **y: 1cm**) pour le présent document,
 - **header** : en-tête de la page, voir la documentation pour des constructions élaborées, ou **none** pour le supprimer totalement,
 - **footer** : pied-de-page, idem.
- L'élément **document()** permet de donner des métadonnées au fichier PDF produit :
 - **title** : le titre,
 - **author** : l'auteur,
 - **keywords** : mots-clés, donnés comme une liste.

C.3.ii. Règles **show-set** (chaussettes)

Tout cela est bien intéressant, mais il y a un problème. Par exemple, la couleur et la taille du texte dans un titre ne sont *pas des propriétés de la fonction heading*, et donc une règle **set** ne peut pas les modifier directement. Mais ce sont des propriétés *de la fonction text utilisée dans les éléments heading*. Pour configurer ces propriétés, il faut utiliser une **règle show** qui configure plus en profondeur comment un élément doit être construit ; et pour l'instant nous allons nous limiter aux règles **show-set** (je ne peux m'empêcher de penser à *règles chaussettes*...) servant à configurer les paramètres d'un élément à l'intérieur d'un autre. La règle prend la forme **#show heading:** et est immédiatement suivie par une règle **set**.

Changeons un petit peu la mise en forme des titres, ainsi que de quelques autres éléments, par exemple mettons en valeur les liens en rouge (par défaut, ils n'ont aucune mise en forme). La création des titres utilise à la fois des éléments **block** (un bloc, avec des options d'espacement, de remplissage et de bordure) et **align** (alignement).

<pre>#show heading: set text(fill: blue, size: 18pt) #show heading: set align(center) #show heading: set block(above: 2em, below: 0.5em) #show link: set text(fill: red) = Mon article Lisez la #link("https://typst.app/docs/") [documentation] si vous voulez progresser ! = Conclusion Typst c'est bien !</pre>	Mon article Lisez la documentation si vous voulez progresser ! Conclusion Typst c'est bien !
--	---

La règle **show** peut aussi s'appliquer uniquement sur certains éléments selon les valeurs de leurs paramètres. Par exemple on peut cibler uniquement les titres de niveau 1 avec la règle **#show heading.where(level: 1)**.

```
#set text(size: 10pt)
#set heading(numbering: "A.1.")
#show heading.where(level: 1): set text(fill: red,
size: 12pt)
#show heading.where(level: 2): set text(size: 10pt)

= Titre de niveau 1
#lorem(10)

== Titre de niveau 2
#lorem(20)

== Encore
#lorem(20)

= Dernier titre
#lorem(30)
```

A. Titre de niveau 1

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.

A.1. Titre de niveau 2

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

A.2. Encore

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

B. Dernier titre

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aequi doleamus animo, cum corpore dolemus, fieri.

De nombreux éléments sont emboîtés dans des `block` et des `align`. Leur appliquer des règles `show` est très facile, il suffit d'être créatif ! Les titres de niveau 1 et 2 de ce document sont rendus par la règle suivante (élargir le bloc, tracer une bordure en bas, en laissant un petit peu d'espace) :

```
#show heading.where(level: 1): set block(
  width: 100%,
  stroke: (bottom: 0.5pt),
  inset: (bottom: 6pt),
)
#show heading.where(level: 2): set block(
  width: 100%,
  stroke: (bottom: (dash: "dashed", thickness: 0.5pt)),
  inset: (bottom: 6pt),
)
```

Comme spécifié dans la documentation, les valeurs `above: 1.4em` et `below: 1em` sont les plus proches de « l'aspect LaTeX ».

Pour la coloration du code, après de nombreux essais, j'utilise un thème appelé **Pastie** (très légèrement retouché) aux couleurs éclatantes, fourni dans un fichier `.tmTheme`, que je trouve bien adapté pour l'enseignement. Le thème est configuré avec une règle `set`, et les éléments `raw` en mode bloc sont stylisés avec fond, bordure, et alignement du bloc.

```
#set raw(theme: "Pastie.tmTheme")
#show raw.where(block: true): set block(
  stroke: 0.5pt,
  fill: silver.lighten(50%),
  radius: 6pt,
  inset: 6pt,
)
#show raw.where(block: true): set align(center)

Écrivons une fonction ``python fibonacci(n)`` qui
renvoie le $n$-ème terme de la suite de Fibonacci :
``python
def fibonacci(n):
    assert n >= 0, "n doit être positif"
    # u et v représentent
    # deux termes successifs
    u = 0
    v = 1
    for i in range(n):
        t = u + v
        v = t
        u = v
    return u
``
```

Écrivons une fonction `fibonacci(n)` qui renvoie le n -ème terme de la suite de Fibonacci :

```
def fibonacci(n):
    assert n >= 0, "n doit être positif"
    # u et v représentent
    # deux termes successifs
    u = 0
    v = 1
    for i in range(n):
        t = u + v
        v = t
        u = v
    return u
```

Pendant que nous y sommes, il est extrêmement facile de charger le code depuis un fichier : c'est la combinaison de l'élément `raw` (qu'il faut alors appeler par la fonction, et non par la syntaxe dédiée) et du chargement de fichier texte `read()` :

```
#raw(block: true, lang: "python", read("fibonacci.py"))
```

Le même genre de manipulations permet de styliser les équations mathématiques : ce sont des éléments `math.equation`, éventuellement avec l'option `block: true` ou `block: false`. On peut notamment vouloir configurer la numérotation des équations.

Les possibilités de bordure (*stroke*) et de remplissage (*fill*) sont très nombreuses et à expérimenter !

C.3.iii. Règles `show`, cas général

Nous verrons après que les règles `show` sont en fait beaucoup plus générales que cela.

D'abord elles s'appliquent à une *cible* qui peut être certains types d'éléments, éventuellement obtenus par un mécanisme de recherche et de sélection élaboré, voire obtenus par une expression régulière.

Et ensuite elles peuvent appliquer à la cible une fonction quelconque, qui indique comment transformer la cible. Le langage Typst est donc beaucoup, beaucoup plus puissant, qu'un simple langage Markdown avec quelques configurations globales des propriétés du texte, tel qu'on peut en avoir l'impression au premier abord !

C.4. La programmation

Cette section est pour moi la plus délicate à écrire ; je ne vais certainement pas donner un cours de programmation.

