

MEANDER

User guide

v0.3.0

2025-11-26

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.

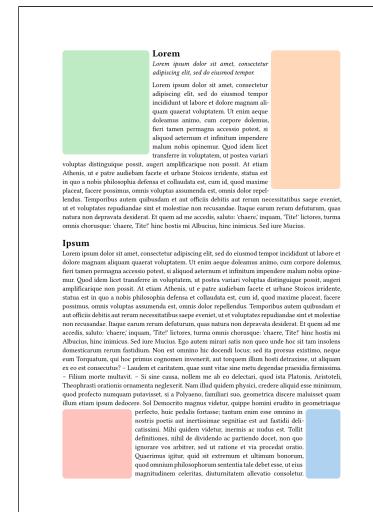
Though very different in its modeling, **MEANDER** can be seen as a Typst alternative to L^AT_EX's `wrapfig` and `parshape`, effectively enabling the same kinds of outputs.

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](#)
- [0.3.0 \(latest\)](#)
- [0.2.5](#)
- [0.2.4](#)
- [0.2.3](#)
- [...](#)



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

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
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	17
Styling	18
IV.1 Paragraph justification	18
IV.2 Font size and leading	19
IV.3 Hyphenation and language	20
IV.4 Styling containers	21
Multi-page setups	22
V.1 Pagebreak	22
V.2 Colbreak	22
V.3 Colfill	23
V.4 Placement	24
V.5 Overflow	25
V.5.1 No overflow	26
V.5.2 Predefined layouts	27
V.5.3 Custom layouts	27
Inter-element interaction	30
VI.1 Locally invisible obstacles	30
VI.2 Position and length queries	31
VI.3 A nontrivial example	32
Showcase	33
Public API	35
VIII.1 Elements	35
VIII.2 Layouts	38
VIII.3 Contouring	39
VIII.4 Queries	42
VIII.5 Options ^(*)	43
VIII.5.1 Pre-layout options	43
VIII.5.2 Dynamic options	44
VIII.5.3 Post-layout options	44
VIII.6 Public internals	45
Internal module details	46
IX.1 Utils	46
IX.2 Types	46
IX.3 Geometry	47
IX.4 Tiling	51
IX.5 Bisection	53
IX.6 Threading	60
About	62
X.1 Related works	62
X.2 Dependencies	62
X.3 Acknowledgements	62

Chapters that are [highlighted^{\(*\)}](#)
have received major updates in
the latest version 0.3.0

Part I

Quick start

Import the latest version of `MEANDER` with:

```
#import "@preview/meander:0.3.0"
```

