

MEANDER

User guide

v0.4.0

2026-01-21

MIT

Page layout engine with image wrap-around and text threading.

NEVEN VILLANI

✉ neven@crans.org

MEANDER implements a content layout algorithm that supports automatically wrapping text around figures, and with a bit of extra work it can handle images of arbitrary shape. In practice, this makes **MEANDER** a temporary solution to [issue #5181](#). When Typst eventually includes that feature natively, either **MEANDER** will become obsolete, or the additional options it provides will be reimplemented on top of the builtin features, greatly simplifying the codebase.

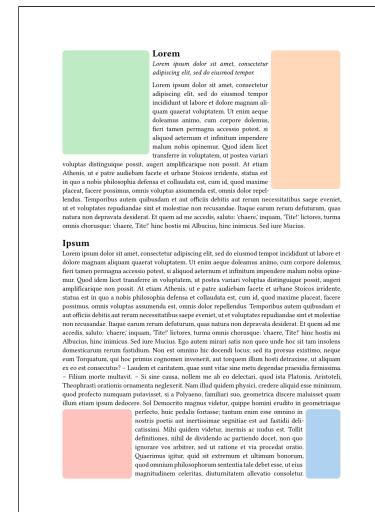
Though very different in its modeling, **MEANDER** can be seen as a Typst alternative to L^AT_EX's `wrapfig` and `parshape`, effectively enabling the same kinds of outputs.

Contributions

If you have ideas for improvements, or if you encounter a bug, you are encouraged to contribute to **MEANDER** by submitting a [bug report](#), [feature request](#), or [pull request](#).

Versions

- [dev](#)
- [0.4.0 \(latest\)](#)
- [0.3.1](#)
- [0.3.0](#)
- [0.2.5](#)
- [...](#)



To migrate existing code, please consult [Section I.6](#) and [Section I.7](#)

Table of Contents

Quick start	3
I.1 A simple example	3
I.2 Multiple obstacles	4
I.3 Columns	5
I.4 Anatomy of an invocation	5
I.5 Going further	6
I.6 0.2.x Migration Guide	6
I.7 0.3.x Migration Guide ⁽⁺⁾	7
Understanding the algorithm	8
II.1 Debugging	8
II.2 Page tiling	8
II.3 Content bisection	9
II.4 Threading	10
Contouring	11
III.1 Margins	11
III.2 Boundaries as equations	12
III.3 Boundaries as layers ^(!)	14
III.3.1 Horizontal rectangles	14
III.3.2 Vertical rectangles	15
III.4 Autocontouring	16
III.5 More to come	18
Styling	19
IV.1 Paragraph justification	19
IV.2 Font size and leading	20
IV.3 Hyphenation and language	21
IV.4 Styling containers	22
Multi-page setups	23
V.1 Pagebreak	23
V.2 Colbreak	23
V.3 Colfill	24
V.4 Placement	25
V.5 Overflow	26
V.5.1 No overflow	27
V.5.2 Predefined layouts	28
V.5.3 Custom layouts	28
Inter-element interaction	31
VI.1 Locally invisible obstacles	31
VI.2 Callbacks and queries ^(!)	32
VI.3 A nontrivial example	33
Showcase	34
VII.1 Side illustrations ^(!)	34
VII.2 Paragraph packing ^(!)	35
VII.3 Drop caps ⁽⁺⁾	36
Public API	37
VIII.1 Elements ^(!!)	37
VIII.2 Layouts	41
VIII.3 Contouring	41
VIII.4 Queries ^(!)	44
VIII.5 Options	45
VIII.5.1 Pre-layout options	45
VIII.5.2 Dynamic options	46
VIII.5.3 Post-layout options	47
VIII.6 Public internals	48
Internal module details	49
IX.1 Utils	49
IX.2 Types	50
IX.3 Geometry	50
IX.4 Tiling	54
IX.5 Bisection	57
IX.6 Threading	64
About	65
X.1 Related works	65
X.2 Dependencies	65
X.3 Acknowledgements	65

Highlighted chapters denote breaking changes^(!), major updates^(!!), minor updates^(!), and new additions⁽⁺⁾, in the latest version 0.4.0

Part I

Quick start

Import the latest version of `MEANDER` with:

```
#import "@preview/meander:0.4.0"
```

The main function provided by `MEANDER` is `#meander.reflow`, which takes as input a sequence of “containers”, “obstacles”, and “flowing content”, created respectively by the functions `#container`, `#placed`, and `#content`. Obstacles are placed on the page with a fixed layout. After excluding the zones occupied by obstacles, the containers are segmented into boxes then filled by the flowing content.

More details about `MEANDER`’s model are given in [Section II](#).

I.1 A simple example

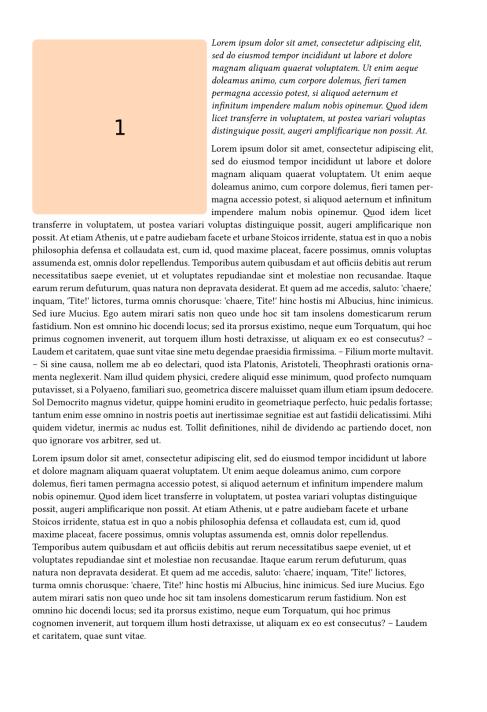
Below is a single page whose layout is fully determined by `MEANDER`. The general pattern of `#placed + #container + #content` is almost universal.

```
#meander.reflow({
  import meander: *
  // Obstacle in the top left
  placed(top + left, my-img-1)

  // Full-page container
  container()

  // Flowing content
  content[
    _#lorem(60)_]
    #[

      #set par(justify: true)
      #lorem(300)
    ]
    #lorem(200)
  ]
})
```



Within a `#meander.reflow` block, use `#placed` (same parameters as the standard function `#place`) to position obstacles made of arbitrary content on the page, specify areas

where text is allowed with `#container`, then give the actual content to be written there using `#content`.

MEANDER is expected to automatically respect the majority of styling options, including headings, paragraph justification, bold and italics, etc. Notable exceptions that must be specified manually are detailed in [Section IV](#).

If you find a style discrepancy, make sure to file it as a [bug report](#), if it is not already part of the [known limitations](#).

I.2 Multiple obstacles

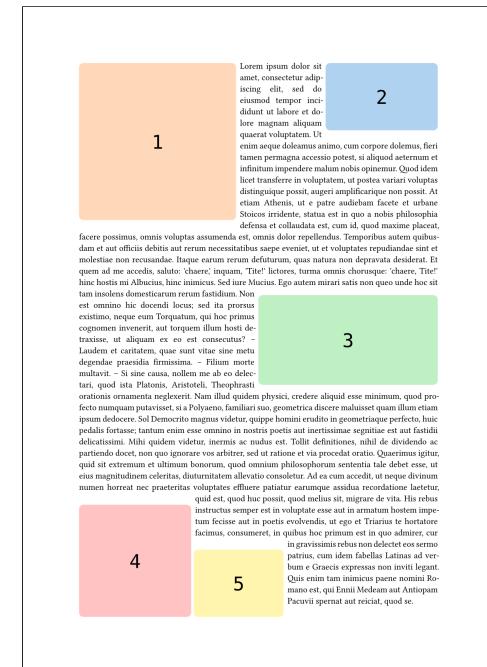
A single `#meander.reflow` invocation can contain multiple `#placed` objects. A possible limitation would be performance if the number of obstacles grows too large, but experiments have shown that up to ~100 obstacles is still workable.

In fact, this ability to handle arbitrarily many obstacles is what I consider **MEANDER**'s main innovation compared to [WRAP-IT](#), which also provides text wrapping but around at most two obstacles.

```
#meander.reflow({
    import meander: *

    // As many obstacles as you want
    placed(top + left, my-img-1)
    placed(top + right, my-img-2)
    placed(horizon + right, my-img-3)
    placed(bottom + left, my-img-4)
    placed(bottom + left, dx: 32%,
           my-img-5)

    // The container wraps around all
    container()
    content[
        #set par(justify: true)
        #lorem(430)
    ]
    1
})
```



Technically, **MEANDER** can only handle rectangular obstacles. However, thanks to this ability to wrap around an arbitrary number of obstacles, we can approximate a non-rectangular obstacle using several rectangles. See concrete applications and techniques for defining these rectangular tilings in [Section III](#).

I.3 Columns

Similarly, `MEANDER` can also handle multiple occurrences of `#container`. They will be filled in the order provided, leaving a (configurable) margin between one and the next. Among other things, this can allow producing a layout in columns, including columns of uneven width (a longstanding [typst issue](#)).

```
#meander.reflow({
    import meander: *
    placed(bottom + right, my-img-1)
    placed(center + horizon, my-img-2)
    placed(top + right, my-img-3)

    // With two containers we can
    // emulate two columns.

    // The first container takes 60%
    // of the page width.
    container(width: 60%, margin: 5mm)
    // The second container automatically
    // fills the remaining space.
    container()

    content[#lorem(470)]
})
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit animi et labore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

3

melius sit, migrare de vita. His rebus institutus semper est in voluptate esse aut in armatum hostem impetrare. Ita usque in poetas evocatores, ita ergo in te hospitare factum, consumetur, in quibus hoc primum est in quo admirari, cor in gravissima rebus non detecta eos sermo patrius, cum idem fabellas Latinas ad verbum e Graecis expressas non possit legant. Quis enim inveniret, quod non in Romano est, qui Enni Medean aut An-

2

tiopam Pacuvii spernat aut recitat, quod se idem Euripidis fabulis detectri dicit, Latinas litteras oderit? Synephebos ego, nonne? Caelestis aut Andram Terrenus quan- utrumque Monandri legant? A quibus tantum dissentio, ut, cum Sophocles vel optime scripsisset Electram, tamen male conversam Attili mili legendam putem, de quo

1

I.4 Anatomy of an invocation

As you can extrapolate from these examples, every `MEANDER` invocation looks like this:

```
1 #meander.reflow({
2     import meander: *
3     // pre-layout options
4
5     // layout and dynamic options
6
7     // post-layout options
8 })
```

The most important part is the layout, composed of

1. pagebreak-separated pages, each made of
 - containers that can hold content,
 - placed obstacles delimiting regions that cannot hold content.
2. flowing content, which may also be interspersed with

- colbreaks and colfills to have finer control over how containers are filled.

Pre-layout options — currently opt.debug and opt.placement — are configuration settings that come before any layout specification. They apply to the entire layout that follows.

Post-layout options — currently opt.overflow — determine how to close the layout and chain naturally with the text that follows.

Dynamic options are not instantiated yet, but they will be settings that can be updated during the layout affecting all following elements.

I.5 Going further

If you want to learn more advanced features or if there's a glitch in your layout, here are my suggestions.

In any case, I recommend briefly reading [Section II](#), as having a basic understanding of what happens behind the scenes can't hurt. This includes turning on some debugging options in [Section II.1](#).

To learn how to handle non-rectangular obstacles, see [Section III](#).

If you have issues with text size or paragraph leading, or if you want to enable hyphenation only for a single paragraph, you can find details in [Section IV](#).

To produce layouts that span more or less than a single page, see [Section V](#). If you are specifically looking to give `MEANDER` only a single paragraph and you want the rest of the text to gracefully fit around, consult [Section V.4](#). If you want to learn about what to do when text overflows the provided containers, this is covered in [Section V.5](#).

For more obscure applications, you can read [Section VI](#), or dive directly into the module documentation in [Section VIII](#).

I.6 0.2.x Migration Guide

From 0.2.5 to 0.3.0, configuration options have been reworked, phasing out global settings in favor of pre- and post-layout settings. Here is a comparison between old and new to help guide your migration.

pre-0.2.5	post-0.3.0
Debugging	
<code>#meander.regions({...})</code>	(command) <code>opt.debug.pre-thread()</code> (pre)
<code>debug: true</code>	(parameter) <code>opt.debug.post-thread()</code> (pre)
Overflow	
<code>overflow: false</code>	(parameter) <code>opt.overflow.alert()</code> (post)
<code>overflow: true</code>	(parameter) <code>opt.overflow.ignore()</code> (post)

overflow: pagebreak	(parameter)	opt.overflow.pagebreak()	(post)
overflow: text	(parameter)	new default	
overflow: panic	(parameter)	discontinued due to convergence issues	
overflow: repeat	(parameter)	opt.overflow.repeat()	(post)
overflow: state("_")	(parameter)	opt.overflow.state("_")	(post)
overflow: (_ => {})	(parameter)	opt.overflow.custom(_ => {})	(post)
Placement			
placement: page	(parameter)	opt.placement.phantom()	(pre)
placement: box	(parameter)	new default	
placement: float	(parameter)	discontinued	

I.7 0.3.x Migration Guide

From 0.3.1 to 0.4.0, the query module received major reworks. Instead of using query functions directly in the layout elements, you should instead use a callback in conjunction with query as below:

old	new
<pre> placed(query.position(..), ...) container(align: query.position(..), width: query.width(..), height: query.height(..),) </pre>	<pre> callback(pos1: query.position(..), align2: query.position(..), width2: query.width(..), height2: query.height(..), env => { placed(env.pos1, ...) container(align: env.align2, width: env.width2, height: env.height2,), }) </pre>

Although slightly more verbose, this new approach is much more flexible and will reduce the amount of repeated computations. For more details, consult [Section VI.2](#).

Part II

Understanding the algorithm

Although it can produce the same results as parshape in practice, `MEANDER`'s model is fundamentally different. In order to better understand the limitations of what is feasible, know how to tweak an imperfect layout, and anticipate issues that may occur, it helps to have a basic understanding of `MEANDER`'s algorithm(s).

Even if you don't plan to contribute to the implementation of `MEANDER`, I suggest you nevertheless briefly read this section to have an intuition of what happens behind the scenes.

II.1 Debugging

The examples below use some options that are available for debugging.

Debug configuration is a pre-layout option, which means it should be specified before any other elements.