C.4.i. Quelques bases

D'abord, en Typst, c'est le mot-clé `let` qui permet de déclarer à la fois des variables et des fonctions. C'est presque la même chose, tout dépend desquelles variables sont libres... Les accolades `{...}` servent à délimiter un bloc de code. Les variables ont leur portée restreinte au bloc dans lequel elles sont déclarées et ne peuvent plus être modifiées : c'est un langage fonctionnel pur. Bien que l'instruction `return` existe, les fonctions renvoient simplement la dernière valeur exprimée, et `none` par défaut.

```
#let x = 3
#let moyenne(a, b) = {
  (a + b) / 2
}
La moyenne de 4 et 5 est #str(moyenne(4, 5));.
```

(le point-virgule sert à terminer le code démarré par `#`, il ne s'affiche pas)

C'est aussi de cette façon qu'on fabriquerait des macros façon TeX.

```
#let Argth = $op("Argth")$
$ y = Argth(x) $
```

Les conditions `if`, `else`, `else if` fonctionnent elles aussi en renvoyant une valeur (`none` par défaut). Il ne serait en effet pas intéressant d'écrire quelque chose comme `if condition { x = 3 }` : la valeur de `x` n'existerait que sous le bloc d'instructions ! Les conditions sont écrites sans parenthèses ni délimiteurs, mais le bloc d'instructions correspondant doit être dans des accolades.

```
#let signe(x) = if x > 0 { "positif" } else if x < 0 { "négatif" } else
{ "nul" }
```

Les boucles `for` et `while` existent, mais on peut largement s'en passer, car on manipule surtout les listes par l'approche fonctionnelle (par exemple la méthode `.map()` qui prend en argument une fonction quelconque et l'applique sur chaque élément de la liste). Voir la section sur les fichiers CSV pour un exemple bien concret.

Lorsqu'on veut créer une fonction qui renvoie du contenu, prêt à insérer dans le document, on n'a pas besoin d'instruction particulière « `print` » pour sortir le contenu ; et on n'a pas non plus besoin de « fabriquer » le contenu avant de le renvoyer (bien que l'opération `+` soit utile pour concaténer des chaînes de caractères ou du contenu). Une telle fonction, dans son corps, se

contente de poser les éléments de contenu au fur et à mesure, appliquer des règles `set` ou `show` (qui ne portent qu'à l'intérieur de la fonction ; il n'y a plus de `#` devant car nous sommes déjà en mode code), et le tout est renvoyé à la fin. C'est extrêmement pratique, voir le cas d'étude ci-après. Un exemple peut ressembler à ceci :

```
#let important(body) = {
  set text(fill: red)
  [*Message important*]
  linebreak()
  body
}

#important[Essayez Typst vous-même et vous serez convaincus !]

#lorem(20)
```

Message important

Essayez Typst vous-même et vous serez convaincus !

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

C.4.ii. Les types

Le langage Typst manipule une grande diversité de types. Bien sûr il y a des nombres entiers, des flottants, des booléens (les valeurs `true` et `false` s'écrivent sans majuscule), des chaînes de caractères ; et il faut noter que les blocs de contenu (introduits entre crochets [...]) forment aussi un type à part entière. Le simple fait d'avoir une séparation claire entre ces types, ainsi que des fonctions à arguments facultatifs, rend déjà la programmation mille fois plus agréable qu'avec TeX !

D'autres quantités plus surprenantes sont des types avec des opérations : les longueurs (on peut parfaitement utiliser la longueur `11pt * 1.5`), les couleurs (avec des opérations et des méthodes, par exemple `red.lighten(50%)` pour éclaircir le rouge), les compteurs.

Les listes, elles, sont écrites entre parenthèses, et il est possible de laisser une virgule vide à la fin : `let L = (1, 3, 5, 7, )`. On peut accéder au *i*-ème élément par la syntaxe `L.at(i)`. Mais en pratique on la manipule par sa méthode `.map()` et une fonction anonyme introduite par la syntaxe `=>`. Par exemple la syntaxe suivante transforme notre liste en une liste de contenus :

```
L.map(x => [Nombre : #str(x)])
```

On peut alors enchaîner avec une méthode `.join()` pour coller tout, autour d'un séparateur.

```
#let L = (1, 3, 5, 7, )
#L.map(x => [Nombre : #str(x)]).join(linebreak())
```

Nombre : 1
Nombre : 3
Nombre : 5
Nombre : 7

Il s'agit, éventuellement, d'une façon très différente de penser par rapport aux itérations classiques en Python `for i in range(len(L))` ; mais c'est bien pratique et on peut tout faire avec.

C.4.iii. Étude de cas

On souhaite maintenant réaliser une fonction `exercice()` qui mette en forme un exercice. Étudions ce cas et améliorons-le au fur et à mesure.

D'abord, notre exercice apparaîtra dans un bloc encadré et avec en titre le mot « Exercice ». La fonction prend donc en argument uniquement l'énoncé et le place dans le bloc.

```
#let exercice(énoncé) = {
  block(width: 100%, stroke: 0.5pt, inset: 6pt, {
    [*Exercice*]
    linebreak()
    énoncé
  })
}

#exercice[
  Démontrer :  $\forall (x, y) \in (\mathbb{R}_+)^2, \sqrt{xy} \leq \frac{x+y}{2}$ .
]
```

Exercice

Démontrer : $\forall (x, y) \in (\mathbb{R}_+)^2, \sqrt{xy} \leq \frac{x+y}{2}$.

À l'intérieur de l'élément `block`, on place un bloc de code. Dedans, on peut aussi bien écrire `[*Exercice*]` (bloc de contenu, en mode markup) ou `strong("Exercice")` (mode code uniquement).

Poursuivons : on souhaite donner à nos exercices un niveau de difficulté, comme un paramètre optionnel. Ce sera un nombre entier et on veut l'afficher comme un nombre de petites étoiles. La syntaxe `"*" * difficulté` multiplie tout simplement une chaîne de caractères par un nombre entier, donc... c'est, encore, très facile.

```
#let exercice(difficulté: none, énoncé) = {
  block(width: 100%, stroke: 0.5pt, inset: 6pt, {
    [*Exercice*]
    if difficulté != none {
      " " + strong("(" + "*" * difficulté + ")")
    }
    linebreak()
    énoncé
  })
}

#exercice[
  Démontrer : $forall (x, y) in (RR_+)^2$,
  $sqrt(x y) lt.eq.slant (x + y) / 2$.
]

#exercice(difficulté: 3)[
  Démontrer l'hypothèse de Riemann.
]
```

Exercice

Démontrer : $\forall (x, y) \in (\mathbb{R}_+)^2, \sqrt{xy} \leq \frac{x+y}{2}$.