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

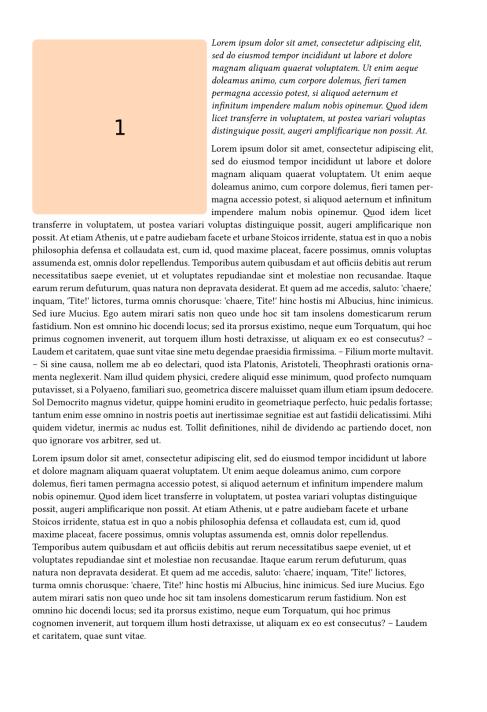
Below is a single page whose layout is fully determined by `MEANDER`. The general pattern of `#placed + #container + #content` is almost universal.

```
#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

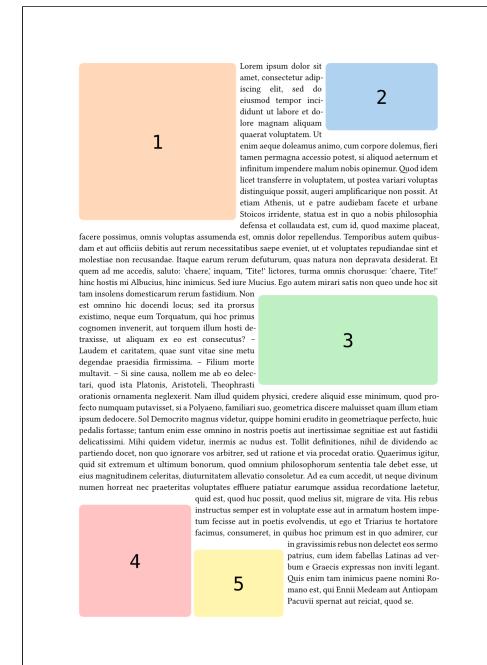
A single `#meander.reflow` invocation can contain multiple `#placed` objects. A possible limitation would be performance if the number of obstacles grows too large, but experiments have shown that up to ~100 obstacles is still workable.

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 factus, consumetur, in quibus hoc primum est in quo admirer, cor in gravissima rebus non detectus eos sermo patrius, cum idem fabellas Latinas ad verbum e Graecis expressas non possit legant. Quis enim invenit, quod non in libro Romano est, qui Enni Medean aut An-

2

tiopam Pacuvii spernit aut recitat, quod se idem Euripidis fabulis detectri dicit, Latinas litteras odit? Synephebos ego, nonne? Ceteraque aut Andram Terentium quan- utrumque Menandri 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	

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.

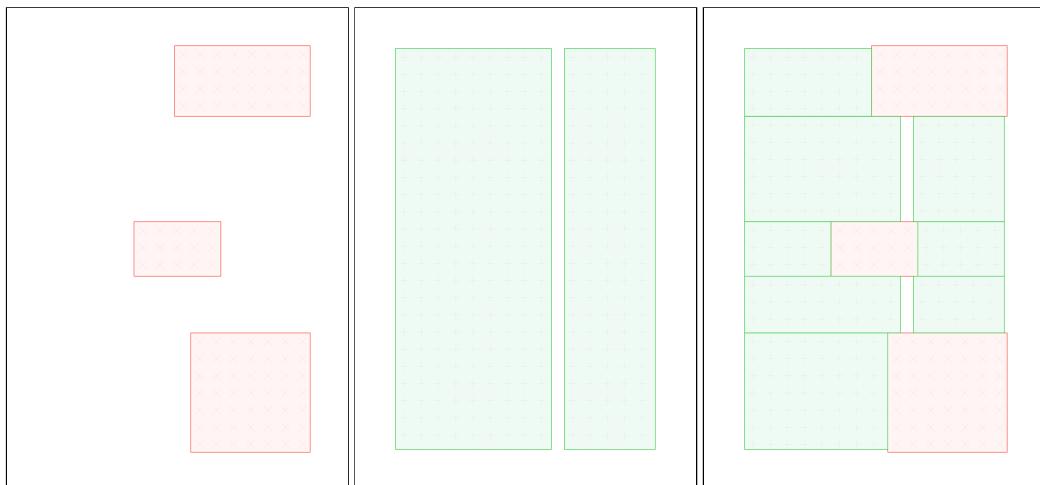


Figure 1: Left to right: the forbidden, allowed, and combined regions.

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.

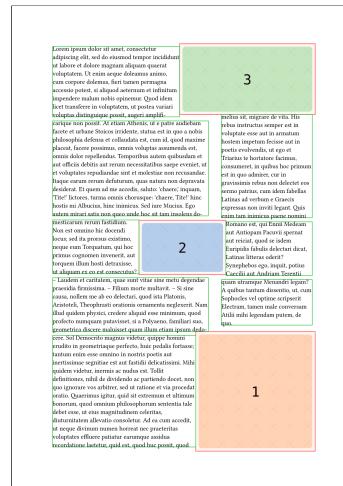


Figure 2: Debug view of the final output via `opt.debug.post-thread()`

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.

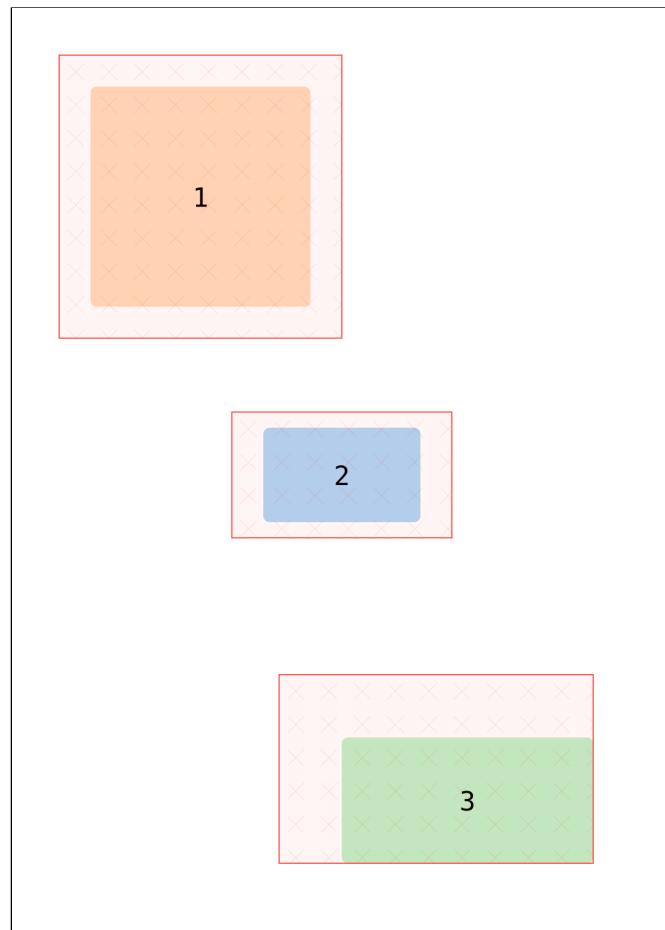
This is not a mistake. The reality is that these statements are only incompatible if we assume that 1 image = 1 obstacle. We call “contouring functions” the utilities that allow splitting one image into multiple obstacles to approximate an arbitrary shape.

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,
  )
})
```



