

MEANDER

User guide

v0.2.3

2025-09-25

MIT

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

NEVEN VILLANI

✉ neven@crans.org

MEANDER implements a content layout algorithm that supports wrapping text around images of arbitrary shape. Though very different in its modeling, **MEANDER** can be seen as a Typst alternative to L^AT_EX's `wrapfig` and `parshape`.

Internally this is enabled by

1. a page segmentation algorithm that detects floating content and arranges containers around them,
2. a text bisection algorithm to recursively explore and split content,
3. a threading algorithm (threading is when text that overflows from one box continues to a different box).

Contributions

If you have ideas or complaints, you're welcome to contribute to **MEANDER** by submitting a [bug report](#), [feature request](#), or [pull request](#).

Versions

- [dev](#)
- [0.2.2 \(latest\)](#)
- [0.2.1](#)
- [0.2.0](#)
- [0.1.0](#)



Table of Contents

Quick start	3	VIII.3 Contouring	39
I.1 A simple example	3	VIII.4 Bisection	42
I.2 Multiple obstacles	4	VIII.5 Threading	48
I.3 Columns	5		
Understanding the algorithm	6	Modularity (WIP)	50
II.1 Page tiling	6	About	51
II.2 Content bisection	7	X.1 Related works	51
II.3 Threading	8	X.2 Dependencies	51
Contouring	9	X.3 Acknowledgements	51
III.1 Margins	9		
III.2 Boundaries as equations	10		
III.3 Boundaries as layers	12		
III.3.1 Horizontal rectangles	12		
III.3.2 Vertical rectangles	13		
III.4 Autocontouring	14		
III.5 More to come	15		
Styling	16		
IV.1 Paragraph justification	16		
IV.2 Font size and par leading	17		
IV.3 Hyphenation and language	18		
IV.3.1 Styling containers	19		
Multi-page setups	20		
V.1 Pagebreak	20		
V.2 Colbreak	20		
V.3 Placement	22		
V.3.1 page	22		
V.3.2 box	22		
V.3.3 float	22		
V.3.4 Use-case	22		
V.4 Overflow	23		
V.4.1 No overflow	23		
V.4.2 Predefined layouts	24		
V.4.3 Custom layouts	26		
Showcase	27		
Public API	29		
VII.1 Elements	29		
VII.2 Layouts	32		
Internal module details	34		
VIII.1 Geometry	34		
VIII.2 Tiling	36		

Part I

Quick start

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](#).

1.1 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.

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.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)
    ]
})
```



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

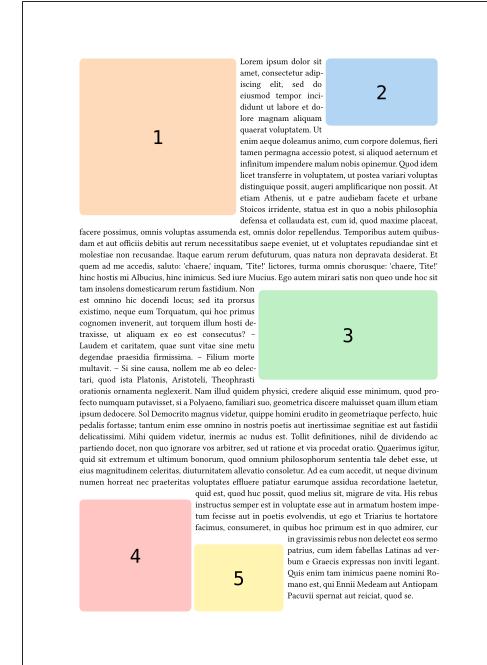
A single `#meander.reflow` invocation can contain multiple `placed(...)` objects. A possible limitation would be performance if the number of obstacles grows too large, but experiments have shown that up to ~100 obstacles is still workable.

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)
    ]
})
```



Technically, **MEANDER** can only handle rectangular obstacles. However, thanks to this ability to handle 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 animus.

3

melius sit, migrare de vita. His rebus institutus semper est in volupitate esse aut in armatum hostem impetrare. Ita se aut in potest evocandi, ut in securitate te horatore factum, consumetur, in quibus hoc primum et in quo admirer, cur in gravissimis rebus non detectet eos sermo patrius, cum idem fabellas Latinas ad verbum e Graecis expressis non possit legant. Quis enim tam audiret, si in securitate Romano est, qui Tanti Mediam aut An-

2

tiopam Pacuvii spernit aut recitat, quod se idem Euripidis fabulis detectri dicit. Latinas litteras oderit? Synephebos ego, inquit, et Caelestis aut Androm. Tercium quid utrimumque Menandri legant? A quibus tam tamen dissentio, ut, cum Sophocles vel optime scripsisset Electram, tamen male conversam Attili mili legendam putem, de quo-

1

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 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: *
    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 are 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.

You can have a visual representation of these regions by replacing `#meander.reflow` with `#meander.regions`, with the same inputs.

