MEANDER

User guide

v0.4.0 2026-01-21 MIT

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

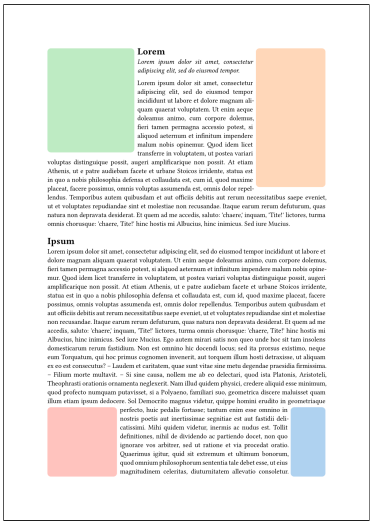
Neven Villani
neven@crans.org

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

Though very different in its modeling, MEANDER can be seen as a Typst alternative to LaTeX'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`
- `0.3.1` (`latest`)
- `0.3.0`
- `0.2.5`
- `0.2.4`
- ...

To migrate existing code, please consult Section I.6 and Section I.7

Table of Contents

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

Part I

Quick start

Import the latest version of MEANDER with:

```
#import "@preview/meander:0.3.1"
```

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.

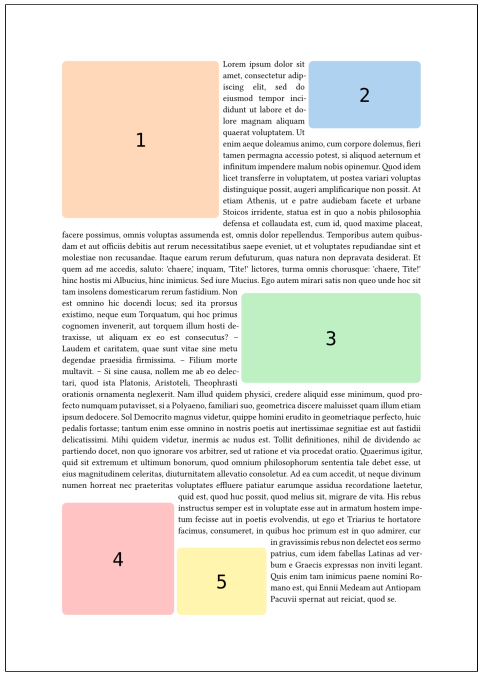
> If you find a style discrepancy, make sure to file it as a bug report, if it is not already part of the known limitations.

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)
  ]
})
```

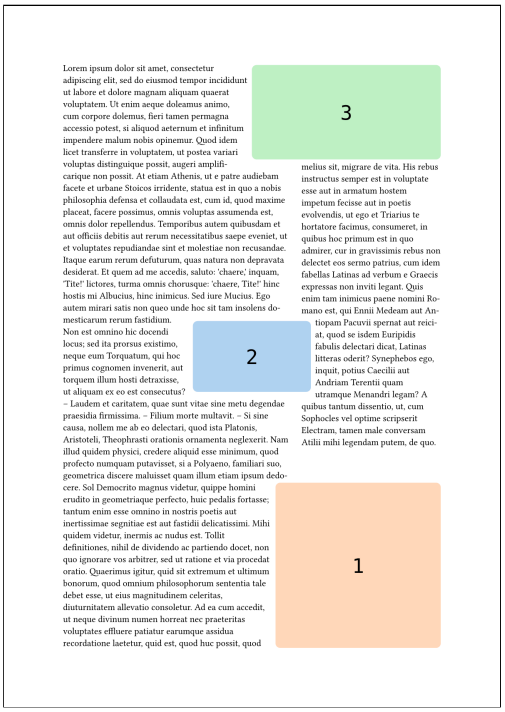


> Technically, MEANDER can only handle rectangular obstacles. However, thanks to this ability to wrap around an arbitrary number of obstacles, we can approximate a non-rectangular obstacle using sereval rectangles. See concrete applications and techniques for defining these rectangular tilings in Section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



I.4 Anatomy of an invocation

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

```
#meander.reflow({
  import meander: *
  // pre-layout options

  // layout and dynamic options

  // post-layout options
})
```

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 instanciated 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			
`#meander.regions({..})`	(command)	`opt.debug.pre-thread()`	(pre)
`debug: true`	(parameter)	`opt.debug.post-thread()`	(pre)
Overflow			
`overflow: false`	(parameter)	`opt.overflow.alert()`	(post)
`overflow: true`	(parameter)	`opt.overflow.ignore()`	(post)

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

I.7 0.3.x Migration Guide

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

old

```
placed(
  query.position(..),
  ..
)
container(
  align: query.position(..),
  width: query.width(..),
  height: query.height(..),
)
```

new