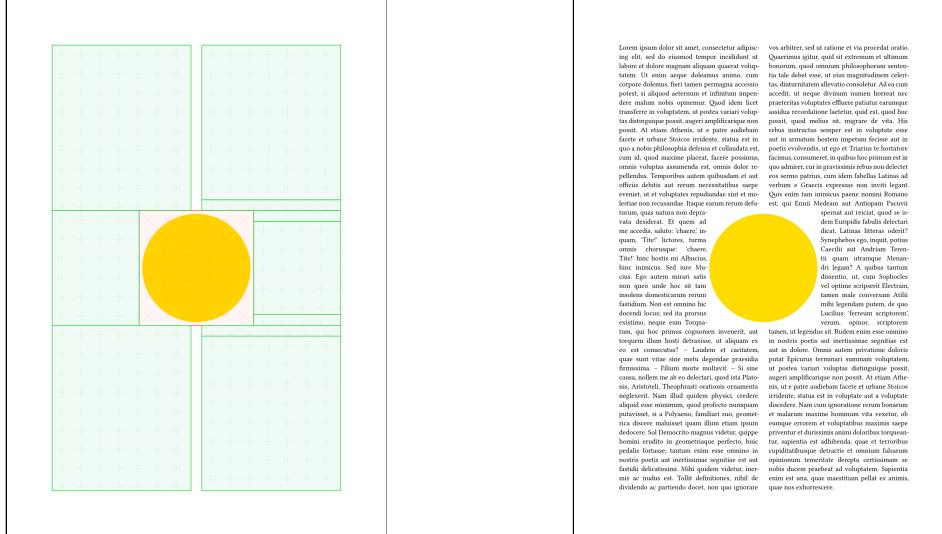
III.2 Boundaries as equations

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()

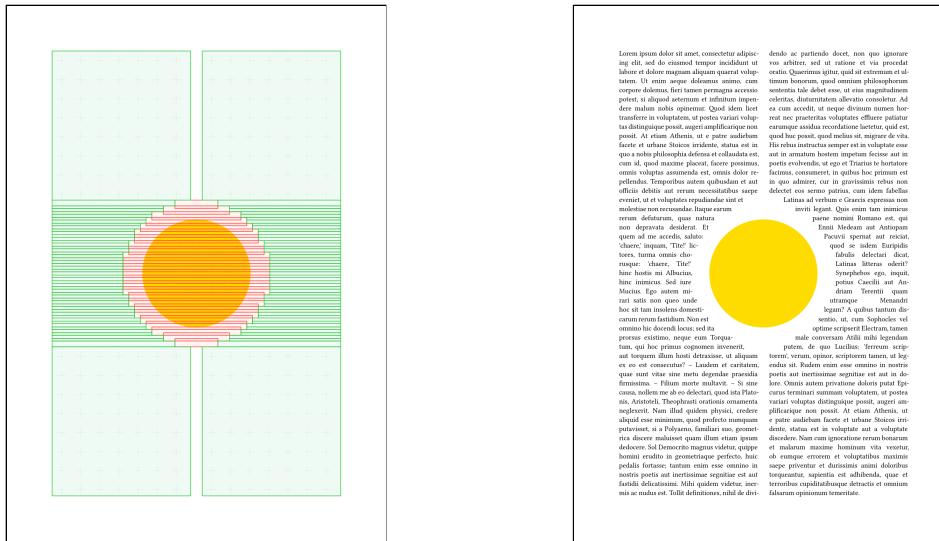
    content[
        #set par(justify: true)
        #lorem(590)
    ]
})
```



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.

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: *
  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: *
  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 Cicero's *Tusculan Disputations* (Book I, section 11) discussing the nature of the soul and its relationship to the body. It includes a diagram of a triangle divided into three horizontal layers.

III.3.2 Vertical rectangles

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



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

III Contouring

III.3 Boundaries as layers

- 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.

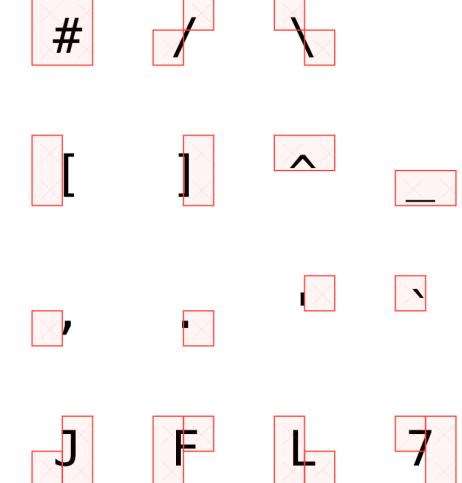
```
#meander.reflow({
    import meander: *
    placed(left + horizon,
          boundary:
            contour.height(
              div: 20,
              flush: horizon,
              x => (0.5, 1 - x),
            ) +
            contour.margin(5mm)
        )[...]
    // ...
})
```

Latin text placeholder: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 anim id est laborum.

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)
  ]
})
```

LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELIT, SED DO EIUSMOD TEMPOR INCIDUNT AT LABORE ET dolore magna aliqua quaeat voluptatem. Ut enim aequo dolorem, animo, cum corpore doleremus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impenderet, malum nobis opinetur. Quod idem licet transferre in voluptatem, ut postea variari voluntas distinguere non possit. At eliam Atheneis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et.