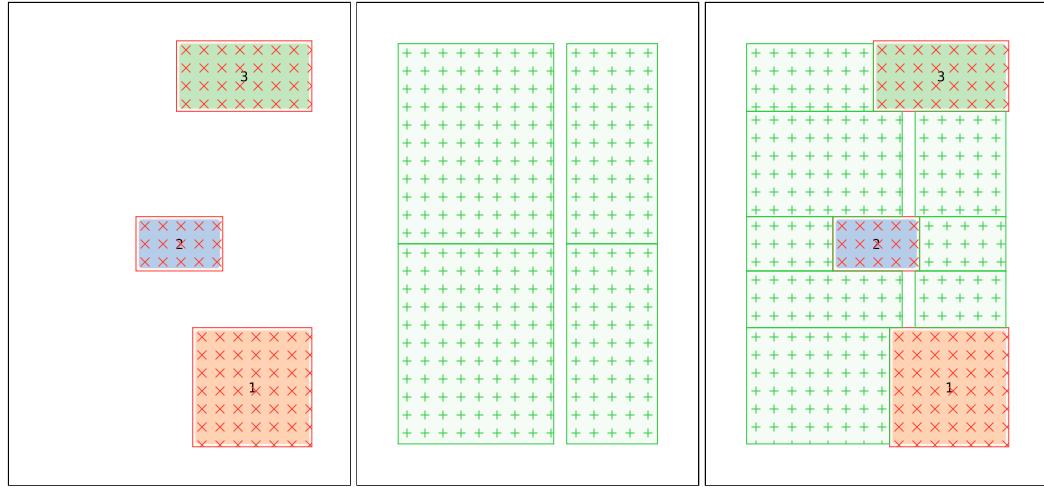


Figure 1: Left to right: the forbidden, allowed, and combined regions

II.2 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.3 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.

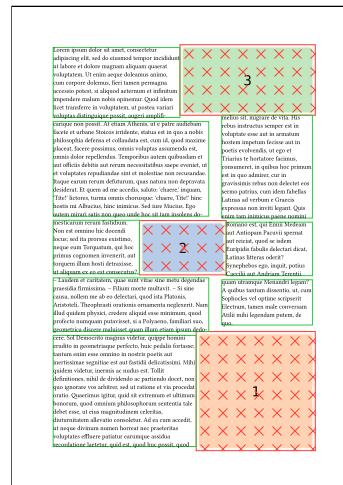


Figure 2: Debug view of the final output of `#meander.reflow`

Every piece of content produced by `#meander.reflow` is placed, and therefore does not affect layout outside of `#meander.reflow`. See [Section V.3](#) 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.

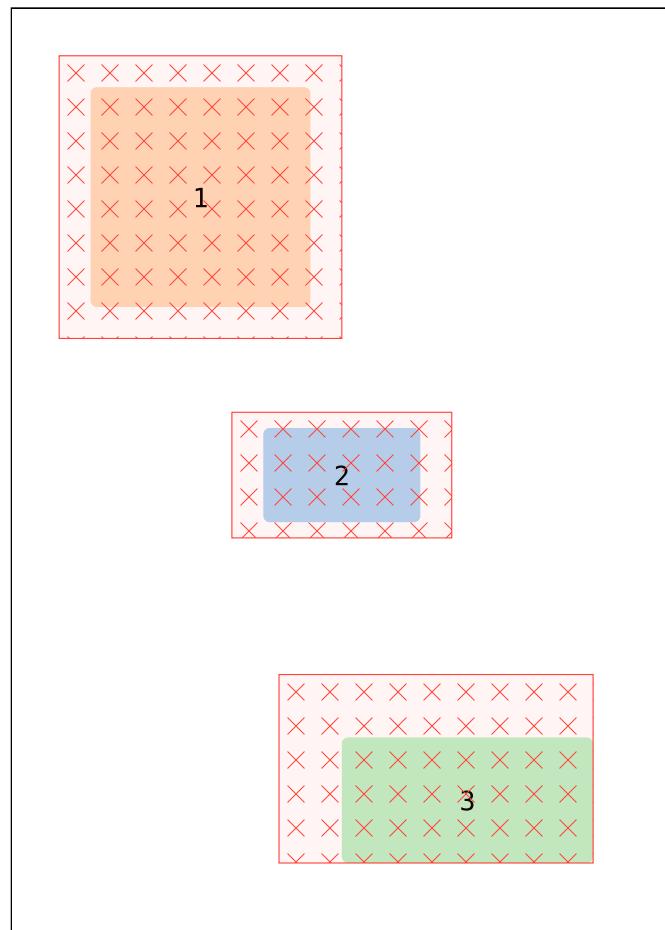
This is not a mistake. The reality is that these statements are only incompatible if we assume that 1 image = 1 obstacle. We call “contouring functions” the utilities that allow splitting one image into multiple obstacles to approximate an arbitrary shape.