```
callback(
  pos1: query.position(..),
  align2: query.position(..),
  width2: query.width(..),
  heigth2: query.height(..),
  env => {
    placed(
      env.pos1,
      ..
    )
    container(
      align: env.align2,
      width: env.width2,
      height: env.height2,
    )
  },
)
```

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

Part II

Understanding the algorithm

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

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

II.1 Debugging

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

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

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

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.

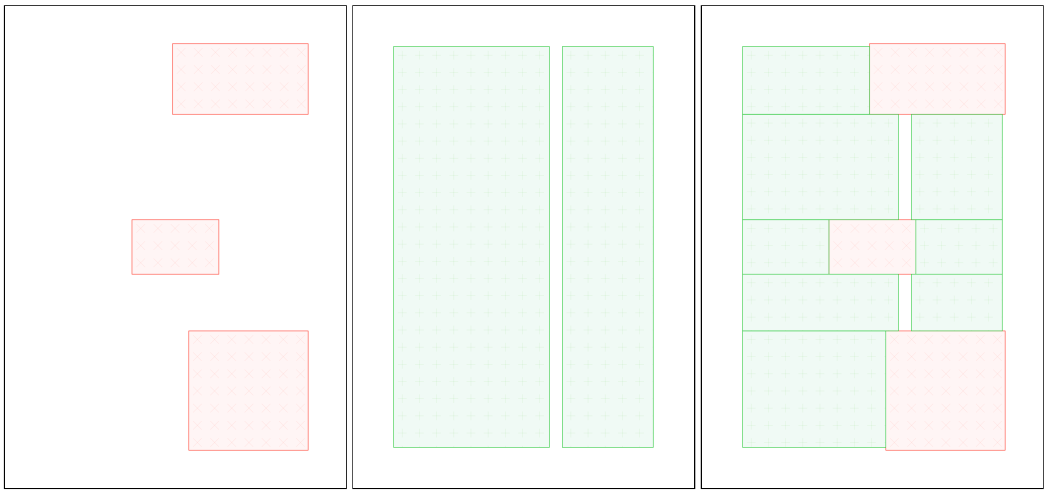


Figure 1: Left to right: the forbidden, allowed, and combined regions.

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 adip-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.

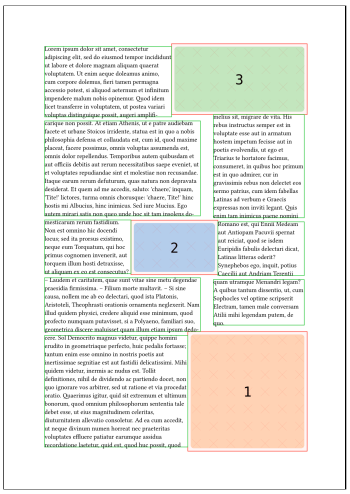


Figure 2: Debug view of the final output via `opt.debug.post-thread()`

> 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.

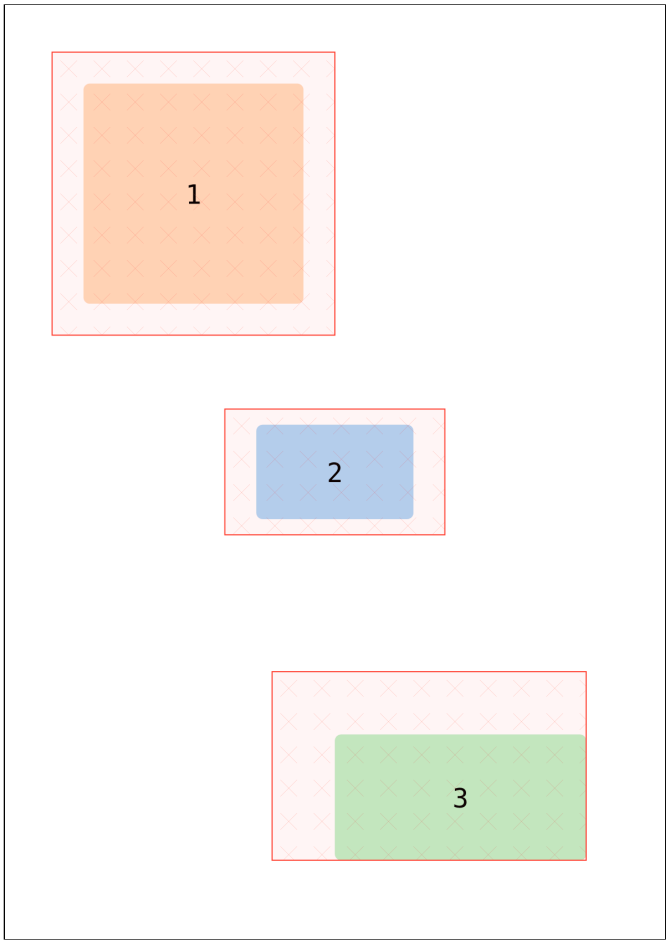
This is not a mistake. The reality is that these statements are only incompatible if we assume that 1 image = 1 obstacle. We call "contouring functions" the utilities that allow splitting one image into multiple obstacles to approximate an arbitrary shape.

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.

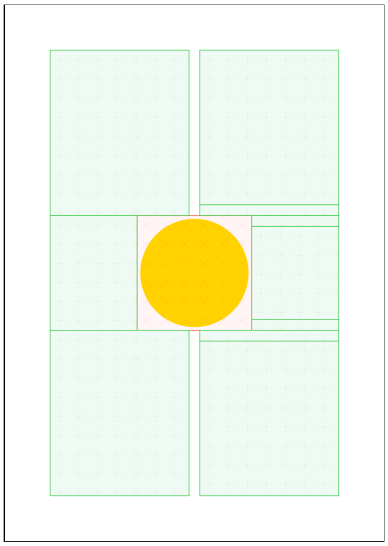
```
#meander.reflow({
  import meander: *
  opt.debug.pre-thread()
  placed(
    top + left,
    boundary:
      contour.margin(1cm),
    my-img-1,
  )
  placed(
    center + horizon,
    boundary:
      contour.margin(
        5mm,
        x: 1cm,
      ),
    my-img-2,
  )
  placed(
    bottom + right,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