```
1 #meander.reflow({
2   import meander: *
3   opt.debug.pre-thread() // <- sets the debug mode to "pre-thread"
4   // ...
5 })
```

The debug modes available are as follows:

- `release`: this is the default, having no visible debug markers.
- `pre-thread`: includes obstacles (in red) and containers (in green) but not content. Helps visualize the usable regions.
- `post-thread`: includes obstacles (in red), containers, and content. Containers have a green border to show the real boundaries after adjustments (during threading, container boundaries are tweaked to produce consistent line spacing).
- `minimal`: does not render the obstacles and is thus an even more streamlined version of `pre-thread`.

II.2 Page tiling

When you write some layout such as the one below, `MEANDER` receives a sequence of elements that it splits into obstacles, containers, and content.

```
#meander.reflow({
    import meander: *
    opt.debug.post-thread()
    placed(bottom + right, my-img-1)
    placed(center + horizon, my-img-2)
    placed(top + right, my-img-3)
    container(width: 60%)
    container(align: right, width: 35%)
    content[#lorem(470)]
})
```

First the `#measure` of each obstacle is computed, their positioning is inferred from the alignment parameter of `#placed`, and they are placed on the page. The regions they cover as marked as forbidden.

Then the same job is done for the containers, marking those regions as allowed. The two sets of computed regions are combined by subtracting the forbidden regions from the allowed ones, giving a rectangular subdivision of the usable areas.

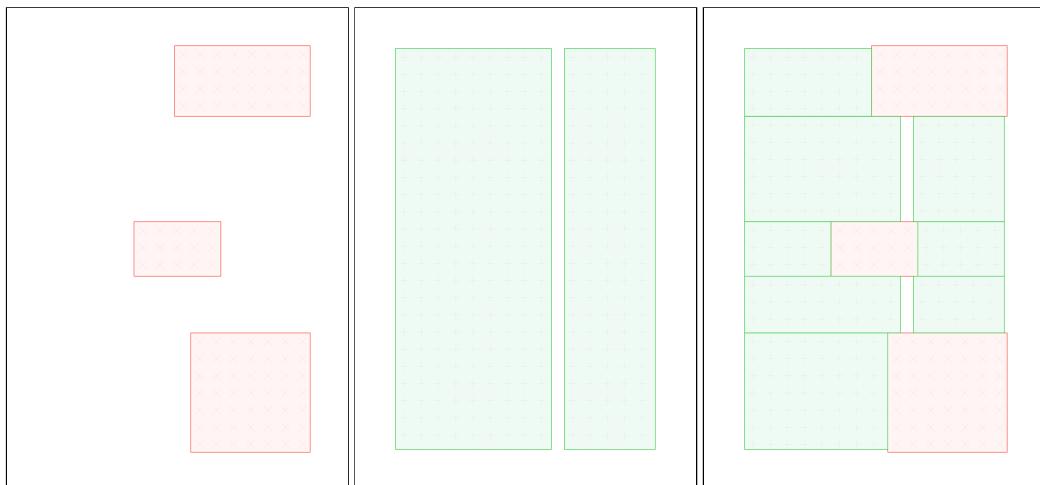


Figure 1: Left to right: the forbidden, allowed, and combined regions.

II.3 Content bisection

The second building block of `MEANDER` is its algorithm to split content. The regions computed by the tiling algorithm must be filled in order, and text from one box might overflow to another. The content bisection rules are all `MEANDER`'s heuristics to split text and take as much as fits in a box.

For example, consider the content `bold(lorem(20))` which does not fit in the container `box(width: 5cm, height: 5cm)`:

**Lorem ipsum dolor sit
 amet, consectetur adipi-
 iscing elit, sed do eius-
 mod tempor incididunt
 ut labore et dolore mag-
 nam aliquam quaerat.**

`MEANDER` will determine that

1. the content fits in the box until “eius-”, and everything afterwards is overflow,
2. splitting `#strong` text is equivalent to applying `#strong` to both halves,
3. therefore the content can be separated into
 - on the one hand, the text that fits `strong("Lorem ... eius-")`
 - on the other hand, the overflow `strong("mod ... quaerat.")`

If you find weird style artifacts near container boundaries, it is probably a case of faulty bisection heuristics, and deserves to be [reported](#).

II.4 Threading

The threading process interactively invokes both the tiling and the bisection algorithms, establishing the following dialogue:

1. the tiling algorithm yields an available container
2. the bisection algorithm finds the maximum text that fits inside
3. the now full container becomes an obstacle and the tiling is updated
4. start over from step 1.

The order in which the boxes are filled always follows the priority of

- container order,
- top → bottom,
- left → right.

In other words, `MEANDER` will not guess columns, you must always specify columns explicitly.

The exact boundaries of containers may be altered in the process for better spacing.

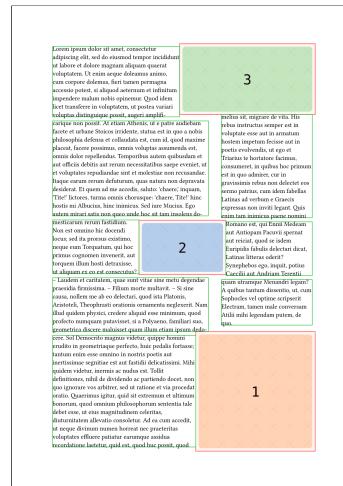


Figure 2: Debug view of the final output via `opt.debug.post-thread()`

Every piece of content produced by `#meander.reflow` is placed, and therefore does not affect layout outside of `#meander.reflow`. See [Section V.4](#) for solutions.

Part III

Contouring

I made earlier two seemingly contradictory claims:

1. `MEANDER` supports wrapping around images of arbitrary shape,
2. `MEANDER` only supports rectangular obstacles.

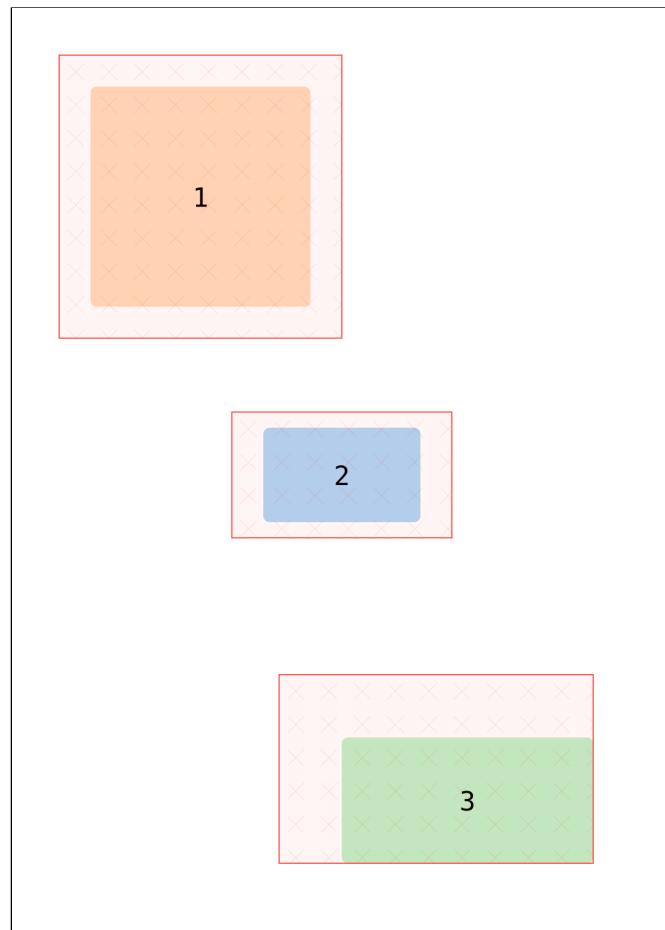
This is not a mistake. The reality is that these statements are only incompatible if we assume that 1 image = 1 obstacle. We call “contouring functions” the utilities that allow splitting one image into multiple obstacles to approximate an arbitrary shape.

All contouring utilities live in the `contour` module.

III.1 Margins

The simplest form of contouring is adjusting the margins. The default is a uniform `5pt` gap, but you can adjust it for each obstacle and each direction.

```
#meander.reflow({
  import meander: *
  opt.debug.pre-thread()
  placed(
    top + left,
    boundary:
      contour.margin(1cm),
    my-img-1,
  )
  placed(
    center + horizon,
    boundary:
      contour.margin(
        5mm,
        x: 1cm,
      ),
    my-img-2,
  )
  placed(
    bottom + right,
    boundary:
      contour.margin(
        top: 2cm,
        left: 2cm,
      ),
    my-img-3,
  )
})
```



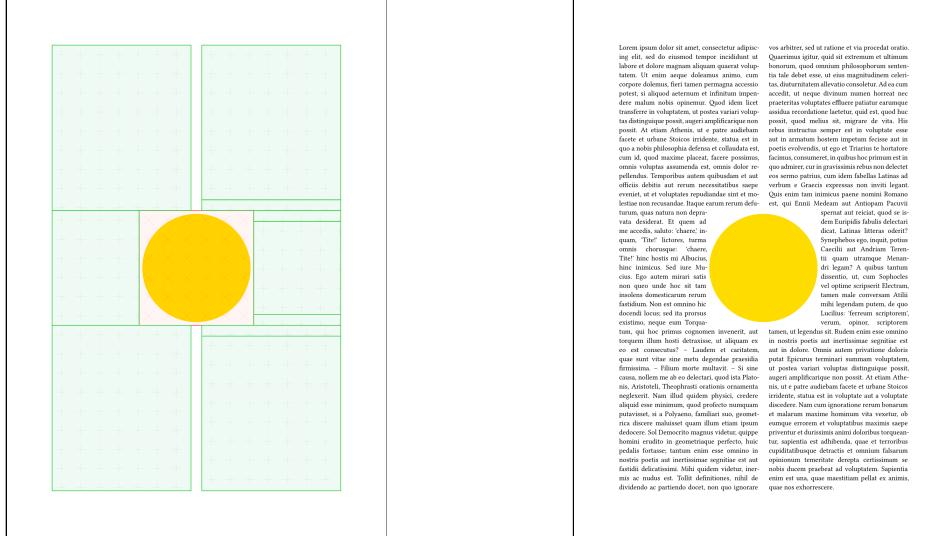
III.2 Boundaries as equations

For more complex shapes, one method offered is to describe as equations the desired shape. Consider the following starting point: a simple double-column page with a cutout in the middle for an image.

```
#meander.reflow({
    import meander: *
    placed(center + horizon)[
        #circle(radius: 3cm, fill: yellow)
    ]

    container(width: 50% - 3mm, margin: 6mm)
    container()

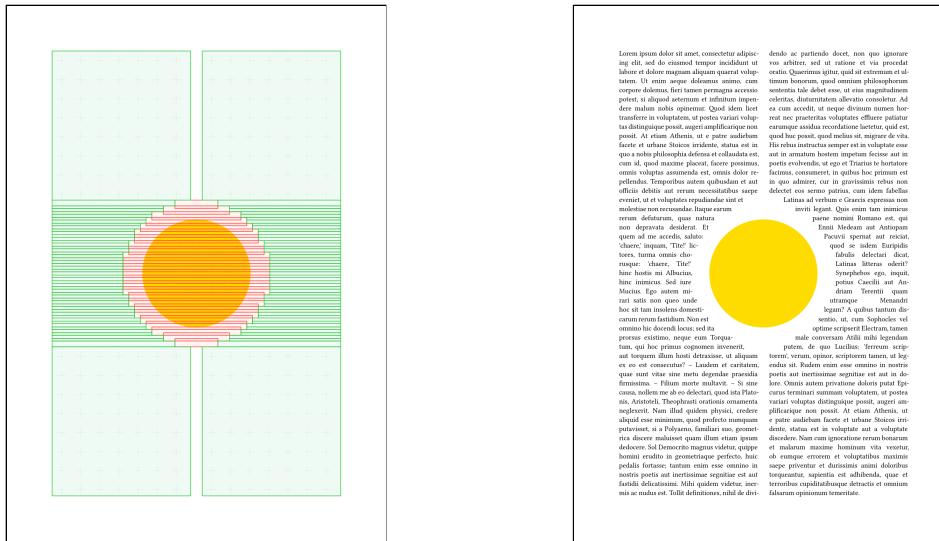
    content[
        #set par(justify: true)
        #lorem(590)
    ]
})
```



MEANDER sees all obstacles as rectangular, so the circle leaves a big ugly **square hole** in the page. Fortunately the desired circular shape is easy to describe in equations, and we can do so using the function `#contour.grid`, which takes as input a 2D formula normalized to $[0, 1] \times [0, 1]$, i.e. a function from $[0, 1] \times [0, 1]$ to `bool`.

```
#meander.reflow({
    import meander: *
    placed(
        center + horizon,
        boundary:
            // Override the default margin
            contour.margin(1cm) +
            // Then redraw the shape as a grid
            contour.grid(
                // 25 vertical and horizontal subdivisions.
                // Just pick a number that looks good.
                // A good rule of thumb is to start with obstacles
                // about as high as one line of text.
                div: 25,
                // Equation for a circle of center (0.5, 0.5) and radius 0.5
                (x, y) => calc.pow(2 * x - 1, 2) + calc.pow(2 * y - 1, 2) <= 1
            ),
            // Underlying object
            circle(radius: 3cm, fill: yellow),
    )
    // ...
})
```

This results in the new subdivisions of containers below.



This enables in theory drawing arbitrary paragraph shapes. In practice not all shapes are convenient to express in this way, so the next sections propose other methods.

Watch out for the density of obstacles. Too many obstacles too close together can impact performance.

III.3 Boundaries as layers

If your shape is not convenient to express through a grid function, but has some horizontal or vertical regularity, here are some other suggestions. As before, they are all normalized between 0 and 1.

Recall that when using all of these you have access to the various settings of `opt.debug` (Section II.1) that can help understand how a given contour segments the obstacle.

III.3.1 Horizontal rectangles

`#contour.horiz` and `#contour.width` produce horizontal layers of varying width. `#contour.horiz` works on a (left, right) basis (the parameterizing function should return the two extremities of the obstacle), while `#contour.width` works on an (anchor, width) basis.