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.regions({
    import meander: *
    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,
    )
})
```



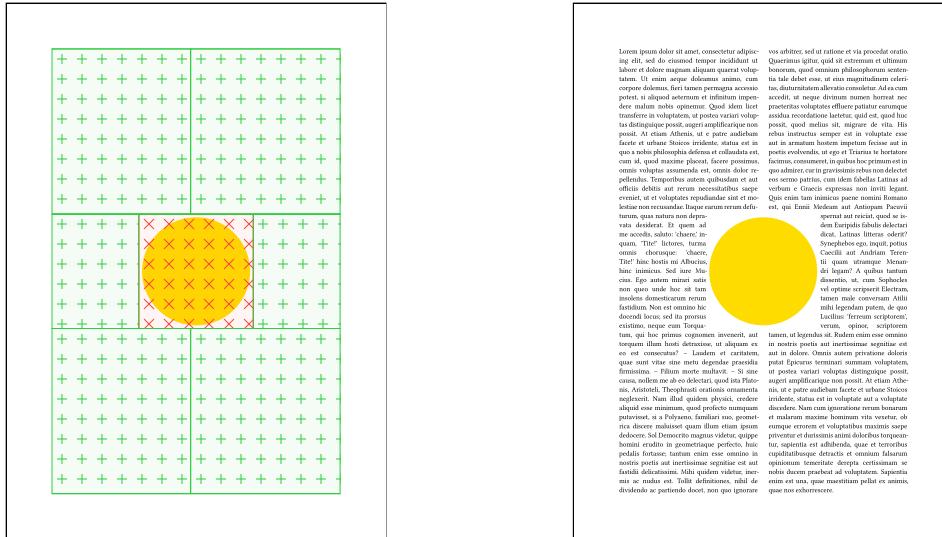
III.2 Boundaries as equations

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()

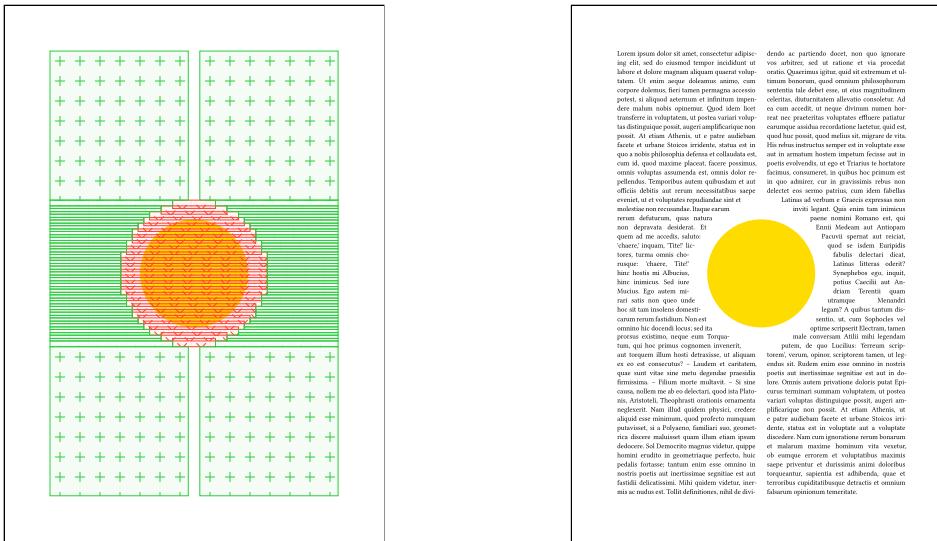
    content[
        #set par(justify: true)
        #lorem(590)
    ]
})
```



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)
  )[...]
  // ...
})
```

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:

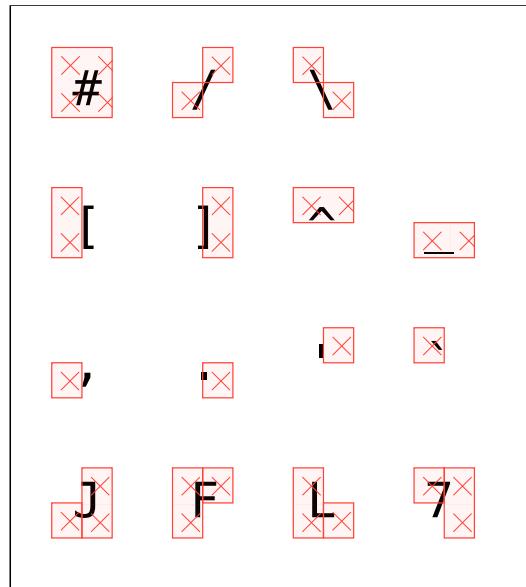
- if flush = horizon, the anchor point will be the middle of the obstacle;
 - if flush = bottom, the anchor point will be the bottom of the obstacle.

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

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(debug: true, {
    import meander: *
    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%
```

```
#meander.reflow({
    import meander: *
    placed(top + left,
        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 snippets. On the left, a red box labeled "Wrong" contains the following code:

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

On the right, a black box labeled "Correct" contains the following code:

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

Both snippets are shown next to a preview window showing justified text. The "Wrong" example results in a warning message: "Warning: Text is not correctly justified. It is assumed that justify is only used on the first line of a paragraph, and that the rest of the lines are aligned to the first line." The "Correct" example does not show this warning.

Correct

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

Latin 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, ut aliquip ex ea commodo consequat. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusamus et erroribus summis nostris doloribus et dolorebus. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusamus et erroribus summis nostris doloribus et dolorebus.

IV.2 Font size and par 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)
  ]
})
```

Latin 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, ut aliquip ex ea commodo consequat. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusamus et erroribus summis nostris doloribus et dolorebus. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusamus et erroribus summis nostris doloribus et dolorebus.

Correct

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

Latin 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, ut aliquip ex ea commodo consequat. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusamus et erroribus summis nostris doloribus et dolorebus. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusamus et erroribus summis nostris doloribus et dolorebus.

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 quærat voluptatem. Ut enim aequo doleamus animo, cum corpore dolemus, fieri tamen permagno accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transference in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique 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 quærat voluptatem. Ut enim aequo doleamus animo, cum corpore dolemus, fieri tamen permagno accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transference in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique 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 quærat voluptatem. Ut enim aequo doleamus animo, cum corpore dolemus, fieri tamen permagno accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transference in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos.