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)
  ]
})
```

LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELIT, SED DO EIUSMOD TEMPOR INCIDUNT AT LABORE ET dolore magna aliqua quaeat voluptatem. Ut enim aequo dolorem, animo, cum corpore doleremus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impenderet, malum nobis opinetur. Quod idem licet transferre in voluptatem, ut postea variari voluntas distinguere non possit. At eliam Atheneis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et.

Correct

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

LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELIT, SED DO EIUSMOD TEMPOR INCIDUNT AT LABORE ET dolore magna aliqua quaeat voluptatem. Ut enim aequo dolorem, animo, cum corpore doleremus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impenderet, malum nobis opinetur. Quod idem licet transferre in voluptatem, ut postea variari voluntas distinguere non possit. At eliam Atheneis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et.

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)]
})`

1

2

3

4

Notice: text from a 1-column layout overflows into a 2-column layout.

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)]


})

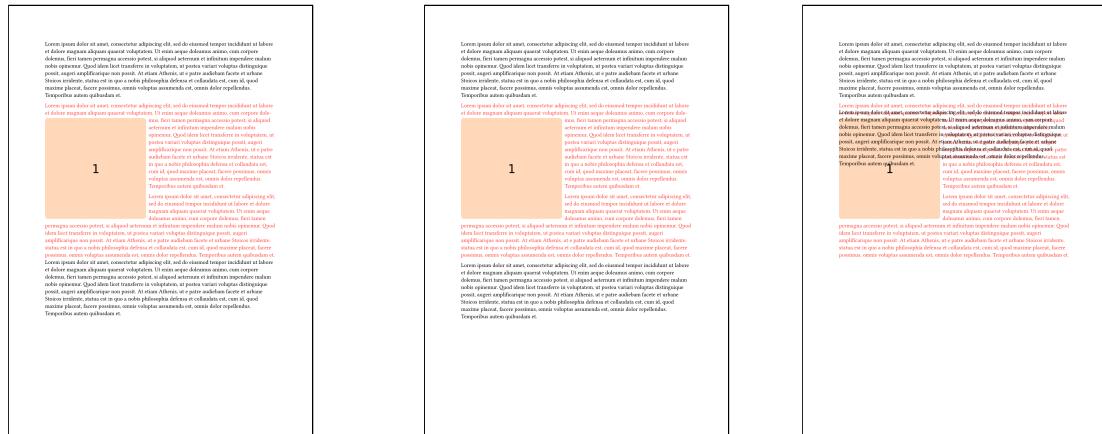
```



V.3 Colfill

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

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.

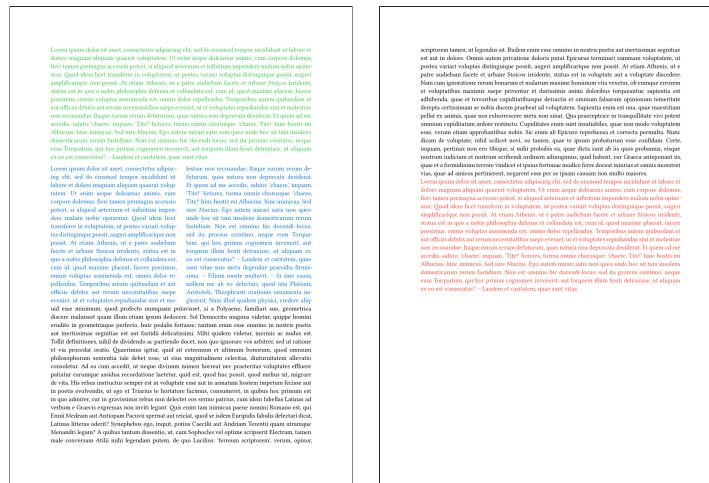
Default heuristic, with arti-
ficial spacing.

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)]
```

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()
})
```