```
#meander.reflow({
  import meander: *
  opt.debug.post-thread()
  placed(right + bottom,
    boundary:
      // The right aligned edge makes
      // this easy to specify using
      // `horiz`
      contour.horiz(
        div: 20,
        // (left, right)
        y => (1 - y, 1),
      ) +
      // Add a post-segmentation margin
      contour.margin(5mm)
    )[...]
    // ...
  )
})
```

The interpretation of `(flush)` for `#contour.width` is as follows:

- if `(flush): left`, the anchor point will be the left of the obstacle;
- if `(flush): center`, the anchor point will be the middle of the obstacle;
- if `(flush): right`, the anchor point will be the right of the obstacle.

```
#meander.reflow({
    import meander: *
    opt.debug.post-thread()
    placed(center + bottom,
        boundary:
            // This time the vertical symmetry
            // makes `width` a good match.
            contour.width(
                div: 20,
                flush: center,
                // Centered in 0.5, of width y
                y => (0.5, y),
            ) +
            contour.margin(5mm)
        )[...]
    // ...
})
```

Latin text from Plato's *Timaeus* describing the construction of the Platonic solids. It discusses the division of the circle into six angles, the creation of triangles, and the assembly of these into various polyhedra.

III.3.2 Vertical rectangles

`#contour.vert` and `#contour.height` produce vertical layers of varying height.



```
#meander.reflow({
    import meander: *
    opt.debug.post-thread()
    placed(top,
        boundary:
            contour.vert(
                div: 25,
                x => if x <= 0.5 {
                    (0, 2 * (0.5 - x))
                } else {
                    (0, 2 * (x - 0.5))
                },
            ) +
            contour.margin(5mm)
        )[...]
    // ...
})
```

The interpretation of `(flush)` for `#contour.height` is as follows:

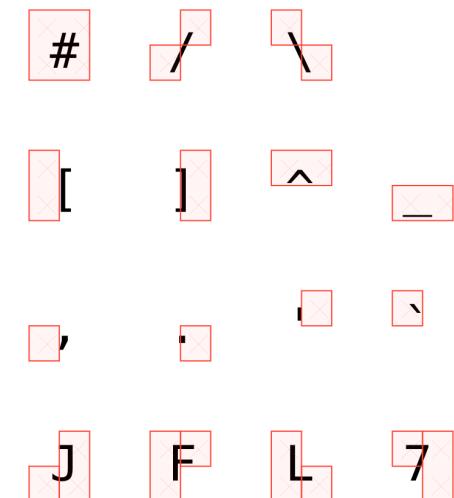
- if `(flush): top`, the anchor point will be the top of the obstacle;
 - if `(flush): horizon`, the anchor point will be the middle of the obstacle;
 - if `(flush): bottom`, the anchor point will be the bottom of the obstacle.

```
#meander.reflow({  
  import meander: *  
  opt.debug.post-thread()  
  placed(left + horizon,  
    boundary:  
      contour.height(  
        div: 20,  
        flush: horizon,  
        x => (0.5, 1 - x),  
      ) +  
      contour.margin(5mm)  
    )[...]  
  // ...  
})
```

III.4 Autocontouring

The contouring function `#contour.ascii-art` takes as input a string or raw code and uses it to draw the shape of the image. The characters that can occur are:

```
#meander.reflow({
    import meander: *
    opt.debug.pre-thread()
    placed(top + left,
        boundary: contour.margin(6mm) +
        contour.ascii-art(
            `#
# / \
[ ] ^ _
, . '
J F L 7
`)
)#[#image]
})
```



If you have [ImageMagick](#) and [Python 3](#) installed, you may use the auxiliary tool `autocontour` to produce a first draft. This small Python script will read an image, pixelate it, apply a customizable threshold function, and produce a `*.contour` file that can be given as input to `#contour.ascii-art`.

```
# Install the script
$ pip install autocontour

# Run on `image.png` down to 15 by 10 pixels, with an 80% threshold.
$ autocontour image.png 15x10 80%

# Then use your text editor of choice to tweak `image.png.contour`
# if it is not perfect.
```

```
#meander.reflow({
    import meander: *
    placed(top + left,
        // Import statically generated boundary.
        boundary: contour.ascii-art(read("image.png.contour")),
        image("image.png"),
    )
    // ...
})
```

You can read more about autocontour on the dedicated [README.md](#)

autocontour is still very experimental.

The output of autocontour is unlikely to be perfect, and it is not meant to be. The format is simple on purpose so that it can be tweaked by hand afterwards.

III.5 More to come

If you find that the shape of your image is not convenient to express through any of those means, you're free to submit suggestions as a [feature request](#).

Part IV

Styling

`MEANDER` respects most styling options through a dedicated content segmentation algorithm, as briefly explained in [Section II](#). Bold, italic, underlined, stroked, highlighted, colored, etc. text is preserved through threading, and easily so because those styling options do not affect layout much.

There are however styling parameters that have a consequence on layout, and some of them require special handling. Some of these restrictions may be relaxed or entirely lifted by future updates.

IV.1 Paragraph justification

In order to properly justify text across boxes, `MEANDER` needs to have contextual access to `#par.justify`, which is only updated via a `#set` rule.

As such **do not** use `#par(justify: true)[...]`.

Instead prefer `#[#set par(justify: true); ...]`, or put the `#set` rule outside of the invocation of `#meander.reflow` altogether.

The diagram illustrates two examples of Meander code. On the left, a red box labeled "Wrong" contains code where `#par(justify: true)` is used directly within a `#meander.reflow` block. On the right, a green box labeled "Correct" shows the same code structure but with the `#set` rule moved outside the `#meander.reflow` block. Both examples include a large block of French text in a black box, which is justified correctly in the "Correct" example but appears unaligned in the "Wrong" example.

```
#meander.reflow({
  // ...
  content[
    #par(justify: true)[
      #lorem(600)
    ]
  ]
})
```

```
#meander.reflow({
  // ...
  content[
    #set par(justify: true)
    #lorem(600)
  ]
})
```

Correct

```
#set par(justify: true)
#meander.reflow({
  // ...
  content[
    #lorem(600)
  ]
})
```

Lorsque nous étudions un concept physique, il est de nos jours courant d'abord de faire une analyse philosophique de ce concept. Cela consiste à essayer de voir si le concept physique en question possède des propriétés qui sont compatibles avec les résultats expérimentaux et théoriques de la physique. Si tel n'est pas le cas, alors on peut dire que le concept physique est incompatible avec la physique. Cependant, il est important de noter que l'incompatibilité entre un concept physique et la physique ne signifie pas nécessairement qu'il est incorrect ou qu'il est en contradiction avec la physique. Il peut simplement être le cas que le concept physique n'a pas été assez bien défini ou n'a pas été assez bien appliqué pour être compatible avec la physique. Par exemple, si on essaie de définir la gravité comme étant une force qui attire deux corps ensemble, alors cela peut être considéré comme une définition correcte de la gravité, mais elle peut également être considérée comme une définition incorrecte si on essaie de l'appliquer à des situations où elle ne fonctionne pas correctement. Par conséquent, il est important de faire une analyse philosophique de tout concept physique pour voir si il est compatible avec la physique ou non.

IV.2 Font size and leading

The font size indirectly affects layout because it determines the spacing between lines. When a linebreak occurs between containers, `MEANDER` needs to manually insert the appropriate spacing there. Since the spacing is affected by font size, make sure to update the font size outside of the `#meander.reflow`. invocation if you want the correct line spacing. Alternatively, `(size)` can be passed as a parameter of `#content` and it will be interpreted as the text size.

Analogously, if you wish to change the spacing between lines, use either a `#set par(leading: 1em)` outside of `#meander.reflow`, or pass `(leading): 1em` as a parameter to `#content`.

Wrong

```
#meander.reflow({  
  // ...  
  content[  
    #set text(size: 30pt)  
    #lorem(80)  
  ]  
})
```

Item ipsum dolor sit domus
consecutus adipiscens clit, sed do-
mod tempor incendiunt ut labore et
dolore magna aliquam quaerat
voluptatem. Ut enim aqueo dole-
mus animo, cum corpore dole-
mus fieri tamen permanga accessio
potest, si aliquod aeternum et infi-
nitum est, non potest esse tempore
mum. Quod idem licet transference in
voluptatem, ut postea variari volu-
tates distinguere possit, augeri am-
plificarique non possit. At etiam
Athenis, ut e patre audiebant facet
et urbane Stoicos irridente, statua
est in quo a nobis philosophia de-
fense et.

Correct

```
#set text(size: 30pt)
#meander.reflow({
    // ...
    content[
        #lorem(80)
    ]
})
```

Lorem ipsum dolor sit amet, consectetur adipicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate, sed quia non numquam eius modi tempora incertus est.

Correct

```
#meander.reflow({
  // ...
  content(size: 30pt)[
    #lorem(80)
  ]
})
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim auctor doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinetur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguenda possit, augeri amplificare non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos.

IV.3 Hyphenation and language

Hyphenation can only be fetched contextually, and highly influences how text is split between boxes. Language indirectly influences layout because it determines hyphenation rules. To control the hyphenation and language, use the same approach as for the text size: either `#set` them outside of `#meander.reflow`, or pass them as parameters to `content`.

Wrong

```
#meander.reflow({
  // ...
  content[
    #set text(hyphenate: true)
    #lorem(70)
  ]
})
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim auctor doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinetur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguenda possit, augeri amplificare non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos.

Correct

```
#set text(hyphenate: true)
#meander.reflow({
  // ...
  content[
    #lorem(70)
  ]
})
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim auctor doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinetur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguenda possit, augeri amplificare non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos.

Correct

```
#meander.reflow({
  // ...
  content(hyphenate: true)[
    #lorem(70)
  ]
})
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim auctor doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinetur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguenda possit, augeri amplificare non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos.

IV.4 Styling containers

#container accepts a `<style>` dictionary that may contain the following keys:

- `(text-fill)`: the color of the text in this container,
- `(align)`: the left/center/right alignment of content,
- and more to come.

These options have in common that they do not affect layout so they can be applied post-threading to the entire box. Future updates may lift this restriction.

```
#meander.reflow({
    import meander: *
    container(width: 25%,
              style: (align: right, text-fill: blue))
    container(width: 75%,
              style: (align: center))
    container(
      style: (text-fill: red))
    content[#lorem(590)]
})
```



Part V

Multi-page setups

V.1 Pagebreak

`MEANDER` can deal with text that spans multiple pages, you just need to place one or more `#pagebreak` appropriately. Note that `#pagebreak` only affects the obstacles and containers, while `#content` blocks ignore them entirely.

The layout below spans two pages:

- obstacles and containers before the `#pagebreak` go to the first page,
- obstacles and containers after the `#pagebreak` go to the second page,
- `#content` is page-agnostic and will naturally overflow to the second page when all containers from the first page are full.

```
#meander.reflow({
    import meander: *

    placed(top + left, my-img-1)
    placed(bottom + right, my-img-2)
    container()

    pagebreak()

    placed(top + right, my-img-3)
    placed(bottom + left, my-img-4)
    container(width: 45%)
    container(align: right, width: 45%)

    content[#lorem(1000)]
})
```

V.2 Colbreak

Analogously, `#colbreak` breaks to the next container. Note that `#pagebreak` is a *container* separator while `#colbreak` is a *content* separator. The next container may be on the next page, so the right way to create an entirely new page for both containers and content is a `#pagebreak and a #colbreak...` or you could just end the `#meander.reflow` and start a new one.

```
#meander.reflow({
  import meander: *

  container(width: 50%, style: (text-fill: red))
  container(style: (text-fill: blue))
  content[#lorem(100)]
  colbreak()
  content[#lorem(500)]

  pagebreak()
  colbreak()

  container(style: (text-fill: green))
  container(style: (text-fill: orange))
  content[#lorem(400)]
  colbreak()
  content[#lorem(400)]
  colbreak() // Note: the colbreak applies only after the overflow is handled.

  pagebreak()

  container(align: center, dy: 25%, width: 50%, style: (text-fill: fuchsia))
  container(width: 50% - 3mm, margin: 6mm, style: (text-fill: teal))
  container(style: (text-fill: purple))
  content[#lorem(400])

})
```

Leem ipsum dolor sit amet, consetetur
adipiscing elit, sed ut etiam tempore
intelligam. Sed ut etiam tempore
voluptatum. Ut enim diametum
animi, cum corporis dolores, fiet tamen
permagno auctor potest, si aliquod asternit
et infinitum impendere natus subit
espnerit. Quid idem licet transfire in
voluptate, ut postea variari voluptas
distinguitur posat, augeri amplificante
non posat. At enim ethica, ut et patre adoevum
et filio amorem, et auctoritate, et voluntate est
qua o nobis philosophia defensio et collaudata
est, cum id, quae miscere placet, faciat
possunt, omnes voluptas acciderit, et
enim deo servare debet. Tempora autem
stultorum sepe exierit, ut et voluptates repudandae

*... ipsum deo sit aet.
adspicere illa, sed de riasse
insidias et laicos et dovere
quærent velutum. Ut enim
animis, cum corpore doveremus,
permagna accessio potest, si
et infinita insperata inoblitio.
Quod illam flet transire in
postos vocari voluntates distingue
augen amplificare non possemus.
Athens, e tate aufadem
Stoico irudente,
status est in quo a
nella phantasie
defensa et collaudata
est, cum sibi, quod
maxime placet,*

V.3 Colfil

Contrary to `#colbreak` which breaks to the next container, `#colfill` fills the current container, *then* breaks to the next container. There is sometimes a subtle difference

between these behaviors, as demonstrated by the examples below. Choose whichever is appropriate based on your use-case.

