

# MEANDER

## User guide

v0.2.4

2025-10-29

MIT

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

NEVEN VILLANI

✉ neven@crans.org

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

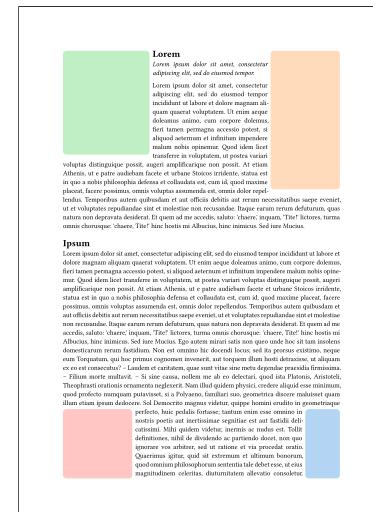
Though very different in its modeling, **MEANDER** can be seen as a Typst alternative to L<sup>A</sup>T<sub>E</sub>X's `wrapfig` and `parshape`, effectively enabling the same kinds of outputs.

### 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](#)
- [0.2.4 \(latest\)](#)
- [0.2.3](#)
- [0.2.2](#)
- [0.2.1](#)
- [0.2.0](#)
- [0.1.0](#)



# Table of Contents

<b>Quick start .....</b>	<b>3</b>
I.1 A simple example .....	3
I.2 Multiple obstacles .....	4
I.3 Columns .....	5
I.4 Going further .....	5
<b>Understanding the algorithm .....</b>	<b>7</b>
II.1 Page tiling .....	7
II.2 Content bisection .....	8
II.3 Threading .....	9
<b>Contouring .....</b>	<b>10</b>
III.1 Margins .....	10
III.2 Boundaries as equations .....	11
III.3 Boundaries as layers .....	13
III.3.1 Horizontal rectangles .....	13
III.3.2 Vertical rectangles .....	14
III.4 Autocontouring .....	15
III.5 More to come .....	16
<b>Styling .....</b>	<b>17</b>
IV.1 Paragraph justification .....	17
IV.2 Font size and leading .....	18
IV.3 Hyphenation and language .....	19
IV.4 Styling containers .....	20
<b>Multi-page setups .....</b>	<b>21</b>
V.1 Pagebreak .....	21
V.2 Colbreak .....	21
V.3 Colfill .....	22
V.4 Placement .....	23
V.4.1 Default .....	23
V.4.2 Inline .....	24
V.4.3 Full page .....	24
V.4.4 Use-case .....	24
V.5 Overflow .....	25
V.5.1 No overflow .....	25
V.5.2 Predefined layouts .....	26
V.5.3 Custom layouts .....	27
<b>Inter-element interaction .....</b>	<b>30</b>
VI.1 Locally invisible obstacles .....	30
VI.2 Position and length queries .....	31
VI.3 A nontrivial example .....	32
<b>Showcase .....</b>	<b>33</b>
<b>Public API .....</b>	<b>35</b>
VIII.1 Elements .....	35
VIII.2 Layouts .....	38
VIII.3 Contouring .....	40
VIII.4 Queries .....	43
VIII.5 Std .....	44
VIII.6 Public internals <sup>(1)</sup> .....	44
<b>Internal module details .....</b>	<b>45</b>
IX.1 Utils .....	45
IX.2 Geometry .....	46
IX.3 Tiling .....	49
IX.4 Bisection .....	52
IX.5 Threading .....	59
<b>Modularity (WIP) .....</b>	<b>60</b>
<b>About .....</b>	<b>61</b>
XI.1 Related works .....	61
XI.2 Dependencies .....	61
XI.3 Acknowledgements .....	61

Chapters that are highlighted<sup>(1)</sup>  
have received major updates in  
the latest version 0.2.4

# Part I

## Quick start

Import the latest version of [MEANDER](#) with:

```
#import "@preview/meander:0.2.4"
```