Correct

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

Quod latine loqui, nesciunt voluntatem, ut postea variari voluntas distinguere possit, angari amplificari non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos.

IV.3.1 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)]
})
```

Latin ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam quaerat volupatem. Ut enim auctor dolorem animo, cum corpore dolens, fieri tamen permagna accessus protestat, si aliquid aeternum et infinitum impinguatur, nobile opinatur. Quod latine loqui, nesciunt voluntatem, ut postea variari voluntas distinguere possit, angari amplificari non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos.

Part V

Multi-page setups

V.1 Pagebreak

`MEANDER` can deal with text that spans multiple pages, you just need to place `pagebreak()`s 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])

})
```

*Consectetur
adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim acque delectarunt animo, e*

*et
n* Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua
quoniam voluptatem. Ut enim aequa dolorem

Laorem ipsum dolor sit amet, consectetur adipisciing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Quis pariatur? 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 animus.

minimum, quod prefecto omnia quia potuisse, si a Polyphemus, familiari suo, geometrica discere maluisse, quam illum etiam ipsum deducere. Sol Deucrato magna videtur, quippe bonitati

compiled: 2025-09-25

V.3 Placement

Placement options control how a `#meander.reflow` invocation is visible by and sees other content. This is important because `MEANDER` places all its contents, so it is by default invisible to the native layout.

V.3.1 page

The default, and least opinionated, mode is `placement: page`.

- suitable for: one or more pages that `meander` has full control over.
- advantages: supports pagebreaks, several invocations can be superimposed, flexible.
- drawbacks: superimposed with content that follows.

V.3.2 box

The option `placement: box` will emit non-placed boxes to simulate the actual space taken by the `MEANDER`-controlled layout.

- suitable for: an invocation that is part of a larger page.
- advantages: supports `pagebreak()`, content that follows is automatically placed after.
- drawbacks: cannot superimpose multiple invocations.

V.3.3 float

Finally, `placement: float` produces a layout that spans at most a page, but in exchange it can take the whole page even if some content has already been placed.

- suitable for: single page layouts.
- advantages: gets the whole page even if some content has already been written.
- drawbacks: does not support pagebreaks, does not consider other content.

V.3.4 Use-case

Below is a layout that is not (as easily) achievable in `page` as it is in `box`. Only text in red is actually controlled by `MEANDER`, the rest is naturally placed before and after. This makes it possible to hand over to `MEANDER` only a few paragraphs where a complex layout is required, then fall back to the native Typst layout engine.

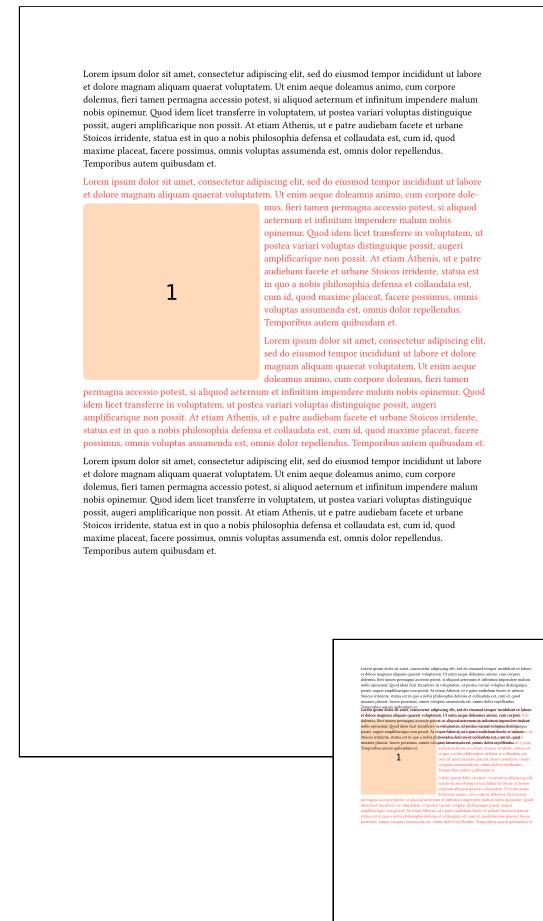
```
#lorem(100)

#meander.reflow(placement: box, {
    import meander: *
    placed(top + left, dy: 1cm,
           my-img-1)
    container()
    content[
        #set text(fill: red)
        #lorem(100)

        #lorem(100)
    ]
})

#lorem(100)
```

For reference, to the right is the same page if we omit `placement: box`, where we can see a glitchy superimposition of text.



V.4 Overflow

By default, if the content provided overflows the available containers, it will show a warning. This behavior is configurable.

V.4.1 No overflow

The default behavior is `overflow: false` because it avoids panics while still alerting that something is wrong. The red warning box suggests adding more containers or a pagebreak to fit the remaining text. Setting `overflow: true` will silently ignore the overflow, while `overflow: panic` will immediately abort compilation.

```
overflow: false
```

```
#meander.reflow(
  overflow: false,
  import meander: *
  container()
  content[#lorem(1000)]
)}
```



```
overflow: true
```

```
#meander.reflow(
  overflow: true,
  import meander: *
  container()
  content[#lorem(1000)])
})
```



```
overflow: panic
```

```
#meander.reflow(
  overflow: panic,
  import meander: *
  container()
  content[#lorem(1000)])
})
```

(panics)

V.4.2 Predefined layouts

The above options are more useful if you absolutely want the content to fit in the defined layout. A commonly desired behavior is for the overflow to simply integrate with the layout as gracefully as possible. That is the purpose of the two options that follow.