```
#meander.reflow({
  import meander: *
  container(width: 50%,
    style: (text-fill: red))
  container(
    style: (text-fill: blue))
  content[#lorem(100)]
  colbreak()
  content[#lorem(500)]
})
```

```
#meander.reflow({
  import meander: *
  container(width: 50%,
    style: (text-fill: red))
  container(
    style: (text-fill: blue))
  content[#lorem(100)]
  colfill()
  content[#lorem(300)]
})
```

Loren ipsum dicit et ait, conseruare apicem ali, sed si excedat tempus, non debet tamquam obsoletum, sed utrumque obsoletum, utrumque obsoletum. Ut enim accepit delectus donum, non debet obsoletum esse, sed utrumque obsoletum, si aliquod amorem et infatuacionem habet, non debet obsoletum esse, sed utrumque obsoletum. Hoc transire in cogitatione, ut potius viciam spiritualium distinctionis post, apud angelorum et spirituum, et in contemplatione dei et rerum creationis, etiam in aliis, non debet obsoletum esse, sed utrumque obsoletum. Quia id est in case i.e. apud mundum placet. Sicut potius, quod in aliis, non debet obsoletum esse, sed utrumque obsoletum. Tempus enim pectoribus et impetuibus.

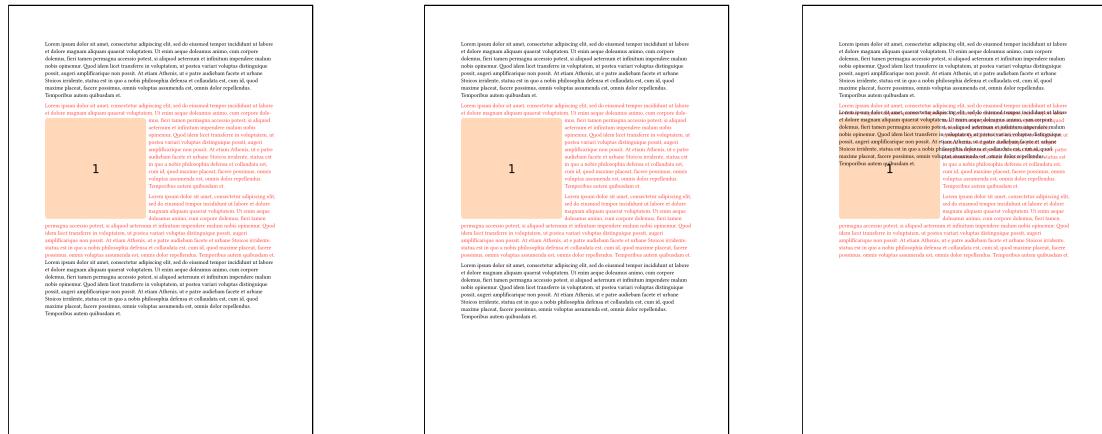
Recall that filled containers count as obstacles for future containers, so there is a difference between dropping containers and filling them with nothing.

V.4 Placement

Placement options control how a `#meander.reflow` invocation is visible by and sees other content. By default, `MEANDER` will automatically insert invisible boxes of the correct height to emulate the text's placement. If this does not behave exactly as you want, this section details the alternatives available.

Placement options are pre-layout, meaning they come in the shape of `opt.placement`.__ before any containers or obstacles.

If the space computed by `MEANDER` is wrong, you can disable it entirely by adding `opt.placement.phantom()` (see below: right), or adjust the margins such as with `opt.placement.spacing(below: 0.65em)` (see below: left).



Manually adjusted spaces
above and below to correct
for paragraph breaks.

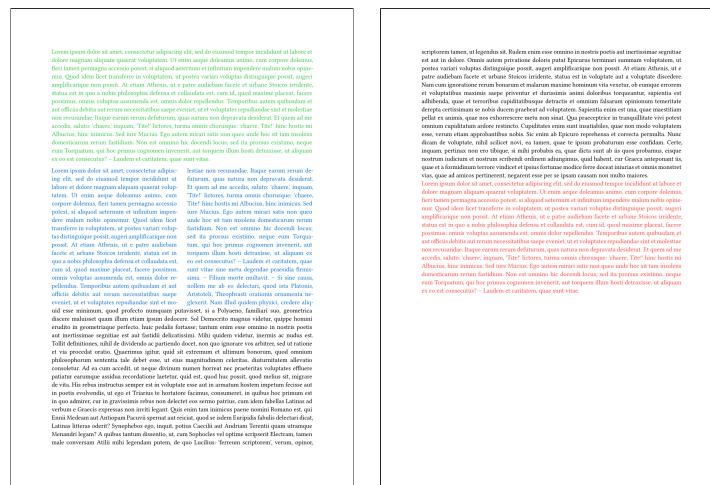
Default heuristic, with arti-
ficial spacing.

No artificial spacing.

V.5 Overflow

By default, if the content provided overflows the available containers, it will spill over after the `MEANDER` invocation. This default behavior means that you don't have to worry about losing text if the containers are too small, and should work for most use-cases.

See below: the text in green / red is respectively before / after the `MEANDER` invocation. The actual layout is only the two blue columns, but the text in black overflows and is thus placed afterwards normally.



```
#text(fill: green)[#lorem(200)]
#meander.reflow({
  import meander: *
  opt.spacing(below: 0.65em)
  container(width: 48%, height: 50%, style: (text-fill: blue))
  container(width: 48%, height: 50%, align: right, style: (text-fill: blue))
  content[#lorem(700)]
})
#text(fill: red)[#lorem(200)]
```

Nevertheless if you need more control over what happens with the overflow, the following options are available.

V.5.1 No overflow

With `opt.overflow.alert()`, a warning message will be added to the document if there is any overflow. The overflow itself will disappear, so the layout is guaranteed to take no more than the allocated space. As for `opt.overflow.ignore()`, it will silently drop any content that doesn't fit.

`opt.overflow.alert()`

```
#meander.reflow({
  import meander: *
  container()
  content[#lorem(1000)]
  opt.overflow.alert()
})
```



`opt.overflow.ignore()`

```
#meander.reflow({
  import meander: *
  container()
  content[#lorem(1000)]
  opt.overflow.ignore()
})
```



V.5.2 Predefined layouts

With `opt.overflow.pagebreak()`, any content that overflows is placed on the next page. This can be particularly interesting if you are also using `opt.placement.phantom()` because that one will not behave well with the default overflow settings.

```
#meander.reflow({  
  import meander: *  
  container(  
    height: 70%, width: 48%,  
    style: (text-fill: blue),  
  )  
  container(  
    align: right,  
    height: 70%, width: 48%,  
    style: (text-fill: blue),  
  )  
  content[#lorem(1000)]  
  opt.overflow.pagebreak()  
})  
#text(fill: red)[#lorem(100)]
```

Laura genna diolto è stato, comunque, un grande successo. Il film ha fatto più di 100 milioni di lire al botteghino italiano. Un vero e proprio clamore mediatico. E' stato anche un grande successo per i principali attori del cast, soprattutto per la regista. Quel che fuori scandalo è, infatti, che il film ha riscosso un buon successo anche all'estero, soprattutto negli Stati Uniti, dove è stato proiettato in molti festival. Siamo convinte, dunque, che la critica non sia stata del tutto giusta nei confronti della nostra pellicola. Non abbiamo mai voluto trasmettere messaggi di odio o di rancore nei confronti di nessuno. So bene che non siamo perfette, ma abbiamo cercato di fare una storia che possa essere accettabile per tutti. La nostra intenzione era quella di creare un racconto che potesse essere compreso da tutti, senza alcuna discriminazione di età, genere o cultura. Non abbiamo voluto creare un film che fosse esclusivo di un gruppo specifico di persone. Siamo state molto attente alle reazioni dei nostri amici e dei nostri familiari, e ce ne siamo assicurate che non avessero alcuna obiezione. Ma non crediamo che ci sia stata una vera e propria censura. La nostra intenzione era di far sentire la nostra voce, quella di Puerto, attraverso un film che potesse essere accolto positivamente da tutti.

Blue text is part of the base layout. The overflow is in black on the next page even though the layout does not occupy the entire page.

Secondly, `opt.overflow.repeat(count: 1)` will duplicate the count last pages (default: 1) of the layout until all the content fits.

```
#meander.reflow({
  import meander: *
  container(width: 70%)
  container()
  content[#lorem(2000)]
}

opt.overflow.repeat()
```

On 15 August 1945, the Japanese government accepted the Potsdam Declaration and agreed to unconditional surrender. The following day, the Emperor issued a radio address to the Japanese people, in which he accepted the terms of the declaration. This was followed by a speech by the Prime Minister, Naoto Kaneko, in which he announced that the war had ended. On 2 September, the Japanese government signed the Instrument of Surrender at the Japanese Embassy in Tokyo. The ceremony was witnessed by representatives from the United States, the United Kingdom, Australia, Canada, New Zealand, and the Soviet Union. The Japanese government also agreed to the formation of a provisional government under the leadership of General Douglas MacArthur, who would be responsible for the administration of Japan until a permanent government could be established.

V.5.3 Custom layouts

Before resorting to one of these solutions, check if there isn't a better way to do whatever you're trying to achieve. If it really is the only solution, consider [reaching out](#) to see if there is a way to make your desired layout better supported and available to other people.

If your desired output does not fit in the above predefined behaviors, you can fall back to storing it to a state or writing a custom overflow handler. Any function (`overflow`) \rightarrow `content` passed to `opt.overflow.custom(_ => {})` can serve as handler,

including another invocation of `#meander.reflow`. This function will be given as input an object of type `overflow`, which is concretely a dictionary with fields:

- `(styled)` is `content` with all styling options applied and is generally what you should use,
- `(structured)` is an array of `elem`, suitable for placing in another `#meander.reflow` invocation,
- `(raw)` uses an internal representation that you can iterate over, but that is not guaranteed to be stable. Use as last resort only.

Similarly if you pass to `opt.overflow.state(label)` a `#state` or a `str`, it will receive any content that overflows in the same 3 formats, and you can use `state.get()` on it afterwards.

For example here is a handler that adds a header and some styling options to the text that overflows:

```
#meander.reflow({
    import meander: *
    container(height: 50%)
    content[#lorem(400)]

    opt.overflow.custom(tt => [
        #set text(fill: red)
        #text(size: 25pt)[
            *The following content overflows:*
        ]
        _#{tt.styled}_
    ])
})
```

Latin text placeholder content:
 Lorum ipsum dolor sit amet. Consectetuer adipiscing elit. Sed de minimis tempore incidentia in libet
 et dolore magna aliquip ex ea commodo consequat. Ut enim imperdiet dolorum animis, sum corporis
 dolores, fieri iansen perennias accessus potest. Ut aliqd aeternum et infinitum impeditore malum
 nobis opinemur. Quod idem licet transire in voluptatem, ut protes variari voluptas distinguere
 possit, angst amplificare non possit. At etiam Albusius, ut e patre audiebas faciet et urbani
 Socratis, quod est invenimus, quod est invenimus, quod est invenimus, quod est invenimus, quod est invenimus.
 Temporibus autem quibusdam et sic officiis debitis aut rerum necessitatibus supereveniet, et ei
 voluptas, et dolores,
 natura non depravata desiderat. Et quod ad me accedit, subiecte clausae inquit. Tunc letters,
 turma omnia cherisque: "cherere, Tunc hinc hostis mi Albusius, hinc innicetus. Sed iure Micius. Igo
 animi, sed iure Micius. Igo animi, sed iure Micius. Igo animi, sed iure Micius. Igo animi, sed iure
 omnino hic decendi locutus, sed ipsa proxima existimatio, neque cum Tropotamus, qui hoc prima
 cognoscere invenirent, aut torque illam hosti detraherem, ut aliquam ex eo est consequitur". - Laudem
 et castigemus, et laudemus, et castigemus, et laudemus, et castigemus, et laudemus, et castigemus, et laudemus,
 sine causa, nolle me ab eo detinere, quod ita Platonus, Aristoteles, Theophrasti orationis ornamenta
 negleguntur. Nam illud quidem physici, credere aliquid esse ministrum, quod profecto muniqueret
 patet. Non enim, ut dicitur, quod est invenimus, quod est invenimus, quod est invenimus, quod est
 deducere. Sol Democritus magnus volerat, quippe homini credidit in geometria perfidis, hinc
 pedalis fortasse; tantum enim esse omnino in nostris portis aut intermissione regnante et aut familiari
 dissensu, ut quod est invenimus, quod est invenimus, quod est invenimus, quod est invenimus, quod est
 partendo doceat, non quo ignorare vos arbitriis, sed ut ratione et via procedat ostendat. Quærimus
 igitur, quid si extremum et ultimum honestum, quod omnium philosophorum sententia tale debet
 esse, ut eius magnitudinem certeferat, distumulatim allevato consideratur. Ad ea cum accedit, ut neque

The following content overflows: Avantus nomen
 horret nec praevertitur voluntas et offensiva patitur. carmine usitatae voluntatis voluntate, quod est
 quod est invenimus, quod est invenimus, quod est invenimus, quod est invenimus, quod est in
 armatum honestum impetrat festus aut in poeta exhortandis, ut ego et Tirostris te horitate facimus,
 consumetur, ut quibus hoc primum aut in quo adiuvet, cari in gratissimis rebus non delectet nos sermo
 patruus, cum

And here is one that stores to a state to be retrieved later:

```
#let overflow = state("overflow")
#meander.reflow({
  import meander: *
  container(height: 50%)
  content[#lorem(400)]
}

opt.overflow.state(overflow)
})

#set text(fill: red)
#text(size: 25pt) [
  *The following content overflows:*
]
_#{context overflow.get().styled}_
```

Use in moderation. Chaining multiple of these together can make your layout diverge.

See also an answer I gave to [issue #1](#) which demonstrates how passing a `#meander.reflow` layout as overflow handler can achieve layouts not otherwise supported. Use this only as a last-resort solution.

Part VI

Inter-element interaction

MEANDER allows attaching tags to elements. These tags can then be used to:

- control visibility of obstacles to other elements,
 - apply post-processing style parameters,
 - position an element relative to a previous one,
 - measuring the width or height of a placed element.

More features are planned, including

- additional styling options,
 - default parameters controlled by tags.

Open a [feature request](#) if you have an idea of a behavior based on tags that should be supported.

You can put one or more tags on any obstacle or container by adding a parameter (`tags`) that contains a `label` or an array of `label`. For example:

- `placed(..., tags: <A>)`
 - `container(..., tags: (, <C>))`

VI.1 Locally invisible obstacles

By passing one or more tags to the parameter `(invisible)` of `container(..)`, you can make it unaffected by the obstacles in question.