Exercice (***)

Démontrer l'hypothèse de Riemann.

Nous souhaitons donner à nos exercices un sous-titre, lui aussi optionnel. Il est espacé du mot « Exercice » (fonction `h()` : espacement horizontal) et en italique (fonction `emph()`).

```
#let exercice(difficulté: none, soustitre: none,
énoncé) = {
  block(width: 100%, stroke: 0.5pt, inset: 6pt, {
    [*Exercice*]
    if difficulté != none {
      " " + strong("(" + "*" * difficulté + ")")
    }
    if soustitre != none {
      h(1em) + emph(soustitre)
    }
    linebreak()
    énoncé
  })
}

#exercice(soustitre: [Inégalité arithmético-
géométrique])[
  Démontrer : $forall (x, y) in (RR_+)^2$,
  $sqrt(x y) lt.eq.slant (x + y) / 2$.
]

#exercice(difficulté: 3)[
  Démontrer l'hypothèse de Riemann.
]
```

Exercice *Inégalité arithmético-géométrique*

Démontrer : $\forall (x, y) \in (\mathbb{R}_+)^2, \sqrt{xy} \leq \frac{x+y}{2}$.

Exercice (***)

Démontrer l'hypothèse de Riemann.

Enfin, il manque un compteur. Ce sujet est légèrement délicat : le compteur est identifié ici par son nom `"exercice"`, supposé initialisé à 0, et incrémenté par sa méthode `.step()`. Le bon ordre, c'est *d'abord incrémenter, puis afficher*, ce qui se fait élégamment comme pour les titres avec un *numbering pattern* (en général, un compteur Typst peut être une liste de nombres, correspondant à plusieurs niveaux). Le mot-clé `context` indique que l'expression qui suit ne peut pas être calculée tout de suite, mais dépend d'autres paramètres qui pourront être résolus plus tard. C'est important pour l'aspect fonctionnel du langage : un compteur *n'est pas* une simple variable globale, mais un « état » du document à chaque position.

```

#let exercice(difficulté: none, soustitre: none,
énoncé) = {
  block(width: 100%, stroke: 0.5pt, inset: 6pt, {
    counter("exercice").step()
    context strong(
      "Exercice " + counter("exercice").display("1")
    )
    if difficulté != none {
      " " + strong("(" + "*" * difficulté + ")")
    }
    if soustitre != none {
      h(1em) + emph(soustitre)
    }
    linebreak()
    énoncé
  })
}

#exercice(soustitre: [Inégalité arithmético-
géométrique])[
  Démontrer : $forall (x, y) in (RR_+)^2$,
  $sqrt(x y) lt.eq.slant (x + y) / 2$.
]

#exercice(difficulté: 3)[
  Démontrer l'hypothèse de Riemann.
]

```

Exercice 1 *Inégalité arithmético-géométrique*

Démontrer : $\forall (x, y) \in (\mathbb{R}_+)^2, \sqrt{xy} \leq \frac{x+y}{2}$.

Exercice 2 (**)

Démontrer l'hypothèse de Riemann.

C.4.iv. Retour sur les règles show

Dans le cas le plus général, une règle `show` s'applique à une cible et la transforme selon une fonction, parfois anonyme et dont l'argument s'appelle traditionnellement `it`. Cela permet de reconfigurer en profondeur l'aspect de certains éléments. Ici pour les titres : `it.body` est le contenu du titre, qu'on affiche séparément de son compteur. En mode markup, la séquence `---` insère un tiret long `—` (on l'obtient aussi en mode code par `sym.dash.em`).

```

#set heading(numbering: "I.1")
#show heading.where(level: 1): it => {
  set text(fill: red, size: 18pt)
  align(center, {
    context {
      [---] + counter(heading).display() + [---]
    }
    linebreak()
    it.body
  })
}

= Premier titre
#lorem(20)

= Deuxième titre
#lorem(20)

```

—I—

Premier titre

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

—II—

Deuxième titre

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

La documentation donne de nombreux exemples très créatifs de règles `show` appliquées aux listes ou aux textes bruts.

D. Des fonctions avancées avec Typst

Dans cette partie je vais lister un peu plus en vrac diverses fonctionnalités du langage Typst, qui pourraient sembler incroyablement avancées à un utilisateur LaTeX mais qui sont implémentées très simplement avec Typst. Et je vais donner à chaque fois des exemples d'application pour un enseignant.

D.1. Templates

La documentation contient spécifiquement un [tutoriel sur la création de templates](#) donc je vais être bref.

Un **template** en Typst, c'est avant tout une fonction qu'on va appeler `template` qui va recevoir l'intégralité du document, dans un argument qu'on nommera `body`, et va le mettre en forme

notamment en appliquant des règles `set` et `show` ; et une règle `show` *inconditionnelle* qui transforme tout ce qui suit selon la fonction, de la forme `#show: template`.

Un exemple minimal peut ressembler à ceci :

```
#let template(body) = {
  set text(
    size: 10pt,
    font: "New Computer Modern",
  )
  set par(justify: true)
  set heading(numbering: "I.1")
  body
}

#show: template

= Introduction
Ceci est mon premier template.

== Sous-partie
#lorem(20)

= Conclusion
#lorem(30)
```

I Introduction

Ceci est mon premier template.

I.1 Sous-partie

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

II Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aequi doleamus animo, cum corpore dolemus, fieri.

En pratique, le template sera souvent dans un fichier séparé, qu'on importe (avec des syntaxes fortement similaires à Python). Le fichier contient fréquemment d'autres fonctions utiles, par exemple pour mettre en forme des exercices ou des théorèmes. La fonction `template` contient souvent d'autres arguments, facultatifs, comme un titre pour le document qu'elle va mettre en forme. On l'appelle alors avec la syntaxe `template.with(...)` qui renvoie la fonction `template` avec des arguments pré-appliqués.

Un exemple un peu plus fourni, qui utilise l'élément `title()` pour mettre en forme le titre du document, dans un fichier `template.typ`

```
#let template(
  titre: none,
  numéro: none,
  body
) = {
  set text(
    size: 11pt,
    font: "New Computer Modern",
  )
  set par(justify: true)
  set heading(numbering: "I.1.")
  show title: set text(size: 18pt)
  show title: set align(center)
  title([Chapitre #str(numéro) : #titre])
  body
}
```

et le résultat, utilisé dans un autre fichier

```
#import "template.typ": *
#show: template.with(
  titre: "Trigonométrie",
  numéro: 4,
)

= Introduction
Le nombre  $\pi$ .

== Le cercle trigonométrique
#lorem(20)

== Sinus et cosinus
#lorem(10)

= La fonction arctangente
#lorem(20)
```

Chapitre 4 : Trigonométrie

I. Introduction

Le nombre π .