With `overflow: pagebreak`, any content that overflows is placed on the next page. This is typically most useful in conjunction with `placement: page`, and is outright incompatible with `placement: float` (because it does not support pagebreaks; see [Section V.3](#)).

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

base text is in blue, overflow is in black, and native text is in red.

base text is in blue, overflow is in black, and native text is in red.

Blue text is part of the base layout. The overflow is in black, and is naturally followed by native text in red.

As for `overflow: text`, it is similarly best suited in conjunction with `placement: box`, and simply writes the text after the end of the layout.

```
#meander.reflow(
  placement: box,
  overflow: text,
  import meander: *
  container(
    width: 48%,
    height: 50%,
    style:
      (text-fill: blue),
  )
  container(
    width: 48%,
    height: 50%,
    align: right,
    style:
      (text-fill: blue))
  content[#lorem(1000)]
) \
#text(fill: red)[
  #lorem(100)
]
```

base text is in blue, overflow is in black, and native text is in red.

base text is in blue, overflow is in black, and native text is in red.

In both cases, any content that follows the `#reflow` invocation will more or less gracefully follow after the overflowing text, possibly with the need to slightly adjust paragraph breaks if needed.

V.4.3 Custom layouts

If your desired output does not fit in the above predefined behaviors, you can fall back to writing a custom overflow handler. Any function that returns content can serve as handler, including another invocation of `#reflow`. This function will be given as input a dictionary with fields:

- `styled` has all styling options applied and is generally what you should use,
 - `structured` is suitable for placing in another `#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.

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

```
#meander.reflow(  
  placement: box,  
  overflow: tt => [  
    #set text(fill: red, size: 25pt)  
    *The following content overflows:  
    _#{tt.styled}_  
  ], {  
  import meander: *  
  container(height: 50%)  
  content[#lorem(400)]  
})
```

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. These applications should remain last-resort measures, and if you find that a layout you would want to achieve is only possible by chaining together several `#meander.reflow` handlers, consider instead [reaching out](#) to see if there is a way to make this layout better supported.

Part VI

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](#).

`seminar.sty`
is a LaTeX style
for typesetting slides
or transparencies, and ac-
companying notes. Here are
some of its special features: It is
compatible with AmS-LaTeX, and you
can use PostScript and AmS fonts. Slides can
be landscape and portrait. There is support for
color and frames. The magnification can be changed
easily. Overlays can be produced from a single slide envi-
ronment. Accompanying notes, such as the
text of a presentation, can be put outside the
slide environments. The slides, notes or both
together can then be typeset in a variety of for-
mats.

[examples/5181-a/main.typ](#)

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

Latin text (from the Iliad):

... et quod profecto manquas putasset, si a Polyaeo, familiari suo, ge-
ometrica discere malueras, illum estinque debet. Sol Democrito magus videtur, quippe hunc per-
ditio et amantricorum perfecto, hinc pedulis fortius, tamquam omni eximius, post positi aut
prosternimus signum est aut fastidi deliciacioni. Mitto videlicet in membra et munita. Tali defin-
itiones, nihil de dividendo a per partendo docet, non quo ignorare vos arbitror, sed at ratione et via pro-
cedat oratio. Quærimus igitur quid sit extremum et ultimum bonorum, quod omnium phantasorum
sentientia tale debet esse, ut eius magnitudinem celestes, diuturnitatem allevatio consuletur. Ad
ea cum accedit, ut neque diruum nomen horret nec praeteritis voluptatis effluere patiatas
earumque assidua reconvoluta lastet, quid est, quod huc possit, quod melius sit, migrare de
vita. His rebus instructus semper est in voluptate esse aut in armatum hostem impetum
fecisse aut in poenis evolvenda, si ego et Tirianus te horritore facimus, consumeret, in
quibus hoc primum est in quo admirer, cur in gravissimis rebus non dilectet eos sermo
patius, cum idem fabella Latinas ad verbura e Graecis expressas non inviti legant. Quis
enim tam innicuus paenit nomini Romani est, qui Enni Medeam aut Antopam
Pacavii spernit aut recitat, quod se isdem Euripidis fabulis delectari dicat, Latinas
litteras oderit? Synephebos ego, inquit, potius Caccilii aut Andriam Terentii quam
utramque Menandri legam? A quibus tantum dissentio, ut, cum Sophocles vel
optime scrinxerit Electram, tamen male conversam Attili mihi legendarum putem.