```
#meander.reflow({
  import meander: *
  placed(
    top + center,
    my-img-1,
    tags: <x>,
  )
  container(width: 50% - 3mm)
  container(
    align: right,
    width: 50% - 3mm,
    invisible: <x>,
  )
  content[#lorem(600)]
})
```

This is already doable globally by setting (boundary) to `#contour.phantom`. The innovation of (invisible) is that this can be done on a per-container basis.

VI.2 Callbacks and queries

The module `#query` contains functions that allow referencing properties of other elements, as well as other properties that may be dynamically updated in the process of computing the layout. They are used in conjunction with a callback invocation that determines the point of evaluation. See [Section VIII.4](#) for a list of properties that can be queried, and `#callback` for other technical details.

A `#callback` takes a list of named parameters and a function, evaluates the parameters at the call site while the layout is ongoing, and then call the function with the appropriate environment. Below is a possible usage:

```
#meander.reflow({
  import meander: *
  placed(
    left + horizon,
    my-img-3,
    tags: <a>,
  )
  callback(
    align: query.position(<a>, at: center),
    width: query.width(<a>, transform: 150%),
    height: query.height(<a>, transform: 150%),
    env => {
      container(
        align: env.align,
        width: env.width, height: env.height,
        tags: <b>,
      )
    },
  )
  callback(
    pos: query.position(<b>, at: bottom + left),
    env => {
      placed(
        env.pos, anchor: top + left, dx: 5mm,
        my-img-2,
      )
    },
  )
)
content[#lorem(100)]
```

3

2

3
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Quisque volupat. Ut enim sequitur magna aliquam. Sed ut labore dolore, fieri tamquam accinio poset, ad aliquid dolorem et infinitum impendere malum nolis optineat. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguere possit, asperg amplificare non posse. At enim Athenae, ut e patre audiebam facete et urbane Stoicos inservire, sicut est in quibus philosophi defensio et collaudatio est, cum id, quod maxime placet, facere possimus, omnia voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

In this example, after giving an absolute position to one image, we create a container with position and dimensions relative to the image, and place another image immediately after the container ends.

VI.3 A nontrivial example

Here is an interesting application of these features. The `#placed` obstacles all receive a tag `<x>`, and the second container has `(invisible): <x>`. Therefore the `#placed` elements count as obstacles to the first container but not the second. The first container is immediately filled with empty content and counts as an obstacle to the second container. The queries reduce the amount of lengths we have to compute by hand.

```
#meander.reflow({
  import meander: *
  let placed-below(
    tag, img, tags: (),
  ) = {
    callback(
      // fetch position of previous element.
      pos: query.position(tag, at: bottom + left),
      env => {
        placed(
          env.pos,
          img, tags: tags,
          // correct for margins
          dx: 5pt, dy: -5pt,
        )
      }
    )
  }
  // Obstacles
  placed(left, my-img-1, tags: (<x>, <a1>))
  placed-below(<a1>, my-img-2, tags: (<x>, <a2>))
  placed-below(<a2>, my-img-3, tags: (<x>, <a3>))
  placed-below(<a3>, my-img-4, tags: (<x>, <a4>))
  placed-below(<a4>, my-img-5, tags: <x>)

  // Occupies the complement of
  // the obstacles, but has
  // no content.
  container(margin: 6pt)
  colfill()

  // The actual content occupies
  // the complement of the
  // complement of the obstacles.
  container(invisible: <x>)
  content[
    #set par(justify: true)
    #lorem(225)
  ]
})
```

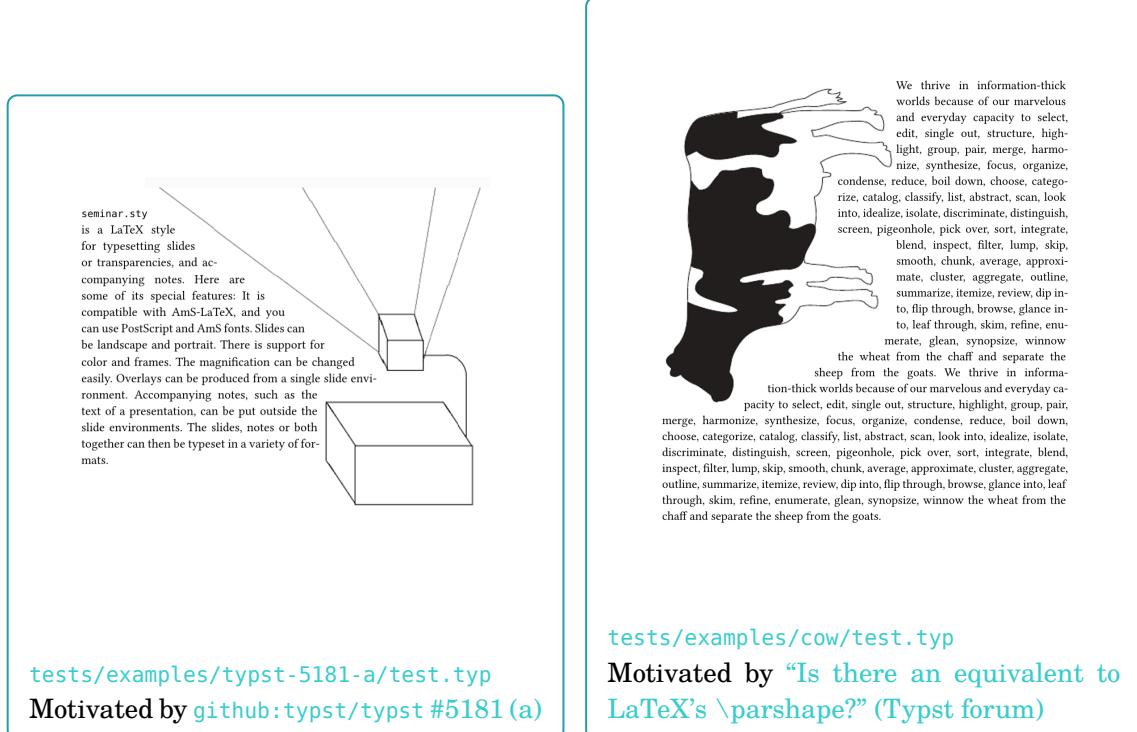
1 Quod idem licet transferre
in voluptatem, ut postea variari voluptas dis-
tinguique possit, augeri amplificarique non
possit. At etiam Athenis, ut e patre audiebam
facete et urbani Stoicos irridente, statua est
in quo a nobis philosophia defensa et collau-
data est, cum id, quod maxime placeat, facere
possimus, omnis voluptas as-
sumenda est, omnis dolor re-
pellendus. Temporibus autem
quibusdam et a officiis debitibus
aut rerum necessitatibus saepe
eveniet, ut et voluptates repu-
diandae sint et molestiae non recusandae. Itaque
earum rerum defuturum, quas natura non deprava
desiderat. Et quem ad me accedit, saluto:
'chaere' inquam, 'Tite!' 2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
5510
5511
5512
5513
5514
5515
5516
5517
5518
5519
5520
5521
5522
5523
5524
5525
5526
5527
5528
5529
55210
55211
55212
55213
55214
55215
55216
55217
55218
55219
55220
55221
55222
55223
55224
55225
55226
55227
55228
55229
552210
552211
552212
552213
552214
552215
552216
552217
552218
552219
552220
552221
552222
552223
552224
552225
552226
552227
552228
552229
5522210
5522211
5522212
5522213
5522214
5522215
5522216
5522217
5522218
5522219
5522220
5522221
5522222
5522223
5522224
5522225
5522226
5522227
5522228
5522229
55222210
55222211
55222212
55222213
55222214
55222215
55222216
55222217
55222218
55222219
55222220
55222221
55222222
55222223
55222224
55222225
55222226
55222227
55222228
55222229
552222210
552222211
552222212
552222213
552222214
552222215
552222216
552222217
552222218
552222219
552222220
552222221
552222222
552222223
552222224
552222225
552222226
552222227
552222228
552222229
5522222210
5522222211
5522222212
5522222213
5522222214
5522222215
5522222216
5522222217
5522222218
5522222219
5522222220
5522222221
5522222222
5522222223
5522222224
5522222225
5522222226
5522222227
5522222228
5522222229
55222222210
55222222211
55222222212
55222222213
55222222214
55222222215
55222222216
55222222217
55222222218
55222222219
55222222220
55222222221
55222222222
55222222223
55222222224
55222222225
55222222226
55222222227
55222222228
55222222229
552222222210
552222222211
552222222212
552222222213
552222222214
552222222215
552222222216
552222222217
552222222218
552222222219
552222222220
552222222221
552222222222
552222222223
552222222224
552222222225
552222222226
552222222227
552222222228
552222222229
5522222222210
5522222222211
5522222222212
5522222222213
5522222222214
5522222222215
5522222222216
5522222222217
5522222222218
5522222222219
5522222222220
5522222222221
5522222222222
5522222222223
5522222222224
5522222222225
5522222222226
5522222222227
5522222222228
5522222222229
55222222222210
55222222222211
55222222222212
55222222222213
55222222222214
55222222222215
55222222222216
55222222222217
55222222222218
55222222222219
55222222222220
55222222222221
55222222222222
55222222222223
55222222222224
55222222222225
55222222222226
55222222222227
55222222222228
55222222222229
552222222222210
552222222222211
552222222222212
552222222222213
552222222222214
552222222222215
552222222222216
552222222222217
552222222222218
552222222222219
552222222222220
552222222222221
552222222222222
552222222222223
552222222222224
552222222222225
552222222222226
552222222222227
552222222222228
552222222222229
5522222222222210
5522222222222211
5522222222222212
5522222222222213
5522222222222214
5522222222222215
5522222222222216
5522222222222217
5522222222222218
5522222222222219
5522222222222220
5522222222222221
5522222222222222
5522222222222223
5522222222222224
5522222222222225
5522222222222226
5522222222222227
5522222222222228
5522222222222229
55222222222222210
55222222222222211
55222222222222212
55222222222222213
55222222222222214
55222222222222215
55222222222222216
55222222222222217
55222222222222218
55222222222222219
55222222222222220
55222222222222221
55222222222222222
55222222222222223
55222222222222224
55222222222222225
55222222222222226
55222222222222227
55222222222222228
55222222222222229
552222222222222210
552222222222222211
552222222222222212
552222222222222213
552222222222222214
552222222222222215
552222222222222216
552222222222222217
552222222222222218
552222222222222219
552222222222222220
552222222222222221
552222222222222222
552222222222222223
552222222222222224
552222222222222225
552222222222222226
552222222222222227
552222222222222228
552222222222222229
5522222222222222210
5522222222222222211
5522222222222222212
5522222222222222213
5522222222222222214
5522222222222222215
5522222222222222216
5522222222222222217
5522222222222222218
5522222222222222219
5522222222222222220
5522222222222222221
5522222222222222222
5522222222222222223
5522222222222222224
5522222222222222225
5522222222222222226
5522222222222222227
5522222222222222228
5522222222222222229
55222222222222222210
55222222222222222211
55222222222222222212
55222222222222222213
55222222222222222214
55222222222222222215
55222222222222222216
55222222222222222217
55222222222222222218
55222222222222222219
55222222222222222220
55222222222222222221
55222222222222222222
55222222222222222223
55222222222222222224
55222222222222222225
55222222222222222226
55222222222222222227
55222222222222222228
55222222222222222229
552222222222222222210
552222222222222222211
552222222222222222212
552222222222222222213
552222222222222222214
552222222222222222215
552222222222222222216
552222222222222222217
552222222222222222218
552222222222222222219
552222222222222222220
552222222222222222221
552222222222222222222
552222222222222222223
552222222222222222224
552222222222222222225
552222222222222222226
552222222222222222227
552222222222222222228
552222222222222222229
5522222222222222222210
5522222222222222222211
5522222222222222222212
5522222222222222222213
5522222222222222222214
5522222222222222222215
5522222222222222222216
5522222222222222222217
5522222222222222222218
5522222222222222222219
5522222222222222222220
5522222222222222222221
5522222222222222222222
5522222222222222222223
5522222222222222222224
5522222222222222222225
5522222222222222222226
5522222222222222222227
5522222222222222222228
5522222222222222222229
55222222222222222222210
55222222222222222222211
55222222222222222222212
55222222222222222222213
55222222222222222222214
55222222222222222222215
55222222222222222222216
55222222222222222222217
55222222222222222222218
55222222222222222222219
55222222222222222222220
55222222222222222222221
55222222222222222222222
55222222222222222222223
55222222222222222222224
55222222222222222222225
55222222222222222222226
55222222222222222222227
55222222222222222222228
55222222222222222222229
552222222222222222222210
552222222222222222222211
552222222222222222222212
552222222222222222222213
552222222222222222222214
552222222222222222222215
552222222222222222222216
552222222222222222222217
552222222222222222222218
552222222222222222222219
552222222222222222222220
552222222222222222222221
552222222222222222222222
552222222222222222222223
552222222222222222222224
552222222222222222222225
552222222222222222222226
552222222222222222222227
552222222222222222222228
552222222222222222222229
5522222222222222222222210
5522222222222222222222211
5522222222222222222222212
5522222222222222222222213
5522222222222222222222214
5522222222222222222222215
5522222222222222222222216
5522222222222222222222217
5522222222222222222222218
5522222222222222222222219
5522222222222222222222220
5522222222222222222222221
5522222222222222222222222
5522222222222222222222223
5522222222222222222222224
5522222222222222222222225
5522222222222222222222226
5522222222222222222222227
5522222222222222222222228
5522222222222222222222229
55222222222222222222222210
55222222222222222222222211
55222222222222222222222212
55222222222222222222222213
55222222222222222222222214
55222222222222222222222215
55222222222222222222222216
55222222222222222222222217
55222222222222222222222218
55222222222222222222222219
55222222222222222222222220
55222222222222222222222221
55222222222222222222222222
55222222222222222222222223

Part VII

Showcase

A selection of nontrivial examples of what is feasible, inspired mostly by requests on issue #5181. You can find the source code for these on the [repository](#).

VII.1 Side illustrations



VII.2 Paragraph packing

Talmudifier Test Page

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis vehicula ligula at est bibendum, in eleifend erat dictum. Etiam sit amet tellus id ex ullamcorper faucibus. Suspendisse sed elit vel neque convallis iaculis id in urna. Sed incidunt varius ipsum at scelerisque. Phasellus lacus lectus, sodales sit amet orci in, rutrum malesuada diam. Cras pulvinar elit sit amet laoreet ligula, in euismod massa. Phasellus euismod dolor sed pretium elementum. Nulla sagittis, elit eget semper portitor, erat nunc commodo turpis, et bibendum ex lorem laoreet ipsum. Morbi auctor dignissim velit eget consequat. R. Seth: Blah blah blah blah. As it is written: "Accens lacinia nisi diam, vel pulvinar metus aliquet ut Sed non lorem quis duimates tunc etiam at nec tempus id quam. Vestibulum id malesuada. Ut suspendisse hendrerit eros sit amet tempor pulvinar. Duis velit mauris, ut tincidunt sed pharetra eu libero. Aenean lobortis tincidunt nisi. L2 Praesent metus lacus, tristique sed portio, tempus id quam. Vestibulum id malesuada. Ut suspendisse hendrerit sapien ligula, in euismod magna. Duis nec lacus arcu. Modis vel fermentum leo. Phasellusque hendrerit sagittis vulputate. Fusce laoreet malesuada odio, sit amet fringilla lectus ultrices porta. Aliquam feugiat finibus turpis id malesuada. Ut suspendisse hendrerit eros sit amet tempor pulvinar. Duis velit mauris, ut tincidunt sed pharetra eu libero. Aenean lobortis tincidunt nisi. L2 Praesent metus lacus, tristique sed portio, tempus id quam. Vestibulum id malesuada. Ut suspendisse hendrerit sapien ligula, in euismod magna. Duis nec lacus arcu. Modis vel fermentum leo. Phasellusque hendrerit sagittis vulputate. Fusce laoreet malesuada odio, sit amet fringilla lectus ultrices porta. Aliquam feugiat finibus turpis id malesuada. Ut suspendisse hendrerit eros sit amet tempor pulvinar. Duis velit mauris, facilisis ut tincidunt sed pharetra eu libero. Aenean lobortis tincidunt nisi. L2 Praesent metus lacus, tristique sed portio, tempus id quam. L3 I use these red Hebrew letters in my own WIP project along with various other font changes... You might find this functionality useful. R. Allen: I strongly encourage you and others to use this. It will have to catch the eyes upon pages of what might as well be lorem ipsum to figure out why we're dishing on each other so much. As it is written: "Sed ut eros id arcu tincidunt accumsan. Vestibulum vitae nisi blandit, commodo odio vitae, dictum nunc. Suspendisse pharetra lorem vitae ex tincidunt ornare. Maecenas efficitur tristique libero, eget commodo urna. Phasellusque libero sem, interdum ut nibis interdum, consequat magnis." IpsumР L2 Aliquam facilisis vel turpis eu semper. Donec eget purus lectus. -> Check out how nice that little hand looks. Nice. -> Fusce purus diam. Etiam venenatis nisi nec tempus fringilla. Vivamus vehicula nunc sed libero scelerisque viverra a quis libero. Integer ac urna ut lectus faucibus mattis ac id nunc. Morbi fermentum magna dui, at rhoncus nibh porttitor quis. Donec dui ante, semper non quam at, accumsan volutpat leo. Maecenas magna risus, finibus sit amet felis ut, vulputate euismod nunc.

[tests/examples/talmudifier/test.typ](#)

From [github:subalterngames/talmudifier](https://github.com/subalterngames/talmudifier)
Motivated by [github:typst/typst #5181 \(c\)](https://github.com/typst/typst/pull/5181)

 It's possible to control the length of lines in a much more general way, if simple changes to \leftskip and \rightskip aren't flexible enough

for your purposes. For example, a semi-circle hole has been cut out

of the present paragraph, in order to make room for a circular illustration that contains some of Galileo's immortal words about circles: all of the line breaks in this paragraph and in the circular quotation were found by TeX's line-breaking algorithm. You can specify an essentially arbitrary paragraph shape by saying \parshape=(number), where the (number) is a positive integer n , followed by $2n$ (dimen) specifications. In general, \parshape=i₁ i₂ i₃ i₄ ... i_n specifies a paragraph whose first n lines will have lengths i_1, i_2, \dots, i_n , respectively, and they will be indented from the left margin by the respective amounts i_1, i_2, \dots, i_n . If the paragraph has fewer than n lines, the additional specifications will be ignored; if it has more than n lines, the specifications for line n will be repeated ad infinitum. You can cancel the effect of a previously specified \parshape by saying \parshape=0.

The area of a circle is a mean proportional between any two regular and similar polygons of which one circumscribes it and the other is inscribed within it. In addition, the area of the circle is less than that of any regular inscribed polygon and greater than that of any circumscribed polygon. And further, of these circumscribed polygons, the one that has the greater number of sides has a smaller area than the one that has a lesser number, and on the other hand, the one that has the greater number of sides is the larger [Galileo, 1638]

[tests/examples/area-of-a-circle/test.typ](#)

Inspired by TeXBook, asked on [Stack Exchange](#)

VII.3 Drop caps

I

DOWN THE RABBIT-HOLE



Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, "and what is the use of a book," thought Alice.

So she was considering in her own mind (as well as she could, for the hot day made her feel very sleepy and stupid), whether the pleasure of making a daisy-chain would be worth the trouble of getting up and picking the daisies, when suddenly a White Rabbit with pink eyes ran close by her.

There was nothing so *very* remarkable in that; nor did Alice think it so *very* much out of the way to hear the Rabbit say to itself, "Oh dear! Oh dear! I shall be late!" (when she thought it over afterwards, it occurred to her that she ought to have wondered at this, but at the time it all seemed quite natural); but when the Rabbit actually *took a watch out of its waistcoat-pocket*, and looked at it, and then hurried

[tests/examples/drop-caps/test.typ](#)

Courtesy of [@wade-cheng](#) on #18

Part VIII

Public API

These are the user-facing functions of `MEANDER`.

VIII.1 Elements

All constructs that are valid within a `#meander.reflow({ ... })` block. Note that they all produce an object that is a singleton dictionary, so that the `#meander.reflow({ ... })` invocation is automatically passed as input the concatenation of all these elements. For clarity we use the more descriptive type `elem`, instead of the internal representation (`dictionary`,)

<code>#callback</code>	<code>#container</code>	<code>#placed</code>
<code>#colbreak</code>	<code>#content</code>	
<code>#colfill</code>	<code>#pagebreak</code>	

↑ Since 0.4.0

`#callback(..{env}, {fun})`

Callback to generate elements that depend on prior layout information.

Argument –

`..{env}`

`dictionary`

Elements from the query module assigned to names. See [Section VIII.4](#) for a list of usable values.

Argument –

`{fun}`

`function(dictionary => list(elem))`

A function that takes as input a dictionary of environment values and outputs elements for the layout.

The function may only generate layout elements, not flow elements. That is, it can call `#placed`, `#container`, `#pagebreak`, but **not** `#content`, `#colbreak`, `#colfill`.

`#colbreak → flowing`

Continue content to next container. Has the same internal fields as the output of `#content` so that we don't have to check for key `in elem` all the time.

`#colfill → flowing`

Continue content to next container after filling the current container with whitespace.