I.1. Le cercle trigonométrique

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

I.2. Sinus et cosinus

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.

II. La fonction arctangente

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

Remarque : ici, le nom du chapitre est de type chaîne de caractères (et non pas, bloc de contenu) ce qui est plus pratique pour travailler avec ou, par exemple, le donner en métadonnée du PDF via une règle `set document(title: titre)` dans le template. De même le numéro est bien de type nombre entier, pour éventuellement effectuer des calculs dessus.

Je n'ai pas besoin d'en dire beaucoup plus. En fait, créer un template en Typst est *tellement simple*, qu'on en vient vite à multiplier les paramètres facultatifs pour gérer beaucoup de cas (ci-dessus, on pourrait vouloir traiter les cas où le titre ou bien le numéro est `none`), mais aussi créer des méta-templates et des templates dérivés (un de base pour tous ses documents, puis un pour les cours, un pour les feuilles d'exercices, etc). À chacun de tester et d'explorer !

D.2. Utiliser un fichier de données CSV

Le langage Typst permet très facilement d'importer des données, depuis divers formats : texte pur, CSV, XML, JSON, YAML, TOML. Si ces derniers ne vous disent rien, ce sont des formats modernes avec une spécification bien claire pour stocker des paramètres et des configurations. Grâce au riche système de types en Typst (chaînes de caractères, listes, dictionnaires, ...) les données sont chargées et représentées de façons assez directe. On peut même charger un fichier binaire pur, pour les plus programmeurs.

D.2.i. Étude de cas

Nous allons nous concentrer un moment sur les fichiers CSV. Supposons qu'on ait un tel fichier contenant une liste d'élèves avec leur nom, prénom, et leur note au dernier DS. Nous souhaitons écrire un programme qui charge le fichier et le rend dans un tableau. C'est d'une simplicité extrême !

Tout d'abord notre tableau est présenté ainsi dans un fichier `notes.csv`. *Les données ont été générées aléatoirement en utilisant une intelligence artificielle tournant localement.*

```
"Nom","Prénom","Note"
"Martin","Louise","12,4"
"Dupont","Antoine","18,0"
"Lefèvre","Camille","9,7"
"Moreau","Hugo","16,2"
"Laurant","Amélie","14,5"
"Petit","Louis","7,3"
"Girard","Clara","20,0"
"Rousseau","Mathieu","13,9"
"Valois","Sophie","11,6"
"Lambert","Pierre","17,8"
```

Au départ, il est tout de même pratique de garder ses notes dans un tableau LibreOffice. On peut obtenir le fichier CSV simplement depuis « Enregistrer sous ». En fait, j'utilise moi une conversion en ligne de commande (qui du coup est inscrite dans un fichier Makefile : une modification du tableur, et tout est reconstruit !), en supposant que le fichier original s'appelle `notes.ods` :

```
$ libreoffice --convert-to "csv:Text - txt - csv
(StarCalc):44,34,76,1,,0,true" notes.ods
```

Les codes barbares sont décrits dans la [documentation du filtre CSV](#) de LibreOffice et correspondent aux options d'encodage des caractères, délimiteurs du fichier CSV, et d'export en échappant tout entre guillemets.

On charge alors le fichier CSV d'un coup en tant que *liste de dictionnaires*, les noms des clés utilisant les en-têtes du fichier CSV.

```
#let data = csv("notes.csv", row-type: dictionary)
```

L'objet `data` ressemble à :

```
(
("Nom": "Martin", "Prénom": "Louise", "Note": "12,4"),
("Nom": "Dupont", "Prénom": "Antoine", "Note": "18,0"),
("Nom": "Lefèvre", "Prénom": "Camille", "Note": "9,7"),
...
)
```

On peut alors parcourir les lignes, par exemple avec une boucle **for**, mais nous allons plutôt utiliser une méthode `.map()`. Si la variable `x` désigne une ligne (un élément de la liste `data` ci-dessus) alors on accède aux cases correspondantes par `x.at("Nom")`, `x.at("Prénom")`, `x.at("Note")`.

Il s'agit alors de rendre cette donnée en tableau à l'aide de l'élément `table()`. Celui-ci prend ses arguments sous forme d'une *liste variable d'arguments*, chacun représentant une cellule ; il a absolument besoin de connaître en avance via son option `columns` le nombre de colonnes (voir leur taille). On doit donc transformer notre objet `data` :

- D'abord en une liste de listes de cellules (chaque ligne comportera deux cellules, car nous regrouperons les noms et prénoms, au passage en capitalisant le nom) avec la méthode `.map()` prenant en argument une fonction à appliquer sur chaque élément du tableau. Nous allons le transformer en le tableau suivant :

```
(
([MARTIN Louise], [12,4]),
([DUPONT Antoine], [18,0]),
([LEFÈVRE Camille], [9,7]),
...
)
```

- Puis l'aplatir en une seule longue liste avec la méthode `.flatten()`, à l'issue de laquelle on obtiendra

```
([MARTIN Louise], [12,4], [DUPONT Antoine], [18,0], [LEFÈVRE Camille],
[9,7], ...)
```

- Et enfin transformer un objet de type `array` (liste) en un objet de type *liste d'arguments de taille variable* (ce que reçoivent les fonctions ayant un nombre d'arguments variables) avec la syntaxe `..data`.

L'exemple minimaliste donne ceci (effectivement, c'est court !) :

```
#let data = csv("notes.csv", row-type: dictionary)
#align(center, table(
  columns: 2,
  [NOM Prénom],
  [Note],
  ..data.map(x => (
    [#upper(x.at("Nom")) #x.at("Prénom")],
    [#x.at("Note")]
  )).flatten()
))
```

NOM Prénom	Note
MARTIN Louise	12,4
DUPONT Antoine	18,0
LEFÈVRE Camille	9,7
MOREAU Hugo	16,2
LAURANT Amélie	14,5
PETIT Louis	7,3
GIRARD Clara	20,0
ROUSSEAU Mathieu	13,9
VALOIS Sophie	11,6
LAMBERT Pierre	17,8

Améliorons-le un petit peu :

- L'argument donné à `columns` peut être soit le nombre de colonnes (nécessaire pour mettre en forme la liste de cellules en un tableau à deux dimensions !) soit une liste donnant la largeur de chaque colonne. La largeur peut être indiquée avec les unités habituelles (centimètre **cm**, point **pt**, taille relative **em**) mais aussi en « fraction de l'espace disponible » **fr** ce qui est très pratique pour créer des tableaux occupant harmonieusement l'espace (quitte à emboîter les tableaux dans un **block** ou un **pad**).