[examples/5181-b/main.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 tellus id ex ullamcorper faucibus. Suspendisse sed vel neque convallis iaculis id ut orna. Sed tincidunt varius ipsum at scelerisque. Phasellus lacus sodales sit amet orci in, rutrum malesuada diam. Cras pulvinar elit sit amet laetus 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 eget consequat. R. Seth: Blah blah blah. As it is written: "Accens lacinia nisi diam, vel pulvinar metus aliquet ut. Sed non lorem quis dui ultricies volutpat quis at diam".¹²³ Quisque at nisi magna. Duis nec lacus arcu. Morbi vel fermentum leo. Pellentesque hendrerit sagittis vulputate. Fusce laoreet malesuada odio, sit amet fringilla lectus ultrices porta. Aliquam feugiat finibus turpis id malesuada. Lx Suspendisse hendrerit eros sit amet tempor pulvinar. Duis velit mauris, facilisis ut tincidunt sed, pharetra eu libero. Aenean lobortis tincidunt nisi. L2 Praesent metus lacus, tristique sed porta non, tempus id quam. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Rx In eu porta velit, quis pellentesque elit. R2 Quisque vehicula massa sit amet justo rhoncus auctor. Ut enim feugiat finibus turpis id malesuada. Accumsan. J Fusce sapien ipsum, cursus a tincidunt vel, dignissim eu mi. Morbi id velit ac turpis ullamcorper lacinia. J Cras bibendum telus vitae eros rutrum scelerisque. Vivamus sed pellentesque elit, non imperdiet massa. Curabitur dictum nisi sollicitudin luctus malesuada. Vestibulum id pulvinar risus, sit amet ornare libero. Etiam a nunc dolor. R2 In ac velit maximus, elementum ex et, blandit massa. Aliquam vehicula at neque sit amet ultricies. Integer id justo est. Quisque luctus erat eget aliquam faucibus. Etiam eu mi ac odio pretium dictum. Vestibulum viverra congue risus, ac egestas est dapibus eget. Aenean ut orci leo. Nulla dignissim vel pulvinar elit facilisis, ac venenatis leo tincidunt. Quisque eu lorem tortor. Quisque nec porttitor elit. Ut finibus ullamcorper odio, in porttitor lorem suscipit ut.

Suspenisse pharetra lorem vitae ex tincidunt orna. Maecenas efficitur tristique libero, eget commodo massa. Pellentesque liber sem, interdum ut nibh interdum, compactum elementum magna.¹²⁴ [https://typst.org/80f420125](#) Aliquam facilisis vel turpis eu semper. Donec eget purus lectus. → Check out how nice that little hand looks. Note: → Fusce porta pretium diam. Etiam fermentum sed malesuada fringilla. Vivamus vehicula nunc sed libero scelerisque viverra a quis libero. Integer ac urna ut lectus faucibus mattis ac id nunc. Morbi fermentum magna dui, at rhoncus nibh porttitor quis. Donec dui ante, semper non quam at, accumsan volutpat leo. Maecenas magna risus, finibus sit amet felis ut, vulputate euismod nunc.

[examples/talmudifier/main.typ](#)

From [github:subalterngames/talmudifier](https://github.com/subalterngames/talmudifier)

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



We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synthesize, winnow the wheat from the chaff and separate the sheep from the goats. We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, iso-

[examples/cow/main.typ](#)

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

Part VII

Public API

VII.1 Elements

#colbreak	#container	#pagebreak
#colfill	#content	#placed

#colbreak → flowing

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

#colfill → flowing

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

```
#container(  
  {align}: top + left,  
  {dx}: 0% + 0pt,  
  {dy}: 0% + 0pt,  
  {width}: 100%,  
  {height}: 100%,  
  {style}: (),  
  {margin}: 5mm,  
  {ignore-labels}: (),  
  {name}: none,  
  {tags}: ()
```

) → container

Core function to create a container.

— Argument —

```
{align}: top + left
```

alignment

Location on the page.

— Argument —

```
{dx}: 0% + 0pt
```

relative

Horizontal displacement.

— Argument —

```
{dy}: 0% + 0pt
```

relative

Vertical displacement.

— Argument —

`(width): 100%`

relative

Width of the container.

— Argument —

`(height): 100%`

relative

Height of the container.

— Argument —

`(style): ()`

dictionary

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

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

— Argument —

`(margin): 5mm`

length

Margin around the eventually filled container so that text from other paragraphs doesn't come too close.

— Argument —

`(ignore-labels): ()`

array(label)

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

— Argument —

`(name): none`

label | none

Optional unique name so that future elements can refer to this one.

— Argument —

`(tags): ()`

array(label)

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

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

→ **flowing**

Core function to add flowing content.

— Argument —

`(data)`

content

Inner content.

— Argument —	<code>{size}: auto</code>	<code>length</code>
<code>Applies #set text(size: ...).</code>		
— Argument —	<code>{lang}: auto</code>	<code>str</code>
<code>Applies #set text(lang: ...).</code>		
— Argument —	<code>{hyphenate}: auto</code>	<code>bool</code>
<code>Applies #set text(hyphenate: ...).</code>		
— Argument —	<code>{leading}: auto</code>	<code>length</code>
<code>Applies #set par(leading: ...).</code>		

#pagebreak

Continue layout to next page.

```
#placed(
  {align},
  {dx}: 0% + 0pt,
  {dy}: 0% + 0pt,
  {boundary}: (auto,),
  {display}: true,
  {name}: none,
  {tags}: ()
```

) [content] → obstacle

Core function to create an obstacle.

— Argument —	<code>{align}</code>	<code>alignment</code>
<code>Reference position on the page (or in the parent container).</code>		
— Argument —	<code>{dx}: 0% + 0pt</code>	<code>relative</code>
<code>Horizontal displacement.</code>		
— Argument —	<code>{dy}: 0% + 0pt</code>	<code>relative</code>
<code>Vertical displacement.</code>		
— Argument —	<code>{boundary}: (auto,)</code>	<code>(..function,)</code>

An array of functions to transform the bounding box of the content. By default, a 5pt margin. See `contour.typ` for a list of available functions.