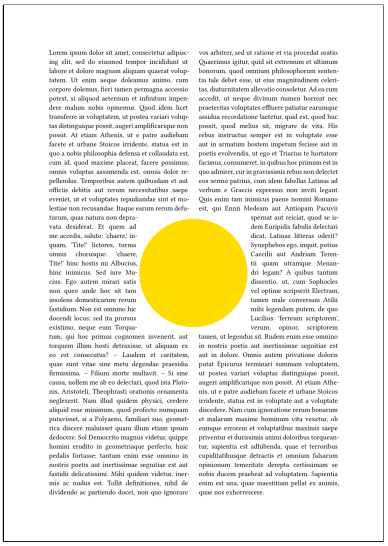


III.2 Boundaries as equations

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)
  ]
})
```

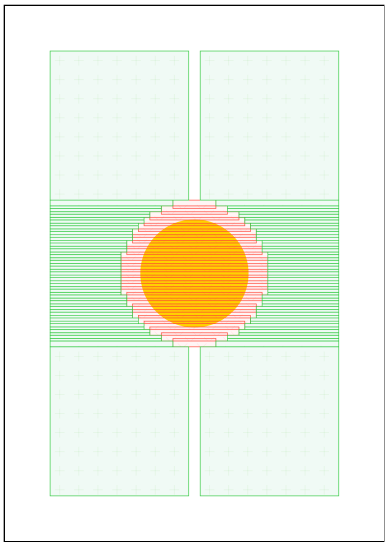




MEANDER sees all obstacles as rectangular, so the circle leaves a big ugly square hole in the page. Fortunately the desired circular shape is easy to describe in equations, and we can do so using the function `#contour.grid`, which takes as input a 2D formula normalized to $[0, 1]$, i.e. a function from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

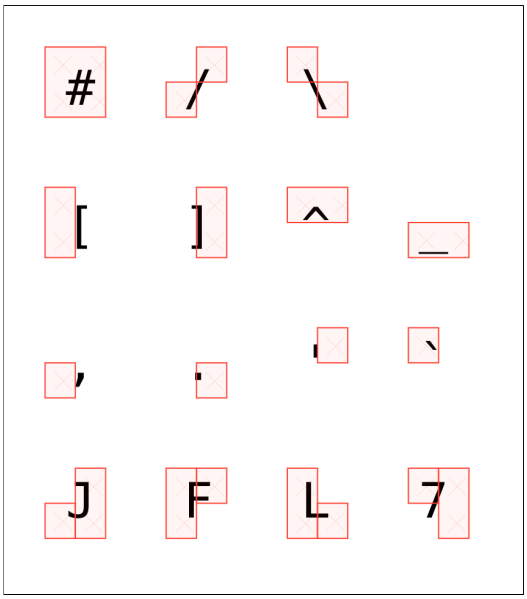
The contouring function #contour.ascii-art takes as input a string or raw code and uses it to draw the shape of the image. The characters that can occur are:

```
#meander.reflow({
  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.

Wrong

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

Correct

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

})
```

Correct

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

    #lorem(600)
  ]

})
```

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.

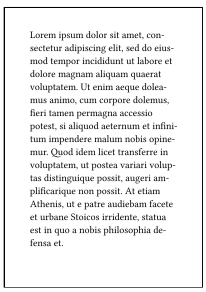
Analogously, if you wish to change the spacing between lines, use either a `#set par(leading: 1em)` outside of `#meander.reflow`, or pass `⟨leading⟩: 1em` as a parameter to `#content`.

Wrong

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

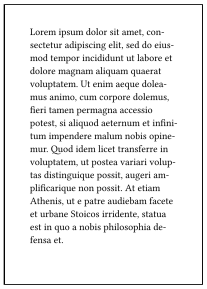
Correct

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



Correct

```
#meander.reflow({
  // ...
  content(size: 30pt)[

    #lorem(80)
  ]
})
```

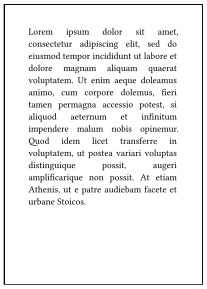


IV.3 Hyphenation and language

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`.

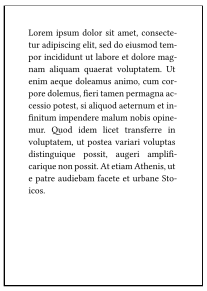
Wrong

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



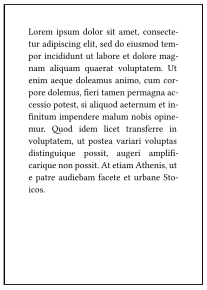
Correct

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



Correct

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



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

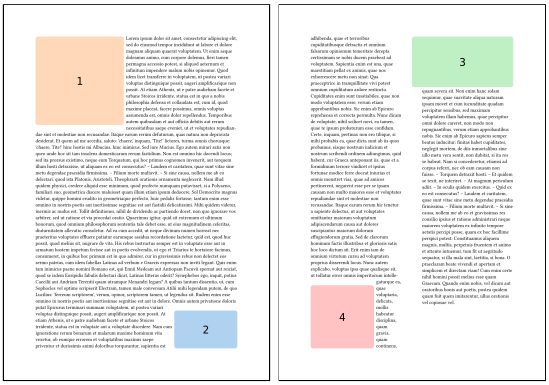
V.1 Pagebreak

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)]
})
```

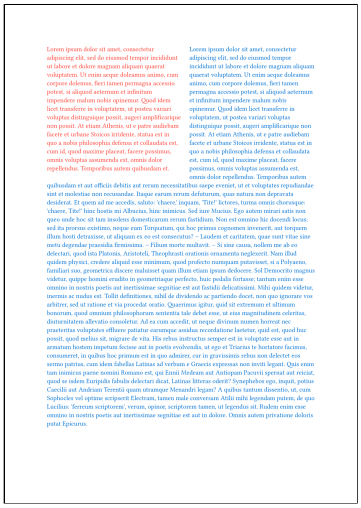


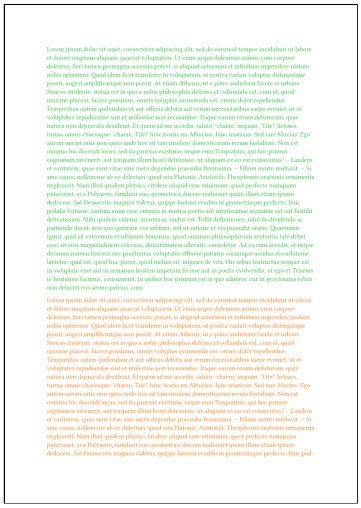
Notice: text from a 1-column layout overflows into a 2-column layout.

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)]

})
```