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 et iam consequitur aliquando est sed de ratiōne tempore incidentia in literis
 et dolore impinguū dispergit quārum salutationem. Ut enim tempore dictaminis annis, sicut corporis
 dolores, fieri tenet per ratiōne accessus potest, si aliqd aeternum et infinitum impinguere malum
 nobis opinemur. Quod idem loci transire in voluptatem, ut protes variari voluptas distinguere
 possit, angusti amplificare non possit. At etiam Alcibiades, ut e patre audiebat facere et urbani
 Statim, quod dicitur, etiam in ratiōne delectari, quod dicitur, etiam in dolore repellendis. Sed
 maxime placuit, facere possumus, omnis voluptas assumenda est, omnis dolor repellendus.
 Temporibus autem quibusdam et sic officiis debitis aut rerum necessitatibus supēvenit, et si voluptas possit, non debet esse, sed si dolosus, non debet esse, sed si dolosus, non debet esse, sed
 natura non depravata desiderat. Et quoniam ad me accedit, subiecte clausae inquit, Tunc litteres,
 turma omnia chorusing: "chara, Tunc hinc hostis mihi Alcibiades, hinc inimicus. Sed iure Micius. Igo
 animi mei non debet esse, sed si dolosus, non debet esse, sed si dolosus, non debet esse, sed si
 omnino hic deocendi locutus, sed ipsa proxima existimat, neque cum Tropaeum, qui hoc prima
 cognoscere invenirent, aut torque illam hosti detraherem, ut aliquam ex eo est consequatur". - Laudem
 et castigant, et amant, et
 sine causa, sed illi me ab eo desiderat, quod ita Platonus, Aristoteles, Theophrasti orationis ornamenta
 neglegunt. Nam illud quidem physici, credere aliquid esse minimum, quod profecto minime
 pertinet ad rationem, et ratione, et
 deducere. Sol Democritus magnus volerat, quippe homini credidit in geometria perfecito, hinc
 pedalis fortasse; tantum enim esse omnino in nostris portis aut intermissione reginatis et aut familiari
 dissensu, et ratiōne, et
 partendo doceat, non quo ignorare vos arbitriis, sed ut ratione et via procedat. Quærimus
 igitur, quid si extremum et ultimum honestum, quod omnium philosophorum sententia tale debet
 esse, ut eius magnitudinem certeferat, distumulatim allevato consideretur. Ad ea cum accedit, ut neque

The following content overflows: Latin text placeholder content:
 honestus nec praeceps voluntatis offere patitur servitque ususq[ue] voluntatis voluntate, quid est
 quod non est voluntatis voluntate? Non enim voluntatis voluntate tempore in se possumus, sed in
 armatum honestum impinguū festis aut in poeta exhortandis, ut ego et Tiro in horitate facturus,
 consummari, ut quibus hoc primum aut in quo adiuvare, 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)]
})
```

mis ac modus est. Tollit definitiones, nihil de discursu, sed de cunctis etiam temporis incidentibus latore et dolore magnum aliquam querunt voluntates. Ut enim animus aduersus animus cum corpore dolens, fieri tamen per magna accessio potest, si aliquod aeternum et infinitum impinguatur. Quid ideo summa? Quid idem licet transferre in voluntatem, ut postea variatio voluntatis possit. Tunc, si agitur amplius, quae non possit. At etiam Athenae, ut e patre audiebam facie et urbane Stoicos iridebat, status est in quo a nobis philosophia defens et collaudata est, cum id, quod maxime placet, facere possumus, omnis voluntas assumenda est, omni modo possit. Tunc, si agitur amplius, quidlibet et aut officie debet, aut rerum necessitatis suae evenit, ut et voluntates repudiandae sint et consolatiae non recessant. Itaque eorum ratione definitum, ut natura non deprivata sit voluntas. Et quare ad eum, qui dicitur 'chariae' inquam, Tite! lectores, turba omnis chorister: 'Vivere! Tite!' hinc hostis mihi Albuscineus, etiam Murens. Ego autem minime satis mihi quis natus habeo, ut tam meum conseruum servem fastidio. Non et commo- decendi locus: sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognovit invertit, aut torquatum illud, ut detinisse, ut aliquippe potest et viceversa? Lasciva et caritatem, quae sunt vites sine metu degradat praelesia firmissima. — Filium morte multavi. — Si sine causa, nolleb me ab eo detinari, quod non paterem, non poterem. Nam illud quod paterem neglexerit. Nam illud quidam physici, credere aliquid esse minimum, quod profecto numeris potuissest, si a Polyceno, familiari suo, genitorem dimicaret, et quoniam illius, non ipse deponet. Sol Democritus minus volenter, quippe homini erudit in geometriam perfecto, huic pedulis fortasse; tantum enim esse commone in nostris poetis aut inertissimae segnitiae est aut fastidi delicatissimi. Mibi quidem videtur, iner-

1. secundum, ut neque divisionem manum horret nec peracterias voluntatis efficeri patitur earamque assidua recordatione latetur, quid est, quid haec possit, quod melius sit. migrale de vita His remus instructus super hoc in scriptis eiusdem ad in se obiectos impinguat, quae fecerit in poëtis evolvens, ut ego et Triarius te hortatice facimus, consumeret, in quibus hoc primum est in quibusque etiam gravissimis rebus nondelicata, sed etiam in aliis, quae non inviti legant. Quis enim tam inimicus parie nominis Romani est, qui Ennius Medea aut Antiloapan Faucii spernat aut recitat, quod se isdem Euripidis et quod se iste in aliis, quae non inviti legant. Sappholes ego, inquit, potius Coclidi aut Andream Terentii quam oratione Merenardi legant? A quibus etiam dissenserit, ut, cum Sophocles vel operis scriptor Eleutherius, etiam Merenardus Attici multo leguisse potest, de quo Lucilius 'terram scriptorem', verum, opinor, scriptorem tamen, ut legendus sit. Rudem enim esse omnem in aliis potest, aut inertissimae segnitiae est in aliis. Quare autem, quod paterem neglexerit, patet Epicurus terminus summum voluntatis ut postea variati voluntatis distinguenda possit, augeri amplificarique non possit. At etiam Athenae, ut paterem minime detinere et voluntate discere. Nam cum ignorantes rerum bonarum et malarum maxime hominis vita vexetur, ob europe etiamque et voluntatis omnes, quae sapientiam et diuinitatem ambo, quae etiam voluntas sapientia est abhorranda, quae et terroribus cupiditatisque distracta et omnium falsumarum opinionum temeritate derupta certissimam se non potest praetendere ad voluntatem. Sapientia enim est una, quae multitudinem pelet ex animis, quae nos ciboscerere metu nos stat. Qua preceptio in tranquillitate vivi potest omnium.

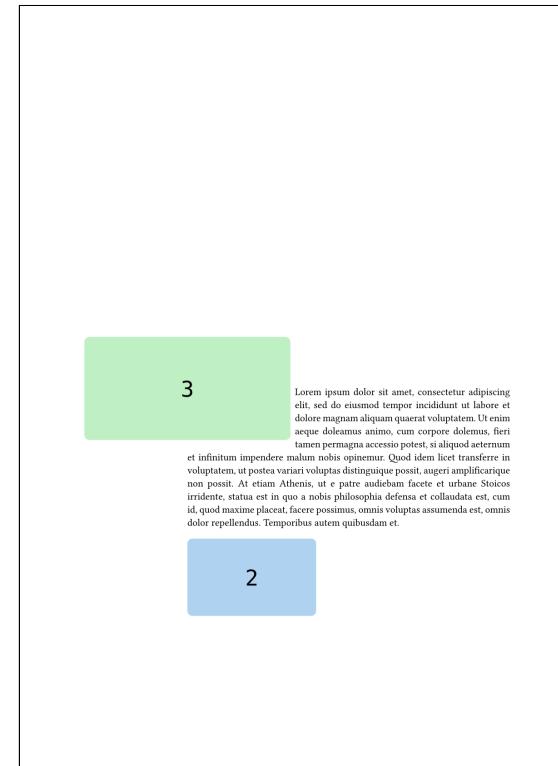
This is already doable globally by setting (boundary) to `#contour.phantom`. The innovation of (invisible) is that this can be done on a per-container basis.

VI.2 Position and length queries

The module `#query` contains functions that allow referencing properties of other elements. For example:

- whenever an `align` is required, such as for `#placed` or `#container`, you can instead pass a location dynamically computed via `#query.position`.
- whenever a `length` is required, such as for `(dx)` or `(height)` or a similar parameter, you can instead pass a length dynamically computed via `#query.height` or `#query.width`.

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



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: (),
    ) = {
    placed(
      // fetch position
      // of previous elem.
      query.position(tag, at: bottom + left),
      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()

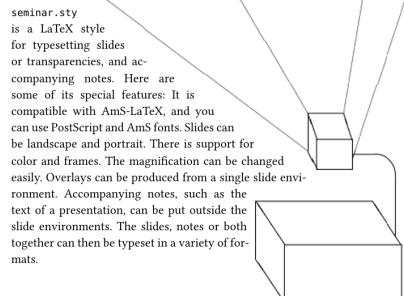
  // The actual content occupies
  // the complement of the
  // complement of the obstacles.
  container(invisible: <x>)
  content[
    #set par(justify: true)
    #lorem(225)
  ]
})
```