```
#container(
  {align}: top + left,
  {dx}: 0% + 0pt,
  {dy}: 0% + 0pt,
  {width}: 100%,
  {height}: 100%,
  {style}: (),
  {margin}: 5mm,
  {invisible}: (),
  {tags}: ()
) → elem
```

Core function to create a container.

— Argument —

{align}: top + left

Location on the page or position of a previously placed container.

— Argument —

{dx}: 0% + 0pt

relative

Horizontal displacement.

— Argument —

{dy}: 0% + 0pt

relative

Vertical displacement.

— Argument —

{width}: 100%

relative

Width of the container.

— Argument —

{height}: 100%

relative

Height of the container.

— Argument —

{style}: ()

dictionary

Styling options for the content that ends up inside this container. If you don't find the option you want here, check if it might be in the `{style}` parameter of `#content` instead.

- align: flush text left/center/right
- text-fill: color of text

— Argument —

{margin}: 5mm

length | dictionary

Margin around the eventually filled container so that text from other paragraphs doesn't come too close. Follows the same convention as `#pad` if given a dictionary (x, y, left, right, rest, etc.)

Argument —

`(invisible): ()`

`label` | `array(label)`

One or more labels that will not affect this element's positioning.

Argument —

`(tags): ()`

`label` | `array(label)`

Optional set of tags so that future element can refer to this one and others with the same tag.

`#content({size}: auto, {lang}: auto, {hyphenate}: auto, {leading}: auto)[data]`

→ `flowing`

Core function to add flowing content.

Argument —

`(data)`

`content`

Inner content.

Argument —

`(size): auto`

`length`

Applies `#set text(size: ...)`.

Argument —

`(lang): auto`

`str`

Applies `#set text(lang: ...)`.

Argument —

`(hyphenate): auto`

`bool`

Applies `#set text(hyphenate: ...)`.

Argument —

`(leading): auto`

`length`

Applies `#set par(leading: ...)`.

`#pagebreak` → `elem`

Continue layout to next page.

```
#placed(
  {align},
  {dx}: 0% + 0pt,
  {dy}: 0% + 0pt,
  {boundary}: (auto,),
  {display}: true,
  {tags}: (),
  {anchor}: auto
)[content] → elem
```

Core function to create an obstacle.

— Argument —

{align}

Reference position on the page or relative to a previously placed object.

— Argument —

{dx}: 0% + 0pt

relative

Horizontal displacement.

— Argument —

{dy}: 0% + 0pt

relative

Vertical displacement.

— Argument —

{boundary}: (auto,)

contour

An array of functions to transform the bounding box of the content. By default, a 5pt margin. See [Section III](#) and [Section VIII.3](#) for more information.

— Argument —

{display}: true

bool

Whether the obstacle is shown. Useful for only showing once an obstacle that intersects several invocations. Contrast the following:

- {boundary} set to `#contour.phantom` will display the object without using it as an obstacle,
- {display}: `false` will use the object as an obstacle but not display it.

— Argument —

{content}

content

Inner content.

— Argument —

{tags}: ()

label | array(label)

Optional set of tags so that future element can refer to this one and others with the same tag.

Argument —

`(anchor): auto`

`auto` | `align`

Anchor point to the alignment. If `auto`, the anchor is automatically determined from `(align)`. If an alignment, the corresponding point of the object will be at the specified location.

VIII.2 Layouts

These are the toplevel invocations. They expect a sequence of `elem` as input, and produce content.

`#meander.reflow`

`#meander.reflow((seq)) → content`

Segment the input sequence according to the tiling algorithm, then thread the flowing text through it.

Argument —

`(seq)`

`array(elem)`

See [Section VIII.1](#) for how to format this content.

VIII.3 Contouring

Functions for approximating non-rectangular boundaries. We refer to those collectively as being of type `contour`. They can be concatenated with `+` which will apply contours successively.

`#contour.ascii-art`
`#contour.grid`
`#contour.height`

`#contour.horiz`
`#contour.margin`
`#contour.phantom`

`#contour.vert`
`#contour.width`

`#contour.ascii-art((ascii)) → contour`

Allows drawing the shape of the image as ascii art.

Blocks

- `"#"`: full
- `" "`: empty

Half blocks

- `"["`: left
- `"] "`: right

- "⌞": top
 - "⌞": bottom
- Quarter blocks
- "⌞⌞": top left
 - "⌞⌞": top right
 - "⌞⌞": bottom left
 - "⌞⌞": bottom right

- Anti-quarter blocks
- "⌞⌞": top left
 - "⌞⌞": top right
 - "⌞⌞": bottom left
 - "⌞⌞": bottom right

- Diagonals
- "⌞⌞": positive
 - "⌞⌞": negative

Argument
(ascii) code | str

Draw the shape of the image in ascii art.

#contour.grid({div}: 5, {fun}) → contour

Cuts the image into a rectangular grid then checks for each cell if it should be included. The resulting cells are automatically grouped horizontally.

Argument
(div): 5 int | (x: int, y: int)

Number of subdivisions.

Argument
(fun) function

Returns for each cell whether it satisfies the 2D equations of the image's boundary.

(fraction, fraction) → bool

#contour.height({div}: 5, {flush}: horizon, {fun}) → function

Vertical segmentation as (anchor, height).

Argument
(div): 5 int

Number of subdivisions.

— Argument —

`(flush): horizon` align

Relative vertical alignment of the anchor.

— Argument —

`(fun)` function

For each location, returns the position of the anchor and the height.

`(fraction) →(fraction , fraction)`

#contour.horiz({div}: 5, {fun}) → contour

Horizontal segmentation as (left, right)

— Argument —

`(div): 5` int

Number of subdivisions.

— Argument —

`(fun)` function

For each location, returns the left and right bounds.

`(ratio) →(ratio , ratio)`

#contour.margin(..{args}) → contour

Contouring function that pads the inner image.

— Argument —

`..{args}`

May contain the following parameters, ordered here by decreasing generality and increasing precedence

- `(length): length` for all sides, the only possible positional argument
- `(x), (y): length` for horizontal and vertical margins respectively
- `(top), (bottom), (left), (right): length` for single-sided margins

#contour.phantom → contour

Drops all boundaries. Having as `{boundary}` a `#contour.phantom` will let other content flow over this object.

#contour.vert({div}: 5, {fun}) → contour

Vertical segmentation as (top, bottom)

— Argument —

`(div): 5` int

Number of subdivisions.

<p>— Argument —</p> <p><code>(fun)</code></p> <p>For each location, returns the top and bottom bounds.</p> <p><code>(fraction) →(fraction , fraction)</code></p>	<p style="text-align: right;"><code>function</code></p>
<p><code>#contour.width({div}: 5, {flush}: center, {fun}) → contour</code></p> <p>Horizontal segmentation as (anchor, width).</p>	
<p>— Argument —</p> <p><code>{div}: 5</code></p> <p>Number of subdivisions.</p>	<p style="text-align: right;"><code>int</code></p>
<p>— Argument —</p> <p><code>{flush}: center</code></p> <p>Relative horizontal alignment of the anchor.</p>	<p style="text-align: right;"><code>align</code></p>
<p>— Argument —</p> <p><code>{fun}</code></p> <p>For each location, returns the position of the anchor and the width.</p> <p><code>(fraction) →(fraction , fraction)</code></p>	<p style="text-align: right;"><code>function</code></p>

VIII.4 Queries

Enables interactively fetching properties from previous elements. See how to use them in [Section VI](#).

<code>#query.height</code>	<code>#query.position</code>
<code>#query.parent-size</code>	<code>#query.width</code>

`#query.height({tag}, {transform}: 100%) → query(length)`

Retrieve the height of a previously placed and labeled element. If multiple elements have the same label, the resulting height is the maximum left-to-right span.

<p>— Argument —</p> <p><code>(tag)</code></p> <p>Reference a previous element by its tag.</p>	<p style="text-align: right;"><code>label</code></p>
<p>— Argument —</p> <p><code>{transform}: 100%</code></p> <p>Apply some post-processing transformation to the value.</p>	<p style="text-align: right;"><code>ratio function</code></p>

`#query.parent-size({clip-to-page}: true, {transform}: 100%) → query(size)`

Returns the size of the container.

— Argument —

`(clip-to-page): true`

bool

By default, returns only the available space, but you can turn off this parameter to include the entire page including space that is not usable by MEANDER.

— Argument —

`(transform): 100%`

ratio | function

Apply a transformation to the value.

`#query.position((tag), (at): center) → query(location)`

Retrieve the location of a previously placed and labeled element. If multiple elements have the same label, the position is relative to the union of all of their boxes.

— Argument —

`(tag)`

label

Reference a previous element by its tag.

— Argument —

`(at): center`

align

Anchor point relative to the box in question.

`#query.width((tag), (transform): 100%) → query(length)`

Retrieve the width of a previously placed and labeled element. If multiple elements have the same label, the resulting width is the maximum top-to-bottom span.

— Argument —

`(tag)`

label

Reference a previous element by its tag.

— Argument —

`(transform): 100%`

ratio | function

Apply some post-processing transformation to the value.

VIII.5 Options

Configuring the behavior of `#meander.reflow`.

VIII.5.1 Pre-layout options

These come before all elements.

Debug settings

Visualizing containers and obstacle boundaries.