Argument —

`(display): true`

bool

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

- `boundary: 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 —

`(name): none`

label | none

Optional unique name so that future elements can refer to this one.

Argument —

`(tags): ()`

array(label)

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

VII.2 Layouts

#reflow

#regions

`#reflow({seq}, {debug}: false, {overflow}: false, {placement}: page) → content`

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

Argument —

`(seq)`

seq

See module `tiling` for how to format this content.

Argument —

`(debug): false`

bool

Whether to show the boundaries of boxes.

— Argument —

`(overflow): false`

any

Controls the behavior in case the content overflows the provided containers.

- `false` -> adds a warning box to the document
- `true` -> ignores any overflow
- `pagebreak` -> the text that overflows is simply placed normally on the next page
- `panic` -> refuses to compile the document
- `any content` => content function -> uses that for formatting

— Argument —

`(placement): page`

Relationship with the rest of the content on the page.

- `page`: content is not visible to the rest of the layout, and will be placed at the current location. Supports pagebreaks.
- `box`: meander will simulate a box of the same dimensions as its contents so that normal text can go before and after. Supports pagebreaks.
- `float`: similar to `page` in that it is invisible to the rest of the content, but always placed at the top left of the page. Does not support pagebreaks.

`#regions({seq}, {display}: true, {placement}: page) → content`

Debug version of the toplevel reflow, that only displays the partitioned layout.

— Argument —

`(seq)`

seq

Input sequence to segment. Constructed from placed, container, and content.

— Argument —

`(display): true`

bool

Whether to show the placed objects (`true`), or only their hitbox (`false`).

— Argument —

`(placement): page`

Controls relation to other content on the page. See analogous placement option on `reflow`.

Part VIII

Internal module details

VIII.1 Geometry

```
#geometry.align          #geometry.clamp        #geometry.resolve  
#geometry.between        #geometry.intersects
```

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

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

— Argument —

(alignment)

alignment

Absolute alignment.

— Argument —

(dx): 0pt

relative

Horizontal displacement.

— Argument —

(dy): 0pt

relative

Vertical displacement.

— Argument —

(width): 0pt

relative

Object width.

— Argument —

(height): 0pt

relative

Object height.

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

Testing $a \leq b \leq c$, 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 —

{val}

any

Base value.

— Argument —

{min}: none

any | none

Lower bound.

— Argument —

{max}: none

any | none

Upper bound.

#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)

(width: length, height: length)

Size of the container as given by the layout function.

— Argument —

..(args)

dictionary

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

VIII.2 Tiling

#tiling.add-self-margin	#tiling.is-ignored	#tiling.push-elem
#tiling.create-data	#tiling.next-elem	#tiling.separate
#tiling.elem-of-container	#tiling.pat-allowed	
#tiling.elem-of-placed	#tiling.pat-forbidden	

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

Applies an element's margin to itself.

#tiling.create-data({size}: none, {elems}: ())

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

— Argument —

(size): none

(width: length, height: length)

Dimensions of the page

— Argument —

{elems}: ()

(..elem,)

Elements to dispense in order

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

See: `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)` container
 Container to segment.

`#tiling.elem-of-placed((data), (obj)) → elem`

See: `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)` placed
 Object to measure, pad, and place.

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

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

Argument
`(container)`
 Must have the field `_.aux.ignore-labels`, as containers do.

Argument
`(obstacle)`
 Must have the fields `_.aux.tags` and `_.aux.name`, as obstacles do.

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

This function is reentering, allowing interactive computation of the layout. Given its internal state data, `next-elem` uses the helper functions `elem-of-placed` and `elem-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.pat-allowed({sz}) → pattern

Pattern with green pluses to display allowed zones.

Argument —

{sz}

length

Size of the tiling.

#tiling.pat-forbidden({sz}) → pattern

Pattern with red crosses to display forbidden zones.

Argument —

{sz}

length

Size of the tiling.

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

Updates the internal state to include the newly created element.

Argument —

{data}

opaque

Internal state.

Argument —

{elem}

elem

Element to register.

#tiling.separate({seq}) → {pages: array, flow: (..content,)}

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

An “obstacle” is data produced by the placed function. It can contain arbitrary content, and defines a zone where flowing content cannot be placed.

A “container” is produced by the function container. It defines a region where (once the obstacles are subtracted) is allowed to contain flowing content.

Lastly flowing content is produced by the function content. It will be threaded through every available container in order.

```
1 #separate({
2     // This is an obstacle
```

```

3   placed(top + left, box(width: 50pt, height: 50pt))
4   // This is a container
5   container(height: 50%)
6   // This is flowing content
7   content[#lorem(50)]
8 }

```

— Argument —

(seq)

seq

A sequence of constructors placed, container, and content.

VIII.3 Contouring

#contour.ascii-art	#contour.height	#contour.phantom
#contour.frac-rect	#contour.horiz	#contour.vert
#contour.grid	#contour.margin	#contour.width

#contour.ascii-art((ascii))

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)`

Draw the shape of the image in ascii art.

[code](#) | [str](#)

```
#contour.frac-rect(({frac}, {abs}, ..(style))
```

→ `(x: length, y: length, width: length, height: length)`

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

Argument
`{frac}` `(x: fraction, y: fraction, width: fraction, height: fraction)`

Child dimensions as fractions.

Argument
`{abs}` `(x: length, y: length, width: length, height: length)`

Parent dimensions as absolute lengths.