1. Quod idem licet transferre
in volutatem, ut postea variari volutas distingue 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 collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluntas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et a officiis debitibus aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedit, saluto: 'chaere' inquam, 'Tite!' 2. Tores, turma omnis choruscum: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari sat non quoque unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut alii 3. am ex eo est consecutus? — Laudem et caritatem, quae sunt vita sine metu degendae praesidia firmissima. — Filium morte multavit. — Si sine causa, nolleb me ab eo delectari, quod ista Platonis, Aristoteli, 5. Theophrasti orationis ornamenta.

Part VII

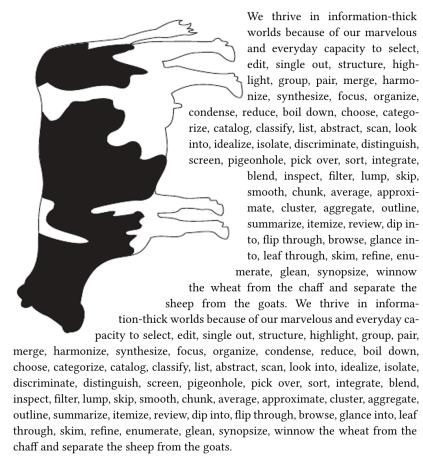
Showcase

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](#).



[tests/examples/typst-5181-a/test.typ](#)

Motivated by [github:typst/typst #5181 \(a\)](#)



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

Motivated by “Is there an equivalent to LaTeX’s \parshape?” (Typst forum)