```
#opt.debug.minimal          #opt.debug.pre-thread
#opt.debug.post-thread      #opt.debug.release
```

#opt.debug.minimal → option

Does not show obstacles or content. Displays forbidden regions in red and allowed regions in green.

#opt.debug.post-thread → option

Shows obstacles and content. Displays forbidden regions in red and allowed regions in green (non-invasive).

#opt.debug.pre-thread → option

Shows obstacles but not content. Displays forbidden regions in red and allowed regions in green.

#opt.debug.release → option

No visible effect.

Placement settings

Controlling the interactions between content inside and outside of the `MEANDER` invocation.

```
#opt.placement.phantom      #opt.placement.spacing
```

#opt.placement.phantom → option

Disables the automatic vertical spacing to mimic the space taken by the `MEANDER` layout.

#opt.placement.spacing({above}: auto, {below}: auto, {both}: auto)

Controls the margins before and after the `MEANDER` layout.

— Argument —

`(above): auto`

auto | length

Margin above the layout

— Argument —

`(below): auto`

auto | length

Margin below the layout

— Argument —

`(both): auto`

auto | length

Affects `(above)` and `(below)` simultaneously.

VIII.5.2 Dynamic options

These modify parameters on the fly.

None yet

VIII.5.3 Post-layout options

These come after all elements.

Overflow settings

What happens to content that doesn't fit.

<code>#opt.overflow.alert</code>	<code>#opt.overflow.ignore</code>	<code>#opt.overflow.repeat</code>
<code>#opt.overflow.custom</code>	<code>#opt.overflow.pagebreak</code>	<code>#opt.overflow.state</code>

`#opt.overflow.alert` → option

Print a warning if there is any overflow.

`#opt.overflow.custom(({fun}))` → option

Arbitrary handler.

— Argument —

`(fun)`

`function`

Should take as input a dictionary with fields `(raw)`, `(styled)`, `(structured)`. Most likely you should use `(styled)`, but if you want to pass the result to another `MEANDER` invocation then `(structured)` would be more appropriate.

`#opt.overflow.ignore` → option

Silently drop any content that overflows.

`#opt.overflow.pagebreak` → option

Insert a pagebreak between the layout and the overflow regardless of the available space.

`#opt.overflow.repeat(({count}): 1)` → option

Loop the last page(s) until all content fits.

— Argument —

`(count): 1`

`int`

Adjust the number of pages copied from the end.

`#opt.overflow.state(({state}))` → option

Store the overflow in the given global state variable.

— Argument —

`(state)`

`state | str`

A state or its label.

VIII.6 Public internals

If `MEANDER` is too high-level for you, you may use the public internals made available as lower-level primitives.

Public internal functions have a lower standard for backwards compatibility. Make sure to pin a specific version.

```
#import "@preview/meander:0.4.0": internals.fill-box
```

This grants you access to the primitive `fill-box`, which is the entry point of the content bisection algorithm. It allows you to take as much content as fits in a specific box. See `#bisect.fill-box` for details.

```
#import "@preview/meander:0.4.0": internals.geometry
```

This grants you access to all the functions in the `geometry` module, which implement interesting 1D and 2D primitives. See [Section IX.3](#) for details.

Part IX

Internal module details

IX.1 Utils

```
#utils.apply-styles           #utils.coerce-to-array
```

```
#utils.apply-styles(({size): auto, (lang): auto, (hyphenate): auto, (leading): auto)[data] → content
```

Applies some of the standard styling options that affect layout and therefore are stored separately in our internal representation.

Argument –	
(data)	content
Text to style.	
Argument –	
(size): auto	length
Applies #set text(size: ...)	
Argument –	
(lang): auto	str
Applies #set text(lang: ...).	
Argument –	
(hyphenate): auto	bool
Applies #set text(hyphenate: ...).	
Argument –	
(leading): auto	length
Applies #set par(leading: ...).	

```
#utils.coerce-to-array((t)) → array(T)
```

Interprets a single element as a singleton.

Argument –	
(t)	T array(T)
Element or array	

IX.2 Types

`#types.query`

`#types.query`

Standalone type of queries

`#opt`

Option marker

- pre: options that come before the layout
- post: options that come after the layout

`#flow`

Flowing content

- content: text
- colbreak: break text to the next container
- colfill: break text to the next container and fill the current one

`#elt`

Layout elements

- placed: obstacles
- container: containers
- pagebreak: break layout to next page

IX.3 Geometry

<code>#geometry.align</code>	<code>#geometry.clamp</code>	<code>#geometry.intersects</code>
<code>#geometry.apply-transform</code>	<code>#geometry.frac-rect</code>	<code>#geometry.resolve</code>
<code>#geometry.between</code>	<code>#geometry.in-region</code>	<code>#geometry.unquery</code>

`#geometry.align(`
`(alignment),`
`(dx): Opt,`
`(dy): Opt,`
`(width): Opt,`
`(height): Opt,`
`(anchor): auto`
`) → (x: relative, y: relative)`

Compute the position of the upper left corner, taking into account the alignment and displacement.

— Argument —

`(alignment)`

`align` | `dictionary`

Absolute alignment. If this is an `alignment`, it will be computed relative to the page. If it is a `(x: length, y: length)`, that will be used as the target position.

Argument —

`(dx): 0pt`

`relative`

Horizontal displacement.

Argument —

`(dy): 0pt`

`relative`

Vertical displacement.

Argument —

`(width): 0pt`

`relative`

Object width.

Argument —

`(height): 0pt`

`relative`

Object height.

Argument —

`(anchor): auto`

`align | auto`

Anchor point.

#geometry.apply-transform({value}, {transform}): 100% → any

Apply a transformation in the form of either a scaling or a function.

Argument —

`{value}`

`any`

Value to transform. Any type as long as it supports multiplication by a scalar.

Argument —

`{transform}: 100%`

`function | ratio`

Scaling to apply, as either a ratio or a function.

#geometry.between({a}, {b}, {c}) → bool

Testing `a <= b <= c`, helps only computing `b` once.

Argument —

`{a}`

`length`

Lower bound.

— Argument —

(b)

length

Tested value.

— Argument —

(c)

length

Upper bound. Asserted to be \geq a.**#geometry.clamp({val}, {min}: none, {max}: none) → block(length)**

Bound a value between `{min}` and `{max}`. No constraints on types as long as they support inequality testing.

— Argument —

{val}

any

Base value.

— Argument —

{min}: none

any | none

Lower bound.

— Argument —

{max}: none

any | none

Upper bound.

#geometry.frac-rect({frac}, {abs}, ..(style)) → block(length)

Helper function to turn a fractional box into an absolute one.

— Argument —

{frac}

block(fraction)

Child dimensions as fractions.

— Argument —

{abs}

block(length)

Parent dimensions as absolute lengths.

— Argument —

..(style)

Currently ignored.

#geometry.in-region({region}, {alignment}) → (x: length, y: length)

Resolves an anchor point relative to a region.

— Argument —

(region)

block

A block (x: length, y: length, width: length, height: length).

— Argument —

(alignment)

align

An alignment within the block.

#geometry.intersects((i1), (i2), (tolerance): Opt)

Tests if two intervals intersect.

— Argument —

(i1)

(length, length)

First interval as a tuple of (low, high) in absolute lengths.

— Argument —

(i2)

(length, length)

Second interval.

— Argument —

(tolerance): Opt

length

Set to nonzero to ignore small intersections.

#geometry.resolve((size), ..(args)) → dictionary

Converts relative and contextual lengths to absolute. The return value will contain each of the arguments once converted, with arguments that begin or end with "x" or start with "w" being interpreted as horizontal, and arguments that begin or end with "y" or start with "h" being interpreted as vertical.

```
1 #context resolve(
2     width: 100pt, height: 200pt),
3     x: 10%, y: 50% + 1pt,
4     width: 50%, height: 5pt,
5 )
```

— Argument —

(size)

size

Size of the container as given by the layout function.

— Argument —

..(args)

dictionary

Arbitrary many length arguments, automatically inferred to be horizontal or vertical.

↓ Until 0.4.0

#geometry.unquery({obj}, {regions}: (:)) → dictionary

Fetch all required answers to geometric queries. See [Section VIII.4](#) for details.
Deprecated in favor of callback-style interactivity

— Argument —

{obj}

dictionary

Every field of this object that has an attribute `(type): query` will be transformed based on previously computed regions.

— Argument —

{regions}: (:)

dictionary(block)

Regions delimited by items already placed on the page.

IX.4 Tiling

```
#tiling.add-self-margin      #tiling.eval-callback-env      #tiling.placement-mode
#tiling.blocks-of-container #tiling.get-page-offset       #tiling.push-elem
#tiling.blocks-of-placed    #tiling.is-ignored          #tiling.separate
#tiling.create-data         #tiling.next-elem
```

#tiling.add-self-margin({elem}) → elem

Applies an element's margin to itself.

— Argument —

{elem}

elem

Inner element.

#tiling.blocks-of-container({data}, {obj}) → blocks

See: `#tiling.next-elem` to explain `{data}`. Computes the effective containers from an input object, as well as the display and debug outputs.

— Argument —

{data}

opaque

Internal state.

— Argument —

{obj}

elem

Container to segment.

↗ context

#tiling.blocks-of-placed({data}, {obj}) → blocks

See: `#tiling.next-elem` to explain `{data}`. This function computes the effective obstacles from an input object, as well as the display and debug outputs.

Argument
`(data)` opaque
 Internal state.

Argument
`(obj)` elem
 Object to measure, pad, and place.

`#tiling.create-data({size}: none, {page-offset}: (x: 0pt, y: 0pt), {elems}: ())`
 \rightarrow opaque

Initializes the initial value of the internal data for the reentering next-elem.

- `none` means no more elements
- `()` means no element right now, but keep trying

Argument
`(size): none` size
 Dimensions of the page

Argument
`(page-offset): (x: 0pt, y: 0pt)` size
 Optional nonzero offset on the top left corner

Argument
`(elems): ()` $(..elem,)$
 Elements to dispense in order

`#tiling.eval-callback-env({env}, {data})`

Evaluates an environment passed to a callback. Most of the computations are done by the values, defined in the query module.

`#tiling.get-page-offset` \rightarrow `(x: length, y: length)`

Gets the position of the current page's anchor. Can be called within a layout to know the true available space. See Issue 4 (<https://github.com/Vanille-N/meander-typ/issues/4>) for what happens when we **don't** have this mechanism.

`#tiling.is-ignored({container}, {obstacle})`

Eliminates non-candidates by determining if the obstacle is ignored by the container.

Argument
`(container)`
 Must have the field `(invisible)`, as containers do.

— Argument —
 (obstacle)

Must have the field `(tags)`, as obstacles do.

`#tiling.next-elem({data}) → (elem, opaque)`

This function is reentering, allowing interactive computation of the layout. Given its internal state `(data)`, `#tiling.next-elem` uses the helper functions `#tiling.blocks-of-placed` and `#tiling.blocks-of-container` to compute the dimensions of the next element, which may be an obstacle or a container.

— Argument —
 (data) opaque

Internal state, stores

- `(size)` the available page dimensions,
- `(elems)` the remaining elements to handle in reverse order (they will be popped),
- `(obstacles)` the running accumulator of previous obstacles;

`#tiling.placement-mode({opts}) → function`

Determines the appropriate layout invocation based on the placement options. See details on `#meander.reflow`.

`#tiling.push-elem({data}, {elem}) → opaque`

Updates the internal state to include the newly created element.

— Argument —
 (data) opaque

Internal state.

— Argument —
 (elem) elem

Element to register.

`#tiling.separate({seq}) → (pages: array(elem), flow: array(elem), opts: dictionary)`

Splits the input sequence into pages of elements (either obstacles or containers), and flowing content.