- Le paramètre `fill` (couleur de fond) prend en argument une unique couleur, ou une liste de couleurs (une par colonne) ou bien une fonction à deux variables qui recevra les numéros de colonnes et de lignes (dans cet ordre, numérotés à partir de 0) et qui renverra une couleur. Cela rend certaines constructions élégantes (lignes de couleurs alternantes) extrêmement simples à mettre en place !
- Il en va de même pour l'argument `align` (alignement du contenu des cellules).
- La fonction `table.cell`, qui n'existe qu'à l'intérieur de la fonction `table`, permet de configurer directement les propriétés d'une cellule (comme ci-dessus : couleur, alignement) en écrasant les propriétés globales.
- La ligne d'en-tête devrait être englobée dans une fonction `table.header`. Cela permet de créer une ligne d'en-tête qui se répète sur chaque page, si le tableau doit être coupé.
- Les notes données ici sont de type chaînes de caractères. Pour effectuer un calcul dessus il faut les transformer en flottants. Jouons à afficher en rouge les notes en dessous de 10, et en vert celles au-dessus de 14.

```
#let data = csv("notes.csv", row-type: dictionary)

#align(center, table(
  columns: (1fr, 8em),
  align: (j, i) => if i == 0 or j == 1 { center } else
{ left },
  fill: (_, i) => if i == 0 { gray },
  table.header([*NOM Prénom*], [*Note*]),
  ..data.map(x => (
    [#upper(x.at("Nom")) #x.at("Prénom")],
    {
      let n = float(x.at("Note").replace(",", "."))
      table.cell(
        fill: if n < 10 { red } else if n >= 14
{ green } else { silver },
        [#x.at("Note")]
      )
    }
  )).flatten()
))
```

NOM Prénom	Note
MARTIN Louise	12,4
DUPONT Antoine	18,0
LEFÈVRE Camille	9,7
MOREAU Hugo	16,2
LAURANT Amélie	14,5
PETIT Louis	7,3
GIRARD Clara	20,0
ROUSSEAU Mathieu	13,9
VALOIS Sophie	11,6
LAMBERT Pierre	17,8

D.2.ii. Diverses pistes d'amélioration

Avec toutes ces méthodes, j'utilise des tableurs LibreOffice uniquement pour entrer les données, mais toutes mes mises en forme sont faites par des tableaux Typst. Le langage permet d'ailleurs de faire du calcul sur les données (trier le tableau, filtrer, etc).

En utilisant des idées similaires, il est facile de fabriquer un programme de trombinoscope : il faut d'une part la liste des élèves en fichier CSV à charger, et d'autre part un dossier contenant leurs photos. Le programme place en tableau les photos et les noms.

D'autres types de données peuvent être chargées et traitées. On peut imaginer des solutions très créatives pour charger un code source (fonction `read()`, texte pur), le découper en lignes (méthode `.split("\n")`), sélectionner certaines lignes (méthode `.filter()`), pour recoller le code (méthode `.join("\n")`) et l'afficher en bloc. Voir le code source du présent document.

D.3. Passer des arguments en ligne de commande

En donnant au compilateur Typst un argument `--input clé=valeur`, on peut passer à son fichier `.typ` une variable globale, qu'on récupère dans en tant que chaîne de caractère `sys.inputs.at("clé")`. C'est une idée très simple et utile... et très compliquée à faire en LaTeX !

L'application qui me vient tout de suite en tête, c'est d'écrire un seul fichier `exercices.typ` qui contiendrait à la fois des exercices et leur correction, et pouvoir compiler le fichier soit avec soit sans les corrections !

Détaillons cette situation. Nous créons un exemple de fichier `exercices.typ`, dans lequel il est d'abord pratique de transformer le paramètre (de type chaîne de caractères) en un booléen au tout début du fichier, pour pouvoir traiter soigneusement les cas d'erreurs.

```
#let avec_correction = {
  let mode = sys.inputs.at("correction", default: "avec")
  assert(mode == "avec" or mode == "sans", message: "Indiquer correction=avec
ou correction=sans")
  if mode == "avec" { true } else { false }
}
```

Puis on crée notre fonction qui prend en argument à la fois un énoncé d'exercice, et une correction (ici, les deux arguments sont obligatoires ; on pourrait facilement rendre la correction optionnelle). On utilise, pourquoi pas, un élément `divider()` pour afficher une ligne horizontale de séparation.

```
#let exercice(énoncé, correction) = {
  block(width: 100%, stroke: 0.5pt, inset: 6pt, {
    [*Exercise*]
    linebreak()
    énoncé
    if avec_correction {
      divider()
      [*Correction*]
      linebreak()
      correction
    }
  })
}
```

Il n'y a plus alors qu'à utiliser la fonction, avec la syntaxe « blocs de contenus collés hors de la fonction ».

```
#exercice[
  Montrer que si  $n \in \mathbb{Z}$  est un entier impair, alors  $n^2$  est impair.
][
  Supposons  $n$  impair : il existe  $k \in \mathbb{Z}$  tel que  $n = 2k+1$ . Alors
   $n^2 = (2k+1)^2 = 4k^2 + 4k + 1 = 2 \times (2k^2 + 2k) + 1$ 
  où  $2k^2 + 2k \in \mathbb{Z}$ . Donc  $n^2$  est impair.
]
```

Compilons-le deux fois (sans oublier de donner des noms différents aux fichiers de sortie !)

```
$ typst compile --input correction="avec" exercices.typ exercices-avec.pdf
$ typst compile --input correction="sans" exercices.typ exercices-sans.pdf
```

Le rendu des deux fichiers côte-à-côte :

Exercice Montrer que si $n \in \mathbb{Z}$ est un entier impair, alors n^2 est impair. Correction Supposons n impair : il existe $k \in \mathbb{Z}$ tel que $n = 2k + 1$. Alors $n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2 \times (2k^2 + 2k) + 1$ où $2k^2 + 2k \in \mathbb{Z}$. Donc n^2 est impair.	Exercice Montrer que si $n \in \mathbb{Z}$ est un entier impair, alors n^2 est impair.
---	--

Dans notre exemple, si la condition du `if avec_correction` est fausse, alors aucun contenu de correction n'est posé (on pourrait rajouter explicitement `else { none }`, le type `none` étant interprété comme un contenu vide) et tout ce passe comme s'il n'y avait rien du tout. Faire disparaître du contenu en Typst est extrêmement facile : il suffit de ne pas le poser !