Argument
`..(style)`

Currently ignored.

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

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(fraction, fraction) => bool`

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

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

Vertical segmentation as (anchor, height).

Argument
`{div}: 5` `int`

Number of subdivisions.

Argument
`{flush}: horizon` `alignment.`

Relative vertical alignment of the anchor.

Argument
`(fun)` `function(fraction) => (fraction, fraction)`
 For each location, returns the position of the anchor and the height.

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

Horizontal segmentation as (left, right)

Argument
`{div}: 5` `int`
 Number of subdivisions.

Argument
`(fun)` `function(fraction) => (fraction, fraction)`
 For each location, returns the left and right bounds.

#contour.margin(..{args}) → function

Contouring function that pads the inner image.

Argument
`..{args}`
 May contain the following parameters, ordered here by decreasing generality and increasing precedence

- 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 → function

Drops all boundaries. Using boundary: phantom will let other content flow over this object.

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

Vertical segmentation as (top, bottom)

Argument
`{div}: 5` `int`
 Number of subdivisions.

Argument
`(fun)` `function(fraction) => (fraction, fraction)`
 For each location, returns the top and bottom bounds.

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

Horizontal segmentation as (anchor, width).

— Argument —	<code>(div): 5</code>	int
Number of subdivisions.		
— Argument —	<code>(flush): center</code>	alignment
Relative horizontal alignment of the anchor.		
— Argument —	<code>(fun)</code>	function(fraction) => (fraction, fraction)
For each location, returns the position of the anchor and the width.		

VIII.4 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.take-it-or-leave-it</code>
<code>#bisect.fits-inside</code>	<code>#bisect.has-text</code>	

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

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

Inspired by wrap-it's implementation (see: [_rewrap in `github:ntjess/wrap-it`](#))

```

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.)

#bisection.dispatch(({fits-inside}, {cfg}))[ct]

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 fits-inside above).

Argument —

(cfg)

dictionary

Extra configuration options.

#bisection.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)

(width: length, height: length)

Container size.

Argument —

(ct)

content

Content to split.

— Argument —

`(size): none` (width: length, height: length)

Parent container size.

— Argument —

`(cfg): (:)` dictionary

Configuration options.

- `list-markers: (..content,)`, default value ([•], [‣], [-], [•], [‣], [-]). If you change the markers of list, put the new value in the parameters so that lists are correctly split.
- `enum-numbering: (..str,)`, default value ("1.", "1.", "1.", "1.", "1.", "1."). If you change the numbering style of enum, put the new style in the parameters so that enums are correctly split.

#bisect.fits-inside({dims}, {size}: none)[ct] → bool

Tests if content fits inside a box.

WARNING: 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, 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 `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]

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` are splittable, the rest are treated as non-splittable.

— Argument —

`(ct)``content`

Content to split.

— Argument —

`(split-dispatch)``function`

Recursively passed around (see `split-dispatch` below).

— Argument —

`(fits-inside)``function`

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

— Argument —

`(cfg)``dictionary`

Extra configuration options.

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

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 split-dispatch below).

Argument –

`(fits-inside)`

function

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

Argument –

`(cfg)`

dictionary

Extra configuration options.

`#bisect.has-children({split-dispatch}, {fits-inside}, {cfg})[ct]`

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 split-dispatch below).

Argument –

`(fits-inside)`

function

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

Argument –

`(cfg)`

dictionary

Extra configuration options.

`#bisect.has-text({split-dispatch}, {fits-inside}, {cfg})[ct]`

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.

Argument –

`(ct)`

content

Content to split.

Argument –

`(split-dispatch)`

function

Recursively passed around (see split-dispatch below).

— Argument —

`(fits-inside)`

function

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

— Argument —

`(cfg)`

dictionary

Extra configuration options.

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

Split an enum.item. 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 `split-dispatch` below).

— Argument —

`(fits-inside)`

function

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

— Argument —

`(cfg)`

dictionary

Extra configuration options.

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

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 `split-dispatch` below).

— Argument —

`(fits-inside)`

function

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

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 `fits-inside` above).

VIII.5 Threading

#`threading.smart-fill-boxes`

#`threading.smart-fill-boxes({avoid}: (), {boxes}: (), {extend}: 1em, {size}: none) [body] → (..content,)`

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,)

Obstacles to avoid. A list of (x: length, y: length, width: length, height: length).

Argument —

`(boxes): ()`

(..block,)

Boxes to fill. A list of (x: length, y: length, width: length, height: length, bound: block).

bound is the upper limit of how much to stretch the container, i.e. also (x: length, y: length, width: length, height: length).

Argument —

`(extend): 1em`

length

How much the baseline can extend downwards (within the limits of bounds).

Argument —

`{size}: none`

`(width: length, height: length)`

Dimensions of the container as given by layout.

Part IX

Modularity (WIP)

Because meander is cleanly split into three algorithms (content bisection, page tiling, text threading), there are plans to provide

- additional configuration options for each of those steps
- the ability to replace entirely an algorithm by either a variant, or a user-provided alternative that follows the same signature.

Part X

About

X.1 Related works

In Typst:

- `WRAP-IT` has essentially the same output as `MEANDER` with at 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 uses `MANTYS` and `TIDY`.

X.3 Acknowledgements

`MEANDER` started out as an idea in the Typst Discord server; thank you to everyone who gave input and encouragements.