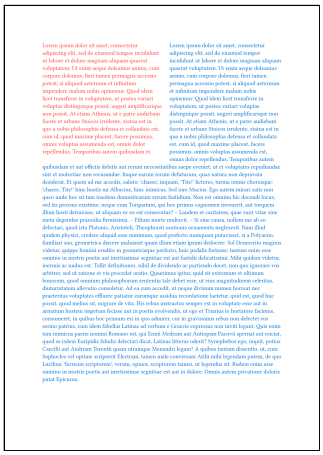


V.3 Colfill

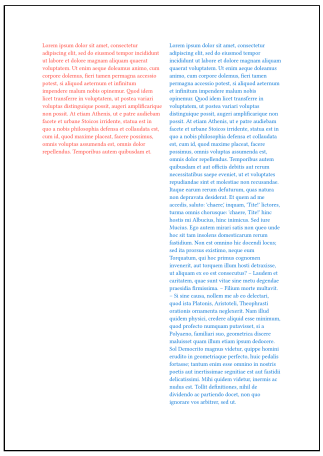
Contrary to `#colbreak` which breaks to the next container, `#colfill` fills the current container, *then* breaks to the next container. There is sometimes a subtle difference

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

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



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



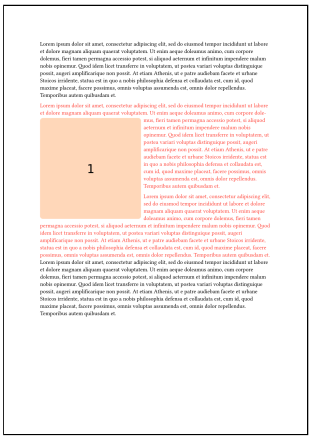
> Recall that filled containers count as obstacles for future containers, so there is a difference between dropping containers and filling them with nothing.

V.4 Placement

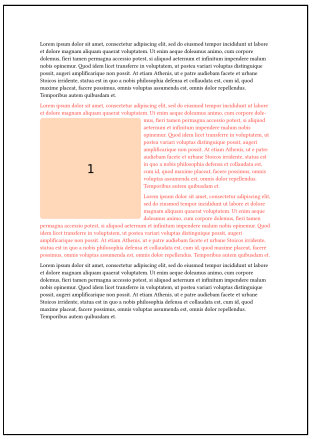
Placement options control how a `#meander.reflow` invocation is visible by and sees other content. By default, MEANDER will automatically insert invisible boxes of the correct height to emulate the text's placement. If this does not behave exactly as you want, this section details the alternatives available.

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

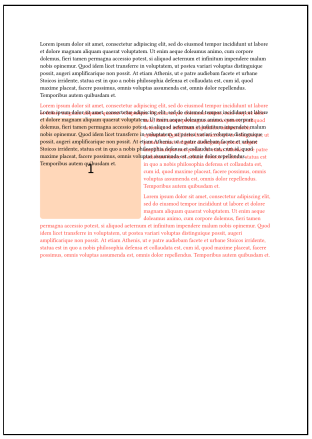
If the space computed by MEANDER is wrong, you can disable it entirely by adding `opt.placement.phantom()` (see below: right), or adjust the margins such as with `opt.placement.spacing(below: 0.65em)` (see below: left).



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



Default heuristic, with artificial spacing.

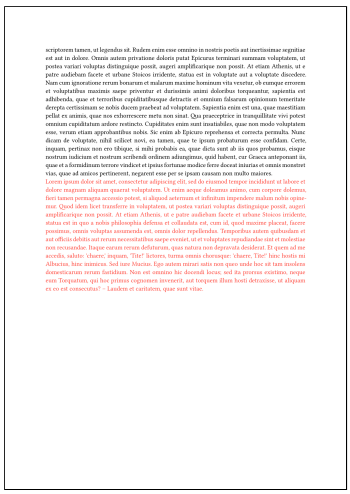


No artificial spacing.

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.placement.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()
})
```

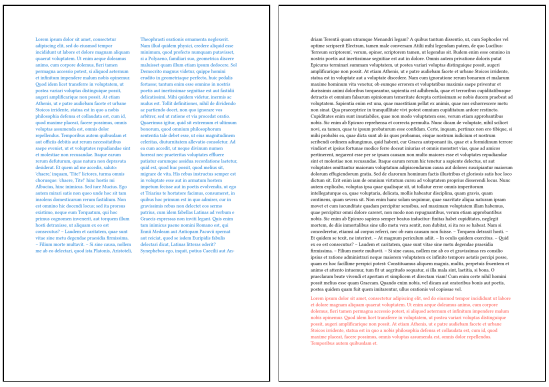
`opt.overflow.ignore()`

```
#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.