```

1 #separate({
2   // This is an obstacle
3   placed(top + left, box(width: 50pt, height: 50pt))
4   // This is a container
5   container(height: 50%)
6   // This is flowing content
7   content[#lorem(50)]

```

```
8 })
```

— Argument —

(seq)

array(elem)

A sequence of elements made from #placed, #content, #container, etc.

IX.5 Bisection

#bisect.default-rebuild	#bisect.has-body	#bisect.is-enum-item
#bisect.dispatch	#bisect.has-child	#bisect.is-list-item
#bisect.fill-box	#bisect.has-children	#bisect.split-word
#bisect.fits-inside	#bisect.has-text	#bisect.take-it-or-leave-it

#bisect.default-rebuild((inner-field))[ct] → (dictionnary, function)

Destructure and rebuild content, separating the outer content builder from the rest to allow substituting the inner contents. In practice what we will usually do is recursively split the inner contents and rebuild the left and right halves separately.

Inspired by WRAP-IT's implementation (see: #_rewrap in [github:ntjess/wrap-it](https://github.com/ntjess/wrap-it))

```
1 #let content = box(stroke: red)[Initial]
2 #let (inner, rebuild) = default-rebuild(
3   content, "body",
4 )
5
6 Content: #content \
7 Inner: #inner \
8 Rebuild: #rebuild("foo")

1 #let content = [*_Initial_*]
2 #let (inner, rebuild) = default-rebuild(
3   content, "body",
4 )
5
6 Content: #content \
7 Inner: #inner \
8 Rebuild: #rebuild("foo")

1 #let content = [a:b]
2 #let (inner, rebuild) = default-rebuild(
3   content, "children",
4 )
5
6 Content: #content \
```

```
7 Inner: #inner \
8 Rebuild: #rebuild(([x], [y]))
```

Argument —

(inner-field)

str

What “inner” field to fetch (e.g. “body”, “text”, “children”, etc.)

#bisect.dispatch({fits-inside}, {cfg})[ct] → (content?, content?)

Based on the fields on the content, call the appropriate splitting function. This function is involved in a mutual recursion loop, which is why all other splitting functions take this one as a parameter.

Argument —

(ct)

content

Content to split.

Argument —

(fits-inside)

function

Closure to determine if the content fits (see #bisect.fits-inside).

Argument —

(cfg)

dictionary

Extra configuration options.

#bisect.fill-box({dims}, {size}: none, {cfg}: (:))[ct] → (content, content)

Initialize default configuration options and take as much content as fits in a box of given size. Returns a tuple of the content that fits and the content that overflows separated.

Argument —

(dims)

size

Container size.

Argument —

(ct)

content

Content to split.

Argument —

{size}: none

size

Parent container size.

Argument —

{cfg}: (:)

dictionary

Configuration options.

- `(list-markers)`: `([•], [‣], [-], [•], [‣], [-])` an array of content describing the markers used for list items. If you change the default markers, put the new value in the parameters so that lists are correctly split.
- `(enum-numbering)`: `("1.", "1.", "1.", "1.", "1.", "1.")` an array of numbering patterns for enumerations. If you change the numbering style, put the new value in the parameters so that enums are correctly split.
- `(hyphenate)`: `auto` determines if the text can be hyphenated. Defaults to `text.hyphenate`.
- `(lang)`: `auto` specifies the language of the text. Defaults to `text.lang`.
- `(linebreak)`: `auto` determines the behavior of linebreaks at the end of boxes. Supports the following values:
 - `true` → justified linebreak
 - `false` → non-justified linebreak
 - `none` → no linebreak
 - `auto` → linebreak with the same justification as the current paragraph

 context

`#bisect.fits-inside(dimensions, size: none)[ct] → bool`

Tests if content fits inside a box.

Horizontal fit is not very strictly checked. A single word may be said to fit in a box that is less wide than the word. This is an inherent limitation of `measure(box(...))` and I will try to develop workarounds for future versions.

The closure of this function constitutes the basis of the entire content splitting algorithm: iteratively add content until it no longer fits inside the box, with what “iteratively add content” means being defined by the content structure. Essentially all remaining functions in this file are about defining content that can be split and the correct way to invoke `#bisect.fits-inside` on them.

```

1 #let dims = (width: 100%, height: 50%)
2 #box(width: 7cm, height: 3cm)[#layout(size => context {
3   let words = [#lorem(12)]
4   [#fits-inside(dims, words, size: size)]
5   linebreak()
6   box(..dims, stroke: 0.1pt, words)
7 })]

1 #let dims = (width: 100%, height: 50%)
2 #box(width: 7cm, height: 3cm)[#layout(size => context {
3   let words = [#lorem(15)]
4   [#fits-inside(dims, words, size: size)]
```

```
5   linebreak()
6   box(..dims, stroke: 0.1pt, words)
7 })]
```

— Argument —

(dims)

(width: relative, height: relative)

Maximum container dimensions. Relative lengths are allowed.

— Argument —

(ct)

content

Content to fit in.

— Argument —

(size): none

(width: length, height: length)

Dimensions of the parent container to resolve relative sizes. These must be absolute sizes.

```
#bisect.has-body({split-dispatch}, {fits-inside}, {cfg})[ct] →
(content?, content?)
```

Split content with a "body" main field. There is a special strategy for list.item and enum.item which are handled separately. Elements #strong, #emph, #underline, #stroke, #overline, #highlight, #par, #align, #link are splittable, the rest are treated as non-splittable.

— Argument —

(ct)

content

Content to split.

— Argument —

(split-dispatch)

function

Recursively passed around (see #bisect.dispatch).

— Argument —

(fits-inside)

function

Closure to determine if the content fits (see #bisect.fits-inside).

— Argument —

(cfg)

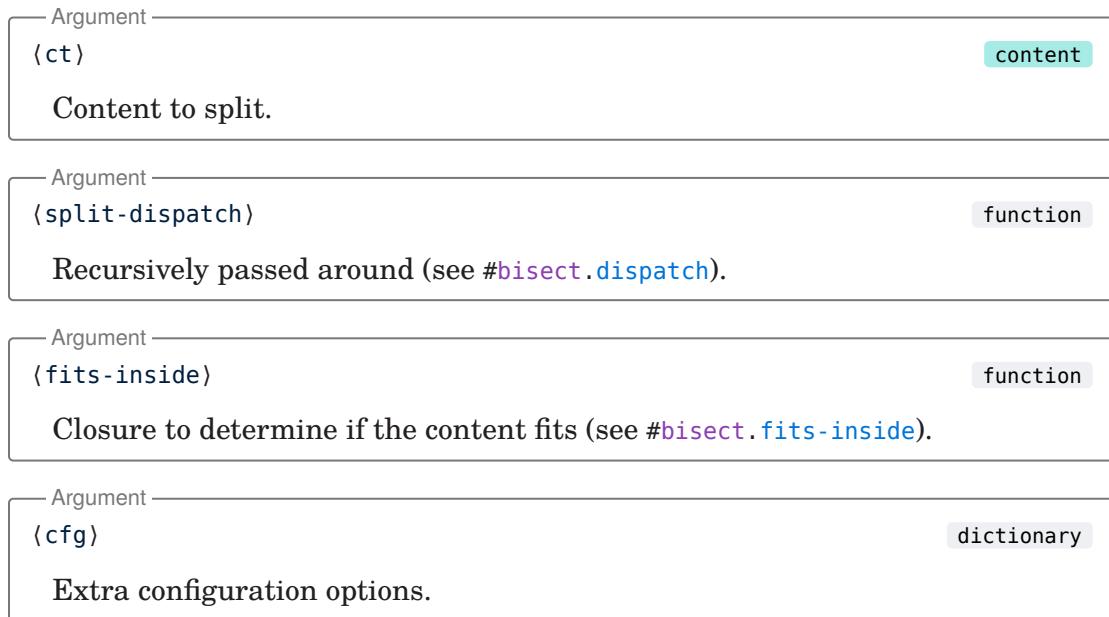
dictionary

Extra configuration options.

```
#bisect.has-child({split-dispatch}, {fits-inside}, {cfg})[ct] →
(content?, content?)
```

Split content with a "child" main field.

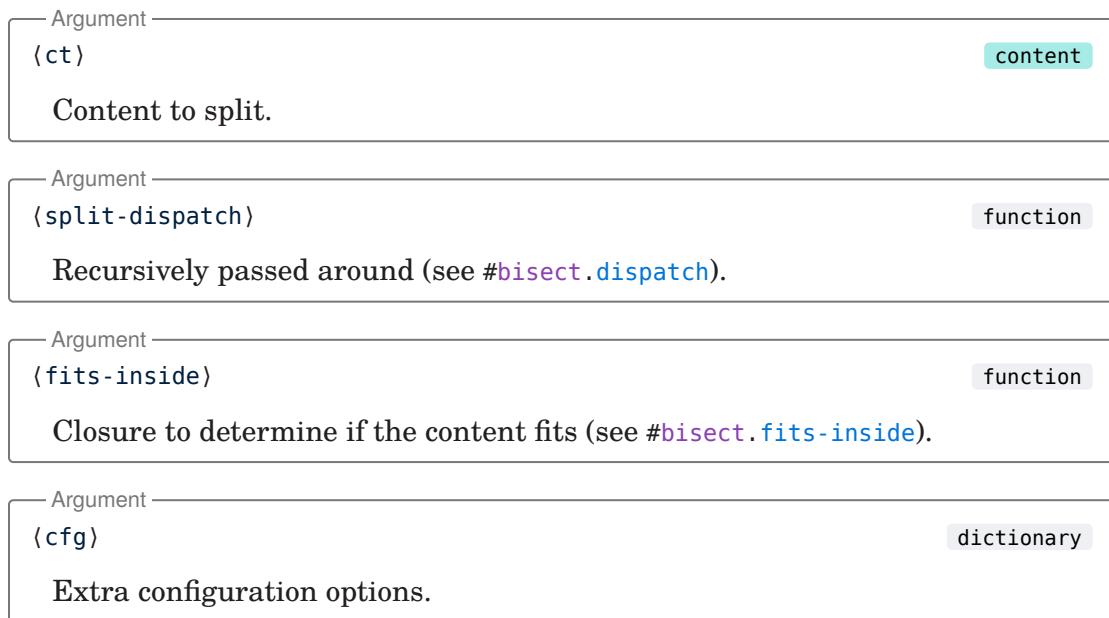
Strategy: recursively split the child.



```
#bisect.has-children(({split-dispatch}, {fits-inside}, {cfg})){ct} →
(content?, content?)
```

Split content with a "children" main field.

Strategy: take all children that fit.



```
#bisect.has-text(({split-dispatch}, {fits-inside}, {cfg})){ct} →
(content?, content?)
```

Split content with a "text" main field.

Strategy: split by " " and take all words that fit. Then if hyphenation is enabled, split by syllables and take all syllables that fit.

End the block with a `#linebreak` that has the justification of the paragraph, or other based on `cfg.linebreak`.

— Argument —
`(ct)` content
 Content to split.

— Argument —
`(split-dispatch)` function
 Recursively passed around (see `#bisect.dispatch`).

— Argument —
`(fits-inside)` function
 Closure to determine if the content fits (see `#bisect.fits-inside`).

— Argument —
`(cfg)` dictionary
 Extra configuration options.

```
#bisect.is-enum-item({split-dispatch}, {fits-inside}, {cfg})[ct] →
(content?, content?)
```

Split an enum.item.

The numbering will reset on the split. I am developing a fix, in the meantime use explicit numbering.

Strategy: recursively split the body, and do some magic to simulate a numbering indent.

— Argument —
`(ct)` content
 Content to split.

— Argument —
`(split-dispatch)` function
 Recursively passed around (see `#bisect.dispatch`).

— Argument —
`(fits-inside)` function
 Closure to determine if the content fits (see `#bisect.fits-inside`).

<p>Argument (cfg)</p> <p>Extra configuration options.</p>	dictionary
<pre>#bisect.is-list-item({split-dispatch}, {fits-inside}, {cfg})[ct] → (content?, content?)</pre> <p>Split a list.item.</p> <p>Strategy: recursively split the body, and do some magic to simulate a bullet point indent.</p>	
<p>Argument (ct)</p> <p>Content to split.</p>	content
<pre>(split-dispatch)</pre> <p>Recursively passed around (see <code>#bisect.dispatch</code>).</p>	
<p>Argument (fits-inside)</p> <p>Closure to determine if the content fits (see <code>#bisect.fits-inside</code>).</p>	function
<p>Argument (ww)</p> <p>Word to split.</p>	str
<pre>(fits-inside)</pre> <p>Closure to determine if the content fits (see <code>#bisect.fits-inside</code>).</p>	
<p>Argument (cfg)</p> <p>Extra configuration options.</p>	dictionary

`#bisect.split-word({ww}, {fits-inside}, {cfg}) → (content?, content?)`

Split one word according to hyphenation patterns, if enabled.

<p>Argument (ww)</p> <p>Word to split.</p>	str
<pre>(fits-inside)</pre> <p>Closure to determine if the content fits (see <code>#bisect.fits-inside</code>).</p>	
<p>Argument (cfg)</p> <p>Extra configuration options.</p>	dictionary

`#bisect.take-it-or-leave-it({fits-inside})[ct] → (content?, content?)`

“Split” opaque content.

— Argument —

(ct)

content

This content cannot be split. If it fits take it, otherwise keep it for later.

— Argument —

(fits-inside)

function

Closure to determine if the content fits (see `#bisect.fits-inside`).

IX.6 Threading

#threading.smart-fill-boxes

^ context

```
#threading.smart-fill-boxes({avoid}: (), {boxes}: (), {size}: none)[body] →
(full: , overflow: overflow)
```

Thread text through a list of boxes in order, allowing the boxes to stretch vertically to accomodate for uneven tiling.

— Argument —

(body)

content

Flowing text.

— Argument —

{avoid}: ()

(..block,)

An array of `block` to avoid.

— Argument —

{boxes}: ()

(..block,)

An array of `block` to fill.

The `(bound)` parameter of `block` is used to know how much the container is allowed to stretch.

— Argument —

{size}: none

size

Dimensions of the container as given by `#layout`.

Part X

About

X.1 Related works

This package takes a lot of basic ideas from [Typst's own builtin layout model](#), mainly lifting the restriction that all containers must be of the same width, but otherwise keeping the container-oriented workflow. There are other tools that implement similar features, often with very different models internally.

In Typst:

- [WRAP-IT](#) has essentially the same output as [MEANDER](#) with only one obstacle and one container. It is noticeably more concise for very simple cases.

In L^AT_EX:

- [wrapfig](#) can achieve similar results as [MEANDER](#) as long as the images are rectangular, with the notable difference that it can even affect content outside of the `\begin{wrapfigure}... \end{wrapfigure}` environment.
- [floatfit](#) and [picins](#) can do a similar job as [wrapfig](#) with slightly different defaults.
- [parshape](#) is more low-level than all of the above, requiring every line length to be specified one at a time. It has the known drawback to attach to the paragraph data that depends on the obstacle, and is therefore very sensitive to layout adjustments.

Others:

- [Adobe InDesign](#) supports threading text and wrapping around images with arbitrary shapes.

X.2 Dependencies

In order to obtain hyphenation patterns, [MEANDER](#) imports [HY-DRO-GEN](#), which is a wrapper around [typst/hyphen](#). This manual is built using [MANTYS](#) and [TIDY](#).

X.3 Acknowledgements

[MEANDER](#) would have taken much more effort to bootstrap had I not had access to [WRAP-IT](#)'s source code to understand the internal representation of content, so thanks to [@ntjess](#).

[MEANDER](#) started out as an idea in the Typst Discord server; thanks to everyone who gave input and encouragements.

Thanks also to the people who use [MEANDER](#) and submit bug reports and feature requests, many regressions would not have been discovered as quickly were it not for their vigilance.