Alternativement, on pourrait vouloir laisser en blanc la place correspondante. On peut alors emboîter le contenu de correction dans un élément `hide` qui comme son l'indique cache son contenu, tout en lui réservant la place dans la mise en page (et aussi, en incrémentant bien les compteurs qui y sont cachés). On propose donc la variante suivante de la fonction `exercice()` :

```
#let exercice(énoncé, correction) = {
  block(width: 100%, stroke: 0.5pt, inset: 6pt, {
    [*Exercice*]
    linebreak()
    énoncé
    divider()
    let x = {
      [*Correction*]
      linebreak()
      correction
    }
    if avec_correction { x } else { hide(x) }
  })
}
```

Le reste ne change pas ; la compilation deux fois donne (les deux documents sont parfaitement alignés) :

Exercice Montrer que si $n \in \mathbb{Z}$ est un entier impair, alors n^2 est impair. Correction Supposons n impair : il existe $k \in \mathbb{Z}$ tel que $n = 2k + 1$. Alors $n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2 \times (2k^2 + 2k) + 1$ où $2k^2 + 2k \in \mathbb{Z}$. Donc n^2 est impair.	Exercice Montrer que si $n \in \mathbb{Z}$ est un entier impair, alors n^2 est impair.
---	---

En poussant plus loin, on peut réellement faire fonctionner un programme Typst comme un programme à part entière, quitte à le faire interagir avec un petit script Bash ou Python !

D.4. Export multiple

Il s'agit de la grande nouveauté de la version 0.15.

D.4.i. Export HTML

La possibilité d'export HTML a été annoncée dès le tout départ, mais a mis du temps à être développée. La version 0.15 a fait grandement avancer le sujet en introduisant l'export des mathématiques en MathML. Comme tout ceci est encore considéré comme « expérimental », il faut activer la fonctionnalité explicitement avec l'argument `--features html` pour la compilation :

```
$ typst compile --features html --format html fichier.typ
```

L'ensemble est de bonne qualité, mais l'apparence est un peu froide. Il y a un problème évident : alors que le PDF sert à créer un document sur une page de taille fixe, et doit être présenté partout de la même manière, un fichier HTML peut au contraire s'adapter à toutes les tailles d'écran et se reformater à volonté. Comment alors rendre les éléments blocs avec leur taille fixe ? Actuellement, un certain nombre d'éléments et de paramètres de mise en forme sont simplement ignorés.

Cependant, la fonction `target()` renvoie le type de document en train d'être créé ; et de nombreuses fonctions dépendantes du module `html` permettent de créer explicitement des éléments HTML dans le document rendu. Il est donc possible, si on le souhaite et qu'on y passe du temps, d'obtenir un rendu HTML de toute aussi haute qualité que le PDF. Certains personnes ont déjà développé les outils pour écrire facilement des billets de blogs en Typst, voire des sites entiers.

D.4.ii. Export en *bundles*

Dans sa dernière version, Typst introduit le *bundle export*. Il s'agit de la possibilité de générer plusieurs documents, PDF ou HTML, ou les deux mélangés, voire même des fichiers binaires, à partir d'un seul fichier Typst via la fonction `document()` (qui existait déjà, mais servait surtout à définir les métadonnées du PDF via une règle `set`). Un gros avantage pour le HTML, c'est d'avoir un ensemble de fichiers `.html` avec les liens hypertextes tous cohérents entre eux ! Un autre avantage est de pouvoir bénéficier de la compilation incrémentale (mode *watch*), même

si en pratique les pages HTML sont dans des fichiers `.typ` différents qui s'importent les uns les autres.

Mais nous allons étudier un cas concret de génération de fichiers PDF multiples. Supposons qu'on ait dans un fichier CSV une liste d'élève avec, cette fois, leur adresse. On veut créer pour chacun une lettre personnalisée (il s'agit d'un exercice avancé de mon TP sur les fichiers en Python). Par exemple le CSV ressemble à ceci. *Encore une fois, ces données ont été générées aléatoirement par intelligence artificielle tournant localement, ces adresses et ces élèves n'existent pas.*

```
"Nom","Prénom","Adresse","Ville"
"Dupont","Éloïse","12 Rue des Vignes","Versailles"
"Leroux","Mathieu","48 Avenue de la République","Le Chesnay"
"Martin","Sophie","33 Rue de la Paix","St-Cyr l'École"
"Bernard","Julien","21 Rue de la Gare","La Celle-Saint-Cloud"
"Leclerc","Clara","5 Place des Arts","Versailles"
"Moreau","Paul","19 Rue des Jardins","Le Chesnay"
"Girard","Emma","14 Boulevard de l'Étoile","St-Cyr l'École"
"Renard","Lucas","30 Rue des Champs","La Celle-Saint-Cloud"
"Cohen","Alison","7 Rue du Parc","Versailles"
"Petit","Thomas","11 Avenue des Lumières","Le Chesnay"
```

Il faut d'abord savoir créer une lettre, par exemple avec une fonction prenant en argument toutes les données :

```
#let lettre(nom, prénom, adresse, ville) = {
  align(right, block([
    #prénom #upper(nom) \
    #adresse \
    #upper(ville)
  ]))
  align(center, strong[Lettre pour #prénom #upper(nom)])
  lorem(30)
}
```

Les règles `set` définies en dehors seront effectives pour tous les fichiers créés. Il suffit ensuite de charger le fichier de données et de créer une lettre pour chacun, par exemple ici avec une boucle `for`, et un peu de syntaxe en plus pour numéroter correctement (variable `n`) les fichiers produits :

```
#let data = csv("adresses.csv", row-type: dictionary)

#for (n, x) in data.enumerate(start: 1) {
  document(
    "lettre-" + str(n) + ".pdf",
    lettre(x.at("Nom"), x.at("Prénom"), x.at("Adresse"), x.at("Ville"))
  )
}
```

Compiler avec la commande :

```
$ typst compile --features bundle --format bundle lettres.typ
```

Par défaut les fichiers sont créés dans un même dossier, nommé d'après le fichier `.typ`. Le résultat, pour les deux premières lettres `lettres/lettre-1.pdf` et `lettres/lettre-2.pdf` :

Éloïse DUPONT 12 Rue des Vignes VERSAILLES Lettre pour Éloïse DUPONT Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aequa doleamus animo, cum corpore dolemus, fieri.	Mathieu LEROUX 48 Avenue de la République LE CHESNAY Lettre pour Mathieu LEROUX Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aequa doleamus animo, cum corpore dolemus, fieri.
---	--

Comme on le constate, un seul fichier Typst fonctionne réellement comme un programme qui peut remplacer d'un tout toute une chaîne plus complexe qu'on obtiendrait avec LaTeX, Bash, Python, etc.