VII Showcase

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Quis ut voluptatem. Ut enim acce delectus animi non compre delectus, fisi tamen pergitur, accessio potest, si aliquod adserimus et infinitum impendere malum nobis opinemus. Quod idem licet transference in voluptatem, ut postea variari voluntas distinguatur possit augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defessa et collaudata est, cum id, quod maxime placeat, facere possumus, omnis voluntas assumenda est, omnis dolor repellendum. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluntates repudiandas sint et molestiae non recusandae. Haecque earum rerum futurorum, quas natura non depravata desiderat. Et quem ad me accedit, saluto: 'chaser' inquit, 'Tito!' lictores, turma omnis chorique: 'chaser! Tito!' hinc hostis mi Albuscius, hinc inimicus. Sed iure Mucius. Ego autem mirari satius non quod hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognovit inveniret, aut torquem illum hosti detrahere, ut aliquam ex eo est consecutus? - Laudes et caritatis, quae sunt vitae sine metu degenerae praelesia firmissima. - Filium morte multavi. - Si sine causa, nollem me ab eo detegarti, quod ista Platonus, Aristoteles, Theophrasti orationis ornamenti neglexerit. Nam illud quidem physici, credere aliquid esse minimum, quod profecto nunquam putavisset, si a Polyaceno, familiari suo geometrica discere maluisse quam illum etiam ipsum dedocere. Si Democritus magnus videtur, quippe homini eruditio in geometria perfecto, huic pedalis fortasse tantum enim esse omnino in nostris poetis aut intermissimae segnitate est aut fastidiu deliciassim. Mili quidem videtur, inermis ac nudus est. Tollit definitiones, nihil de dividendo as pertinendo docet, non quo ignorare vos arbitris, sed ut ratione et via procedat oratio. Quaerimus igitur, quid sit extremum et ultimum bonorum, quod omnium philosophorum sententia tale debet esse, ut eius magnitudinem celestis, diuturnitatem allevato consoletur. Ad ea cum accedit, ut neque divinum humen horreat nec praeteritis voluntates effluere patitur earumque assidua recordatione lactet, quid est, quod huc possit, quod melius sit, migrare de vita. His rebus instrutus semper est in voluptate esse aut in armatum hostem impetu fecisse aut in poeta evolvidis, ut ego et Triarius te horatore facimus, consumeret, in quibus hoc primum est in quo admiratur, cur in gravissimis rebus non delectet eos sermo patrus, cum idem fabellas Latinas ad verbum e Gracis expressas non inviti legant. Quis enim tam inimicus pauci nomini Romana est, qui Enni Medeum aut Antiponum Pacuvii spernat aut reicit, quod isdem Euripidis fabellas delectari dicat, Latinas litteras oderit? Synephebos ego, inquit, potius Caecilius aut Andriam Terentii quam utramque Menandri legam? A quibus tantum dissentio, ut, cum Sophocles vel optime scrinxerit Electram, tamen male conversam Attili mihi legendam putem.



[tests/examples/typst-5181-b/test.typ](#)

Motivated by [github:typst/typst #5181 \(b\)](#)

Talmudifier Test Page

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis vehicula ligula at est bibendum, in eleifend erat dictum. Etiam sit amet telus id ex ullamcooper fauibus. Suspendsit sed elit vel neque orbi in. Sed invenit varius etiam ad elecentur. Phasellus lacus lectus sed amet orbi in, nrum malesuada sit diam. Cras pulvinar elit sit amet lacus fringilla, in elementum mauris maximus. Phasellus euismod dolor sed pretium elementum. Nulla sagittis, elit eget semper porttitor, erat nunc commodo turpis, et bibendum ex lorem laoreet ipsum. Morbi auctor dignissim velit egos. Aenean lobortis tincidunt nisi. Aliquam fringilla lectus ultrices porta. Aliquam feugiat finibus turpis id arcu tincidunt. Sed suspendisse tincidunt in arcu etiam pulvinar. Duis vel mattis facilisis ut tincidunt sed, pharetra libero. Aenean lobortis tincidunt nisi. 12 Praesent metus lacus, tristique sed porta non, tempus id quam. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Ris in porta velit, quis pellentesque elit. R2 Quisque vehicula massa laoreet sagittis vulputate. Fusce fringilla lectus ultrices porta. Aliquam feugiat finibus turpis id malesuada. Suspendsit hendrerit eros sit amet tempore vivendum. Duis velit mauris, facilisis ut etiam pulvinar. Fusce fringilla lectus ultrices porta non, tempus id quam. 1x I use these red Hebrew letters in my own WIP project along with various other font changes... You might find this functionality useful. R. Alter: I strongly disagree with you, and future readers of this will have to comb through pages upon pages of what might as well be lorem ipsum to figure out why we're dunking on each other so much. As it is written: "Sed ut eros id arcu tincidunt accusamus. Vestibulum vitae nisi blandit, commodo odio vitae, dictum nunc. Suspendsit pharetra lorem vitae ex arcu etiam pulvinar. Fusce fringilla lectus ultrices porta non, tempus id quam. 1x I use these red Hebrew letters in my own WIP project along with various other font changes... > Check out how nice that little hand looks. Nice. > Fusce porta pretium diam. Etiam venenatis nisl 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 du, 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](#)

Motivated by [github:typst/typst #5181 \(c\)](#)

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

↑ Since 0.2.2

`#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.

↑ Since 0.2.3

`#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

↑ Since 0.2.2 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)

↑ Since 0.2.3 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.

↑ Since 0.2.2

— Argument —

`(size): auto`

`length`

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

↑ Since 0.2.2

— Argument —

`(lang): auto`

`str`

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

↑ Since 0.2.2

— Argument —

`(hyphenate): auto`

`bool`

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

↑ Since 0.2.2

— Argument —

`(leading): auto`

`length`

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

↑ Since 0.2.1

`#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)`

`align` | `position`

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.regions`

`#meander.reflow((seq), (debug): none, (overflow): none, (placement): none) → 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.