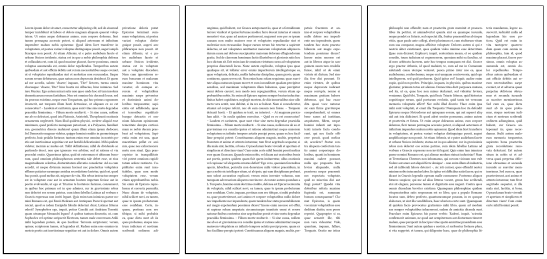
```
#meander.reflow({
  import meander: *
  container(
    height: 70%, width: 48%,
    style: (text-fill: blue),
  )
  container(
    align: right,
    height: 70%, width: 48%,
    style: (text-fill: blue),
  )
  content[#lorem(1000)]
  opt.overflow.pagebreak()
})
#text(fill: red)[#lorem(100)]
```



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)→content` passed to `opt.overflow.custom(_ => {})` can serve as handler,

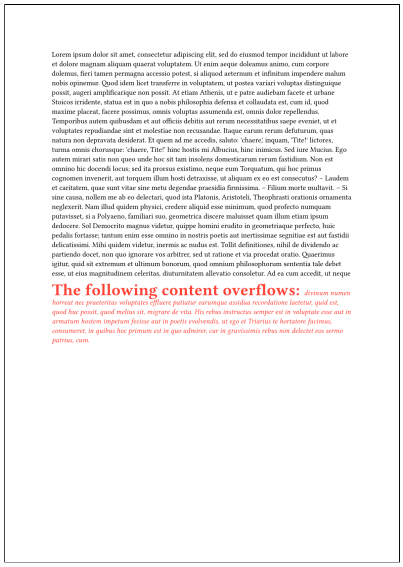
including another invocation of `#meander.reflow`. This function will be given as input an object of type `overflow`, which is concretely a dictionary with fields:

- `⟨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.

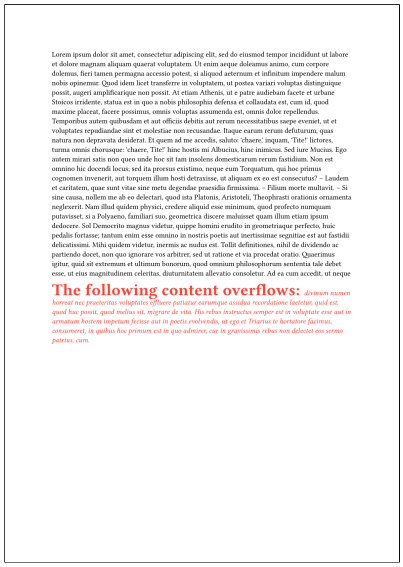
For example here is a handler that adds a header and some styling options to the text that overflows:

```
#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}_
  ])
})
```



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: (<B>, <C>))`

VI.1 Locally invisible obstacles

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

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

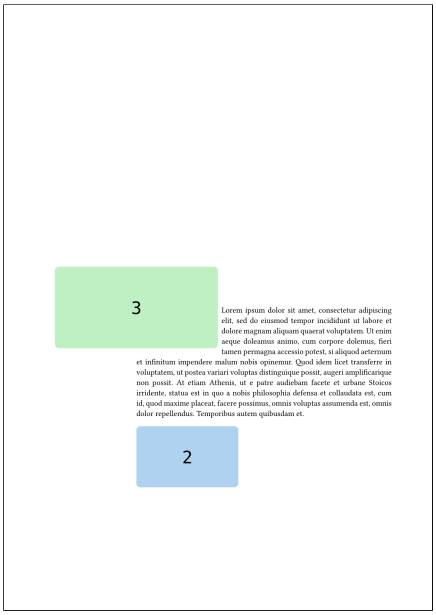
This is already doable globally by setting ⟨boundary⟩ to #contour.phantom. The innovation of ⟨invisible⟩ is that this can be done on a per-container basis.

VI.2 Callbacks and queries

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

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

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

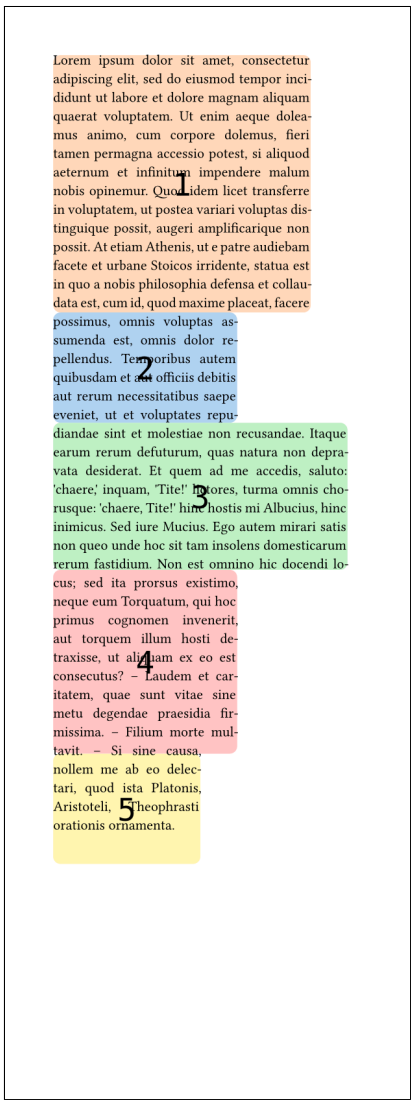


In this example, after giving an absolute position to one image, we create a container with position and dimensions relative to the image, and place another image immediately after the container ends.