D.5. Quelques paquets utiles

Les paquets sont disponibles en ligne sur le [Typst Universe](#) et sont téléchargés automatiquement par le compilateur lors de leur importation (il suffit donc de copier-coller les codes et ça marche). Je vais décrire quelques paquets forts intéressants pour nous, et qui ont une réputation bien établie dans la communauté Typst.

- [CeTZ](#) prétend remplacer le paquet TikZ en LaTeX, pour créer de magnifiques diagrammes et illustrations avec une syntaxe Typst. Je ne l'utilise pas car j'apprends à produire et importer des illustrations au format SVG, avec le logiciel Inkscape ou bien par des programmes Python/Matplotlib ou SageMath.
- [VarTable](#) et [FuncTable](#) pour créer de magnifiques tableaux de variation, une fonctionnalité évidemment importante pour un enseignant ! J'ai testé le premier, mais le second semble plus récent. Le rendu est de bonne qualité ; pour les utiliser, il faut la documentation et un exemple sous la main. Pour VarTable (d'ailleurs basé sur CeTZ) :

```
#import "@preview/vartable:0.2.4": tabvar

$ #tabvar(
  table-style: (stroke: 0.5pt),
  arrow-style: (stroke: 0.5pt),
  line-style: (stroke: 0.5pt),
  first-line-height: 0.8cm,
  first-column-width: 2cm,
  element-distance: 2cm,
  variable: $x$,
  label: (($x$, 0.8cm, "s"), ($A(x)$, 0.8cm, "s"),
($B(x)$, 0.8cm, "s"), ($f'(x)$, 0.8cm, "s"),
($f(x)$, 2cm, "v"), ),
  domain: ($-\infty$, $-1$, $0$, $1$, $+\infty$, ),
  contents: (
    ($-$, $-$, ("0", $+$), $-$),
    ($+$, $+$, $+$, ("0", $-$)),
    ($-$, ("0", $+$), $+$, $+$),
    ($+$, ("0", $-$), ("0", $+$), ("0", $-$)),
    (
      (bottom, $-\infty$, (top, $1$), (bottom, $0$),
      (top, $2$), (bottom, $0$),
    ),
  ),
) $
```

x	$-\infty$	-1	0	1	$+\infty$
x	$-$	$-$	0	$+$	$-$
$A(x)$	$+$	$+$	$+$	0	$-$
$B(x)$	$-$	0	$+$	$+$	$+$
$f'(x)$	$+$	0	$-$	0	$+$
$f(x)$	$-\infty$	1	0	2	0

- [Touying](#) permet de faire des slides, comme Beamer pour LaTeX.
- [Codly](#) pour améliorer la mise en forme du code, qui intéressera certainement ceux qui enseignent l'informatique. La présentation par défaut :

```
#import "@preview/codly:2.0.0" as codly

#show raw.where(block: true): codly.new

```python
def fibonacci(n):
 assert n >= 0, "n doit être positif"
 # u et v représentent
 # deux termes successifs
 u = 0
 v = 1
 for i in range(n):
 t = u + v
 v = t
 u = v
 return u
```
```

```
1 def fibonacci(n):
2     assert n >= 0, "n doit être positif"
3     # u et v représentent
4     # deux termes successifs
5     u = 0
6     v = 1
7     for i in range(n):
8         t = u + v
9         v = t
10        u = v
11    return u
```

- **Unify** intéressera les enseignants en physique-chimie pour mettre en forme correctement les unités et les grandeurs en notation scientifique.

Les enseignants de chimie y trouveront tout pour rendre leurs molécules ; les enseignants d'informatique trouveront de quoi rendre les automates finis, les graphes et les arbres de preuves ; les enseignants en mathématiques y trouveront des paquets pour mettre en forme des théorèmes, parfois avec de jolis cadres (remplacement à `tcolorbox` en LaTeX). Il y a dans l'ensemble tout ce dont on a besoin.

On trouve aussi de nombreux modèles de CV ou de lettres, d'ailleurs pas tous aussi bien aboutis. Il n'est pas toujours nécessaire d'importer ces paquets : parfois, le code est tellement simple et lisible, qu'il suffit d'en extraire et copier-coller la partie intéressante.

Certains paquets sont plus exotiques, voire franchement très exotiques : un jeu de Tetris en Typst, des projets de calcul formel ou d'interpréteurs de langages entièrement rédigés en Typst, et même des interpréteurs LaTeX. Je ne sais pas s'ils sont réellement utiles, mais ce sont des *proof of concepts* de Typst et l'ensemble est très impressionnant ! C'est parce que la communauté Typst comprend de nombreux bidouilleurs qui ont très vite cherché à explorer toutes les possibilités du langage.

E. Quelques limitations

Pour être honnête, je me dois de décrire les limitations actuelles du langage Typst, qui pour le coup sont des choses bien implémentées en LaTeX.

E.1. Gestion très fine des espacements

Le langage TeX/LaTeX a été affiné sur des dizaines d'années pour offrir une typographie impeccable. Les paramètres permettant de décider quand couper une ligne, quand un morceau est autorisé à déborder dans la ligne, les *badness level* et autres, sont réglés extrêmement finement ; une notion de *glue* permet d'élargir ou de resserrer des espaces pour que le contenu occupe harmonieusement la page et ne la coupe pas n'importe où. Les paquets `amsmath` puis `mathtools` fournissent eux aussi de nombreux affinages typographiques pour les mathématiques.

Peut-on retrouver ce comportement avec Typst ? Je dirais que l'espacement « horizontal » est assez bien géré, les phrases sont correctement découpées et la césure est dans l'ensemble satisfaisante. Les formules mathématiques sont trop fréquemment coupées : on peut l'empêcher en les emboitant dans un élément `box`, voire en emboitant automatiquement toutes les formules dans des `box` (attention alors à ne pas écrire des formules trop longues) :

```
show math.equation.where(block: false): box
```

et de même pour les éléments `raw` en ligne. Les fonctions `text` et `par` admettent malgré cela de nombreuses options (que je ne décris pas du tout ici) pour contrôler finement la typographie, l'alignement et la césure.