— Argument —

`(debug): none`

↓ Until 0.2.4

Deprecated in favor of `opt.debug.post-thread()`. See [Section I.4](#) and [Section II.1](#) for more information.

— Argument —

`(overflow): none`

↑ Since 0.2.1

↓ Until 0.2.5

Deprecated in favor of `opt.overflow` options.

— Argument —

`(placement): none`

↑ Since 0.2.2

↓ Until 0.2.5

Deprecated in favor of `opt.placement` options.

↓ Until 0.2.4

`#meander.regions(..(args))`

Deprecated in favor of `#meander.reflow` with `opt.debug.pre-thread()`.

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.

<code>#contour.ascii-art</code>	<code>#contour.horiz</code>	<code>#contour.vert</code>
<code>#contour.grid</code>	<code>#contour.margin</code>	<code>#contour.width</code>
<code>#contour.height</code>	<code>#contour.phantom</code>	

↑ Since 0.2.1

`#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

- "J": top left
- "L": top right
- "7": bottom left
- "F": 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.

Argument —

{fun}

function

For each location, returns the top and bottom bounds.

(fraction) → (fraction, fraction)

#contour.width({div}: 5, {flush}: center, {fun}) → contour

Horizontal segmentation as (anchor, width).

Argument —

{div}: 5

int

Number of subdivisions.

Argument —

`(flush): center`

`align`

Relative horizontal alignment of the anchor.

Argument —

`(fun)`

`function`

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

`(fraction) → (fraction , fraction)`

VIII.4 Queries

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

`#query.height`

`#query.position`

`#query.width`

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

↑ Since 0.2.3

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.

Argument —

`(tag)`

`label`

Reference a previous element by its tag.

Argument —

`(transform): 100%`

`ratio | function`

Apply some post-processing transformation to the value.

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

↑ Since 0.2.3

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.

↑ Since 0.2.3 `#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.

<code>#opt.debug.minimal</code>	<code>#opt.debug.pre-thread</code>
<code>#opt.debug.post-thread</code>	<code>#opt.debug.release</code>

`#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.

<code>#opt.placement.phantom</code>	<code>#opt.placement.spacing</code>
-------------------------------------	-------------------------------------

#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.

#opt.overflow.alert	#opt.overflow.ignore	#opt.overflow.repeat
#opt.overflow.custom	#opt.overflow.pagebreak	#opt.overflow.state

#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:0.2.5": internals.fill-box
```

↑ Since 0.2.4

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:0.2.5": internals.geometry
```

↑ Since 0.2.4

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

```
#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

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

```
#geometry.align(
  {alignment},
  {dx}: 0pt,
  {dy}: 0pt,
  {width}: 0pt,
  {height}: 0pt,
  {anchor}: auto
) → (x: relative, y: relative)
```

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

— Argument —

{alignment}

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)) → boolTesting $a \leq b \leq c$, 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) → any**

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

Argument	<code>{val}</code>	any
Base value.		
Argument	<code>{min}: none</code>	any none
Lower bound.		
Argument	<code>{max}: none</code>	any none
Upper bound.		

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

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

Argument	<code>{frac}</code>	block(fraction)
Child dimensions as fractions.		
Argument	<code>{abs}</code>	block(length)
Parent dimensions as absolute lengths.		
Argument	<code>...{style}</code>	
Currently ignored.		

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

Resolves an anchor point relative to a region.

Argument	<code>{region}</code>	block
A block (x: length, y: length, width: length, height: length).		
Argument	<code>{alignment}</code>	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.

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

Fetch all required answers to geometric queries. See [Section VIII.4](#) for details.

— 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.get-page-offset      #tiling.push-elem
#tiling.blocks-of-container #tiling.is-ignored          #tiling.separate
#tiling.blocks-of-placed   #tiling.next-elem           #tiling.placement-mode
#tiling.create-data         #tiling.polygon            #tiling.push-block
```

#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}: ())

→ opaque

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

— Argument —

`(size): none``size`

Dimensions of the page

— Argument —

`(page-offset): (x: Opt, y: Opt)``size`

Optional nonzero offset on the top left corner

— Argument —

`(elems): ()``(..elem,)`

Elements to dispense in order

#tiling.get-page-offset → `(x: Length, y: Length)`

^ context

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.



`#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 })
```



IX.5 Bisection

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

`#bisect.default-rebuild({inner-field})[ct] → (dictionary, 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((dims), (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.

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-children(({split-dispatch}, {fits-inside}, {cfg}))[ct] →
(content?, content?)
```

Split content with a "children" main field.

Strategy: take all children that fit.

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-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).

Argument —

{cfg}

dictionary

Extra configuration options.

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

Split a list.item.

Strategy: recursively split the body, and do some magic to simulate a bullet point 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).

Argument
`(cfg)` dictionary
 Extra configuration options.

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

Split one word according to hyphenation patterns, if enabled.

Argument
`{ww}` str
 Word to split.

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

Argument
`{cfg}` dictionary
 Extra configuration options.

#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 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.