VI.3 A nontrivial example

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

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

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

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

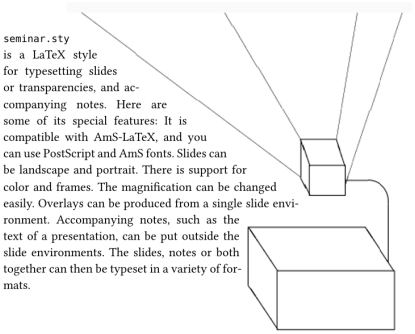


Part VII

Showcase

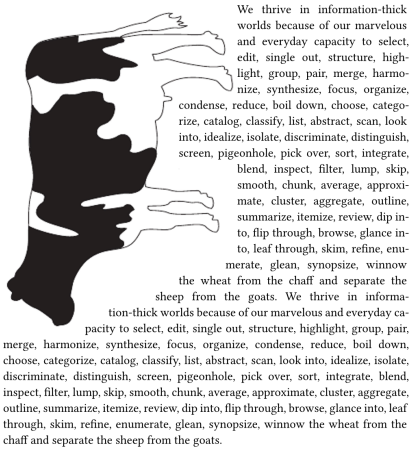
A selection of nontrivial examples of what is feasible, inspired mostly by requests on issue #5181. You can find the source code for these on the repository.

VII.1 Side illustrations



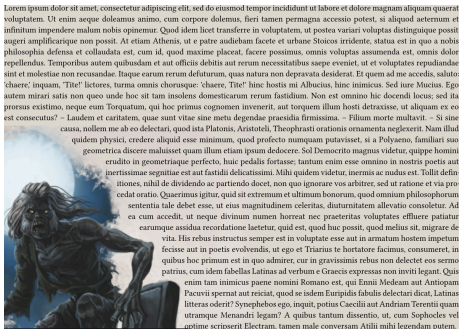
`tests/examples/typst-5181-a/test.typ`

Motivated by `github:typst/typst` #5181 (a)



`tests/examples/cow/test.typ`

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



`tests/examples/typst-5181-b/test.typ`

Motivated by `github:typst/typst` #5181 (b)

VII.2 Paragraph packing

Talmudifier Test Page

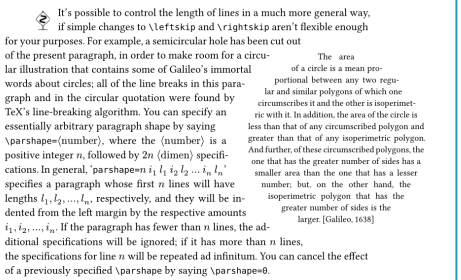
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis vehicula ligula at est bibendum, in eleifend erat dictum. Etiam sit amet tellus id ex ullamcorper faucibus. Suspendisse sed elit vel neque convallis iaculis id in urna. Sed tincidunt varius ipsum at scelerisque. Phasellus lacus lectus, sodales sit amet orci in, rutrum malesuada 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 eget consequat. R. Seth: Blah blah blah blah. As it is written: "Aecenas lacinia nisi diam, vel pulvinar metus aliquet ut. Sed non lorem quis dui ultrices volutpat quis at diam."=Lorem&1:23 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. Suspendisse hendrerit eros sit amet tempor pulvinar. Duis velit mauris, facilisis ut tincidunt sed, pharetra eu libero. Aenean lobortis tincidunt nisi. Praesent metus lacus, tristique sed porta non, tempus id quam. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; In eu porta velit, quis pellentesque elit. Quisque vehicula massa sit amet justo rhoncus auctor.

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. Suspendisse hendrerit eros sit amet tempor pulvinar. Duis velit mauris, facilisis ut tincidunt sed, pharetra eu libero. Aenean lobortis tincidunt nisi. Praesent metus lacus, tristique sed porta non, tempus id quam. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; In eu porta velit, quis pellentesque elit. Quisque vehicula massa sit amet justo rhoncus auctor.

Aenean sed dolor suscipit, dignissim nisl at, blandit eros. Pellentesque scelerisque viverra ligula, vel congue lorem semper non. Nulla nec convallis neque. Donec ut leo velit. Donec felis odio, tincidunt non vestibulum ut, consectetur eget lectus. Donec varius finibus scelerisque. Aliquam nisl odio, sollicitudin eget sem non, tincidunt tempor felis. Curabitur et ullamcorper lacus. Cras eu ante quis arcu dictum ultrices. Proin vel ante quis elit sollicitudin auctor. Nullam ultricies tempor neque, varius scelerisque neque. Nam et arcu ut odio maximus tempor et sit amet elit. Morbi fermentum dapibus elementum. Proin id metus ipsum. Aenean posuere nunc quis lacus varius, eget molestie mauris accumsan. | Fusce sapien ipsum, cursus a tincidunt vel, dignissim eu mi. Morbi id velit ac turpis ullamcorper lacinia. | Cras bibendum tellus 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. In ac velit maximus, elementum ex et, blandit massa. Aliquam vehicula at neque sit amet ultrices. 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 erat pulvinar elit facilisis, ac venenatis leo tincidunt. Quisque eu lorem tortor. Quisque nec porttitor elit. Ut finibus ullamcorper odio, in porttitor lorem suscipit ut. Vestibulum vitae nisl blandit, commodo odio vitae, dictum nunc. Suspendisse pharetra lorem vitae ex tincidunt ornare. Maecenas efficitur tristique libero, eget commodo urna. Pellentesque libero sem, interdum ut nibh interdum, consequat elementum magna."=Ipsum&69:420 Aliquam facilisis vel turpis eu semper. Donec eget purus lectus. -> 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 dui, at rhoncus nibh porttitor quis. Donec dui ante, semper non quam at, accumsan volutpat leo. Maecenas magna risus, finibus sit amet felis ut, vulputate euismod nunc.