L'espacement vertical n'est, il est vrai, pas particulièrement intelligent : les blocs sont posés les uns à la suite des autres, sans vrai soucis d'harmoniser la page, si ce n'est l'option de rendre

certaines blocs incassables (propriété `breakable`: `false`, désactivée par défaut mais activée ici sur le code) et de forcer à ce que l'élément ne se retrouve pas seul en bas de page mais colle à la suite (propriété `sticky`: `true`). Pour un article sur plusieurs pages ce n'est pas toujours grave ; pour un document à imprimer et à distribuer aux élèves, cela peut nécessiter d'ajouter des sauts manuels de page ou de colonne et des blocs incassables.

En contrepartie, les éléments `block` ou `grid` ainsi que la notion de taille « fraction de l'espace restant » permettent une mise en forme bien millimétrée pour réaliser une affiche, ou un formulaire ou un Mémo à imprimer qui doit tenir sur une seule page avec des beaux cadres.

Enfin la typographie des mathématiques est globalement de très bonne qualité et on peut déjà faire des réglages très fins, comme les classes de caractères et les délimiteurs de taille variables. Ce n'est pas cela qui peut être limitant.

E.2. Éléments personnalisés

Les règles `set` sont très utiles, mais ne sont applicables que sur les fonctions déjà prévues pour et pas sur celles créées par l'utilisateur. Elles remplacent en quelque sorte des paramètres globaux : ceux-ci n'existent pas en Typst (du moins, ne pourraient pas être modifiés hors de la portée d'une fonction). Par exemple, à la manière du paquet LaTeX `amsthm`, on aimerait créer une fonction `théorème()` permettant comme son nom l'indique de mettre en forme un théorème, prenant comme argument au minimum le contenu et un titre optionnel. On aimerait alors donner à l'utilisateur la possibilité d'en régler globalement quelques paramètres (encadré ou non ? style de numérotation ? dans un bloc en couleurs ?) idéalement avec une règle `set` qui serait appelée au début du fichier ou bien via un template.

Actuellement, c'est un petit peu compliqué à réaliser : les règles `set` ne s'appliquent que sur les *element functions* (celles qui créent du contenu, et sont un peu particulières) et sur leurs propriétés *settable*. Une possibilité est de créer une fonction méta `_théorème` avec beaucoup d'options et de laisser l'utilisateur en dériver les siennes, par exemple `let lemme = _théorème.with(nom: "Lemme")`.

Les références, que je n'ai pas abordées, fonctionnent aussi uniquement sur les fonctions prévues pour. On peut s'en sortir avec des éléments `figure` qui eux sont référençables. Cela pose aussi quelques soucis pour les compteurs, actuellement il n'est pas aisé de créer un compteur de théorèmes qui soit automatiquement dépendant du compteur de sections comme avec `amsthm`. On s'en sort quand même, soit en faisant les choses à la main sans grandes ambitions, soit en utilisant des paquets qui traitent ce problème...

La possibilité pour l'utilisateur de créer ses propres *elements functions* est une fonctionnalité très demandée, qui est certainement en réflexion et apparaîtra dans les versions futures de Typst.

E.3. Quelques mots sur l'intelligence artificielle

Je ne suis pas très expérimenté pour parler de ce sujet.

Le langage Typst est apparu en même temps que ChatGPT pour le grand public. Au départ, celui-ci ne connaissait donc pas Typst, mais rapidement les bidouilleurs lui ont donné à manger toute la documentation en PDF et ont réussi à produire du code. Il semble que les modèles d'intelligence artificielle récents connaissent correctement Typst et permettent une certaine productivité à ceux qui aiment ces outils.

D'autre part, actuellement les outils d'intelligence artificielle sont efficaces pour générer du LaTeX, éventuellement en convertissant directement depuis divers documents voire des notes manuscrites. On peut donc déjà obtenir un très beau rendu sans connaissances poussées en TeX.

Faut-il pour autant abandonner l'apprentissage de Typst et laisser la main à l'IA pour écrire du LaTeX ? Certainement pas : même s'il était possible de déléguer entièrement à une IA la génération du code, le compilateur Typst (ainsi que tout l'écosystème Rust) a infiniment plus d'avantages.

F. Conclusion

À la question initiale de savoir si Typst peut bien remplacer LaTeX, j'espère avoir répondu : **oui**, et dès maintenant, les avantages surpassant de très loin les manques, qui seront certainement comblés dans des versions futures.

Une frustration au premier abord peut venir du fait que nous sommes *tellement habitués* à LaTeX que nous voulons à tout prix que Typst nous donne exactement le même rendu. Cela est à peu près possible, mais je voudrait aussi plaider une chose : c'est l'occasion de **repenser nos habitudes typographiques** ! Les blocs, boîtes, grilles, permettent de faire très rapidement des mises en forme originales et artistiques que même en rêve nous n'aurions même pas pu imaginer avec LaTeX tant ce langage est compliqué. Si au niveau Master on peut se satisfaire d'un document froid fait de `\section{}` empilés, je m'adresse plutôt là à ceux qui enseignent à des jeunes étudiants. C'est l'occasion d'être créatif sur la mise en page, tout en programmant et en gardant une typographie de haute qualité !

Un autre problème évident, c'est l'« inertie » à bouger tout l'écosystème existant en TeX. Évidemment que le monde de la publication scientifique ne pas pas basculer de si tôt vers une alternative. Mais je voudrais là encore m'adresser particulièrement aux enseignants : nous, nous ne dépendons pas d'une grosse machine de production au-dessus qui nous impose comment rédiger nos documents, et la plupart sont d'ailleurs de fort peu d'enjeu puisqu'ils sont simplement distribués à une classe d'élèves. Nous sommes totalement libres. **Cette liberté nous permet justement d'expérimenter les toutes dernières technologies**, et de vouloir les diffuser quand elles en valent la peine. Ainsi comme je l'ai dit, j'ai démarré Typst par des documents peu intéressants avant d'être convaincu et de vouloir convertir tout mon contenu. Bien sûr les enseignants bien en place qui ont tout leur contenu 100 % prêt ne se donneront pas cette peine, mais si vous avez des documents nouveaux à réaliser, c'est l'occasion d'essayer !

Enfin pour ceux qui ont lu jusque là, je vais publier le code source de ce document et je vous encourage à aller le lire, vous inspirer, et partager vos créations !