`tests/examples/talmudifier/test.typ`
From `github:subalterngames/talmudifier`
Motivated by `github:typst/typst` #5181 (c)

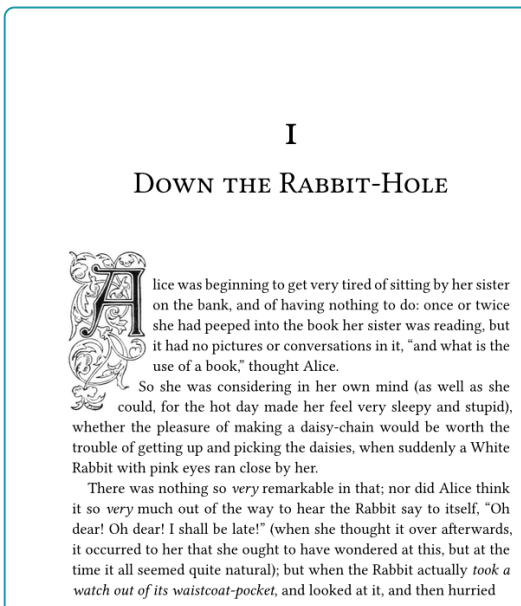


It's possible to control the length of lines in a much more general way, if simple changes to `\leftskip` and `\rightskip` aren't flexible enough for your purposes. For example, a semicircular hole has been cut out of the present paragraph, in order to make room for a circular illustration that contains some of Galileo's immortal words about circles; all of the line breaks in this paragraph and in the circular quotation were found by TeX's line-breaking algorithm. You can specify an essentially arbitrary paragraph shape by saying `\parshape=`⟨number⟩, where the ⟨number⟩ is a positive integer n, followed by $2n$ ⟨dimen⟩ specifications. In general, '`parshape=`$n\ i_1\ l_1\ i_2\ l_2 \ldots\ i_n\ l_n$' specifies a paragraph whose first n lines will have lengths $l_1, l_2, \ldots, l_n$, respectively, and they will be indented from the left margin by the respective amounts $i_1, i_2, \ldots, i_n$. If the paragraph has fewer than n lines, the additional specifications will be ignored; if it has more than n lines, the specifications for line n will be repeated ad infinitum. You can cancel the effect of a previously specified `\parshape` by saying `\parshape=0`.

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

`tests/examples/area-of-a-circle/test.typ`
Inspired by TeXBook, asked on Stack Exchange

VII.3 Drop caps



I

DOWN THE RABBIT-HOLE

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

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

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

`tests/examples/drop-caps/test.typ`

Courtesy of @wade-cheng on #18

Part VIII
Public API

These are the user-facing functions of MEANDER.

VIII.1 Elements

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

`#callback`	`#container`	`#placed`
`#colbreak`	`#content`	
`#colfill`	`#pagebreak`	

↑ Since 0.4.0

`#callback(..⟨env⟩, ⟨fun⟩)`

Callback to generate elements that depend on prior layout information.

Argument

`..⟨env⟩` `dictionary`

Elements from the `query` module assigned to names. See Section VIII.4 for a list of usable values.

Argument

`⟨fun⟩` `function(dictionary => list(elem))`

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

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

`#colbreak` → `flowing`

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

`#colfill` → `flowing`

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

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

Core function to create a container.

Argument

⟨align⟩: top + left — align | location

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

Argument

⟨dx⟩: 0% + 0pt — relative

Horizontal displacement.

Argument

⟨dy⟩: 0% + 0pt — relative

Vertical displacement.

Argument

⟨width⟩: 100% — relative

Width of the container.

Argument

⟨height⟩: 100% — relative

Height of the container.

Argument

⟨style⟩: (:) — dictionary

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

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

Argument

⟨margin⟩: 5mm — length | dictionary

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

Argument

`⟨invisible⟩: ()` `label` | `array(label)`

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

Argument

`⟨tags⟩: ()` `label` | `array(label)`

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

`#content(⟨size⟩: auto, ⟨lang⟩: auto, ⟨hyphenate⟩: auto, ⟨leading⟩: auto)[data]` → `flowing`

Core function to add flowing content.

Argument

`⟨data⟩` `content`

Inner content.

Argument

`⟨size⟩: auto` `length`

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

Argument

`⟨lang⟩: auto` `str`

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

Argument

`⟨hyphenate⟩: auto` `bool`

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

Argument

`⟨leading⟩: auto` `length`

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

`#pagebreak` → `elem`

Continue layout to next page.

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

Core function to create an obstacle.

Argument

⟨align⟩ `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.reflow(⟨seq⟩)` → `content`

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

Argument

`⟨seq⟩` `array(elem)`

See Section VIII.1 for how to format this content.

VIII.3 Contouring

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

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

`#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.parent-size         #query.width
```

#query.height(⟨tag⟩, ⟨transform⟩: 100%) → query(length)

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

Argument

⟨tag⟩ `label`

Reference a previous element by its tag.

Argument

⟨transform⟩: 100% `ratio` | `function`

Apply some post-processing transformation to the value.

#query.parent-size(⟨clip-to-page⟩: true, ⟨transform⟩: 100%) → query(size)

Returns the size of the container.

Argument

`⟨clip-to-page⟩: true` `bool`

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

Argument

`⟨transform⟩: 100%` `ratio` | `function`

Apply a transformation to the value.

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

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

Argument

`⟨tag⟩` `label`

Reference a previous element by its tag.

Argument

`⟨at⟩: center` `align`

Anchor point relative to the box in question.

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

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

Argument

`⟨tag⟩` `label`

Reference a previous element by its tag.

Argument

`⟨transform⟩: 100%` `ratio` | `function`

Apply some post-processing transformation to the value.

VIII.5 Options

Configuring the behavior of `#meander.reflow`.

VIII.5.1 Pre-layout options

These come before all elements.

Debug settings

Visualizing containers and obstacle boundaries.

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

#opt.debug.minimal → `option`

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

#opt.debug.post-thread → `option`

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

#opt.debug.pre-thread → `option`

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

#opt.debug.release → `option`

No visible effect.

Placement settings

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

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

#opt.placement.phantom → `option`

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

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

Controls the margins before and after the MEANDER layout.

Argument

`(above): auto` `auto` | `length`

Margin above the layout

Argument

`(below): auto` `auto` | `length`

Margin below the layout

Argument

`(both): auto` `auto` | `length`

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

VIII.5.2 Dynamic options

These modify parameters on the fly.

None yet

VIII.5.3 Post-layout options

These come after all elements.

Overflow settings

What happens to content that doesn't fit.

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

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

```
#import "@preview/meander:0.3.1": internals.geometry
```

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

Part IX
Internal module details

IX.1 Utils

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

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

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

Argument

`⟨data⟩` `content`

Text to style.

Argument

`⟨size⟩: auto` `length`

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

Argument

`⟨lang⟩: auto` `str`

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

Argument

`⟨hyphenate⟩: auto` `bool`

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

Argument

`⟨leading⟩: auto` `length`

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

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

Interprets a single element as a singleton.

Argument

`⟨t⟩` `T | array(T)`

Element or array

IX.2 Types

```
#types.query
```

#types.query
Standalone type of queries

#opt

Option marker
- `pre`: options that come before the layout
- `post`: options that come after the layout

#flow

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

#elt

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

IX.3 Geometry

```
#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⟩ — `align` | `dictionary`

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

Argument
`(dx): 0pt` `relative`
Horizontal displacement.

Argument
`(dy): 0pt` `relative`
Vertical displacement.

Argument
`(width): 0pt` `relative`
Object width.

Argument
`(height): 0pt` `relative`
Object height.

Argument
`(anchor): auto` `align` | `auto`
Anchor point.

`#geometry.apply-transform((value), (transform): 100%) → any`
Apply a transformation in the form of either a scaling or a function.

Argument
`(value)` `any`
Value to transform. Any type as long as it supports multiplication by a scalar.

Argument
`(transform): 100%` `function` | `ratio`
Scaling to apply, as either a ratio or a function.

`#geometry.between((a), (b), (c)) → bool`
Testing `a <= b <= c`, helps only computing `b` once.

Argument
`(a)` `length`
Lower bound.

Argument
`⟨b⟩` `length`

Tested value.

Argument
`⟨c⟩` `length`

Upper bound. Asserted to be `>= 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.frac-rect(⟨frac⟩, ⟨abs⟩, ..⟨style⟩)` → `block(length)`

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

Argument
`⟨frac⟩` `block(fraction)`

Child dimensions as fractions.

Argument
`⟨abs⟩` `block(length)`

Parent dimensions as absolute lengths.

Argument
`..⟨style⟩`

Currently ignored.

`#geometry.in-region(⟨region⟩, ⟨alignment⟩)` → `(x: length, y: length)`

Resolves an anchor point relative to a region.

Argument

`(region)` `block`

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

Argument

`(alignment)` `align`

An alignment within the block.

`#geometry.intersects((i1), (i2), (tolerance): 0pt)`

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): 0pt` `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.

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

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

`#tiling.add-self-margin(⟨elem⟩) → elem`

Applies an element's margin to itself.

Argument

`⟨elem⟩` `elem`

Inner element.

`#tiling.blocks-of-container(⟨data⟩, ⟨obj⟩) → blocks`

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

Argument

`⟨data⟩` `opaque`

Internal state.

Argument

`⟨obj⟩` `elem`

Container to segment.

context

`#tiling.blocks-of-placed(⟨data⟩, ⟨obj⟩) → blocks`

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

Argument

`⟨data⟩` `opaque`

Internal state.

Argument

`(obj)` `elem`

Object to measure, pad, and place.

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

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

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

Argument

`(size): none` `size`

Dimensions of the page

Argument

`(page-offset): (x: 0pt, y: 0pt)` `size`

Optional nonzero offset on the top left corner

Argument

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

Elements to dispense in order

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

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

`#tiling.get-page-offset → (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.

Argument

⟨data⟩ opaque

Internal state.

Argument

⟨elem⟩ elem

Element to register.

#tiling.separate(⟨seq⟩) → (pages: array(elem), flow: array(elem), opts: dictionary)

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

```
#separate({
  // This is an obstacle
  placed(top + left, box(width: 50pt, height: 50pt))
  // This is a container
  container(height: 50%)
  // This is flowing content
  content[#lorem(50)]
})
```

Argument

`(seq)` `array(elem)`

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

IX.5 Bisection

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

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

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

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

```
#let content = box(stroke: red)[Initial]
#let (inner, rebuild) = default-rebuild(
  content, "body",
)

Content: #content \
Inner: #inner \
Rebuild: #rebuild("foo")
```

```
#let content = [*_Initial_*]
#let (inner, rebuild) = default-rebuild(
  content, "body",
)

Content: #content \
Inner: #inner \
Rebuild: #rebuild("foo")
```

```
#let content = [a:b]
#let (inner, rebuild) = default-rebuild(
  content, "children",
)

Content: #content \
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.

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

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

- `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/hypher. This manual is built using MANTYS and TIDY.

X.3 Acknowledgements

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

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

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