**Do not `#import "@preview/meander:0.2.4"`:** \* globally, it would shadow important functions.

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)
    ]
})
```

**L**orem ipsum dolor sit amet, consecutetur adipiscing elit, sed ut eiusmod tempor incididunt ut labore et dolore magna aliquam quaerat volupatem. Ut enim aequo dolamus animo, cum corpore doloemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis oportineat. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguere possit, augeri amplificare non possit. At ipsum dolor sit amet, consecutetur adipiscing elit, sed ut eiusmod tempor incididunt ut labore et dolore magna aliquam quaerat volupatem. Ut enim aequo dolamus animo, cum corpore doloemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis oportineat. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguere possit, augeri amplificare non possit. At etiam Athenis, ut e patre audiebam facete et urbe Stocis irridente, statua est in ipso a nobis philosophia defensa et collaudata est, cum id maxime placet; facere possumus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aucti officiis debitis aut rerum necessitatibus saepe evenit, ut et voluptates repudiandae sint et molestiae non recusande. Itaque earum rerum delectum, sicut nam non depravata desiderat. Et quem ad me accedit, saluto: 'haec, inquam, Titè' lectores, turna omnis chorusc' chare', Titè' hinc hostis mihi Albaicus, hinc inimicus Sed iure Mucius. Ego autem miatis sunt non quo inde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic doctendi locis sed ita prout sua existimatio, neque eum Tropianum, qui hoc primogenium inventerit, aut torquem illam via metu deridat, praesidia nostra. — Filium multe havit, et caritatem quae virtutis via metu deridat, ita pietate, ita misericordia, ita benevolencia, ita clementia neglegit. Nam filii quidam physice, credere aliiquid esse minimum, quod profecto numpum putavisset, si a Polyeno familiari suo, geometrica districte maluissest quam ilium ipsum indicaret Sol Decenarius magnum videtur, quippe homini erudit in geometria perfecto, hinc portulis fastidio: tantum enim esse omnino in nostris poetis aut incertimis segnitate est aut fidelasti dictum: Mibi quidem videtur, inermis ac nudus est. Tollit definitio, nihil de dividiendo ac partendo docet, non quo ignorare vobis sit, sed ut.

Loem ipsum dolor sit amet, consecutetur adipiscing elit, sed ut eiusmod tempor incididunt ut labore et dolore magna aliquam quaerat volupatem. Ut enim aequo dolamus animo, cum corpore doloemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis oportineat. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguere possit, augeri amplificare non possit. At etiam Athenis, ut e patre audiebam facete et urbe Stocis irridente, statua est in ipso a nobis philosophia defensa et collaudata est, cum id maxime placet; facere possumus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aucti officiis debitis aut rerum necessitatibus saepe evenit, ut et voluptates repudiandae sint et molestiae non recusande. Itaque earum rerum delectum, sicut nam non depravata desiderat. Et quem ad me accedit, saluto: 'haec, inquam, Titè' lectores, turna omnis chorusc' chare', inquam, Titè' hinc hostis mihi Albaicus, hinc inimicus Sed iure Mucius. Ego autem miatis sunt non quo inde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic doctendi locis sed ita prout sua existimatio, neque eum Tropianum, qui hoc primogenium inventerit, aut torquem illam via metu deridat, praesidia nostra. — Filium multe havit, et caritatem quae virtutis via metu deridat, ita pietate, ita misericordia, ita benevolencia, ita clementia neglegit. Nam filii quidam physice, credere aliiquid esse minimum, quod profecto numpum putavisset, si a Polyeno familiari suo, geometrica districte maluissest quam ilium ipsum indicaret Sol Decenarius magnum videtur, quippe homini erudit in geometria perfecto, hinc portulis fastidio: tantum enim esse omnino in nostris poetis aut incertimis segnitate est aut fidelasti dictum: Mibi quidem videtur, inermis ac nudus est. Tollit definitio, nihil de dividiendo ac partendo docet, non quo ignorare vobis sit, sed ut.

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

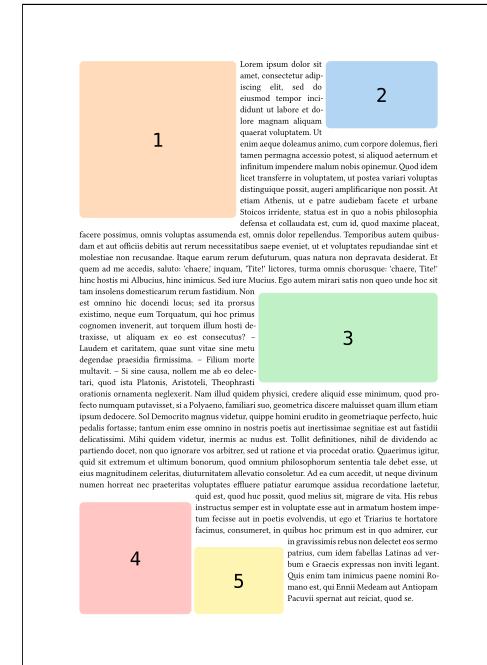
A single `#meander.reflow` invocation can contain multiple `#placed` objects. A possible limitation would be performance if the number of obstacles grows too large, but experiments have shown that up to ~100 obstacles is still workable.

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 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)]
})
```

Quod si dicitur quod est amans, consequenter etiam amans est deus. Quod si dicitur quod est amans, consequenter etiam amans est homo. Quod si dicitur quod est amans, consequenter etiam amans est animal. Quod si dicitur quod est amans, consequenter etiam amans est vegetans. Quod si dicitur quod est amans, consequenter etiam amans est res inanimatae. Quod si dicitur quod est amans, consequenter etiam amans est deus, et hoc est manifestum. Nam deus est amans, et non potest non esse amans. Quod si dicitur quod est amans, consequenter etiam amans est homo. Quod si dicitur quod est amans, consequenter etiam amans est animal. Quod si dicitur quod est amans, consequenter etiam amans est vegetans. Quod si dicitur quod est amans, consequenter etiam amans est res inanimatae. Quod si dicitur quod est amans, consequenter etiam amans est deus, et hoc est manifestum. Nam deus est amans, et non potest non esse amans.

3  
ius sit, migrare de vita. His rebus  
ructus semper est in volupte  
aut in armatum hostem  
tempore fruscere aut in poesi  
venit, unde et triarius te  
factius, consumetur; ne  
tamen in aliis tempore  
cur in prædictis iudicis non  
est esse summi patrum, cum iden  
tias Latinas ad urbem a Grecis  
ressas non invitauit. Quis  
tan inimicus pere nomini Ro  
to est, qui enim Medeum aut an  
thropam Pacuvii spernat aut reici  
tabilis detectari dicit. Latinas  
ad urbem a Syracusis ego,  
Andrian Terentii quam  
strutramen Menanisti legamus? A  
tum tantum dico, ut cum  
hocvel optime scripturam  
tam, tamen male conversam  
mii legendam putem, de quo.

1

## I.4 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.

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](#).

# 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.

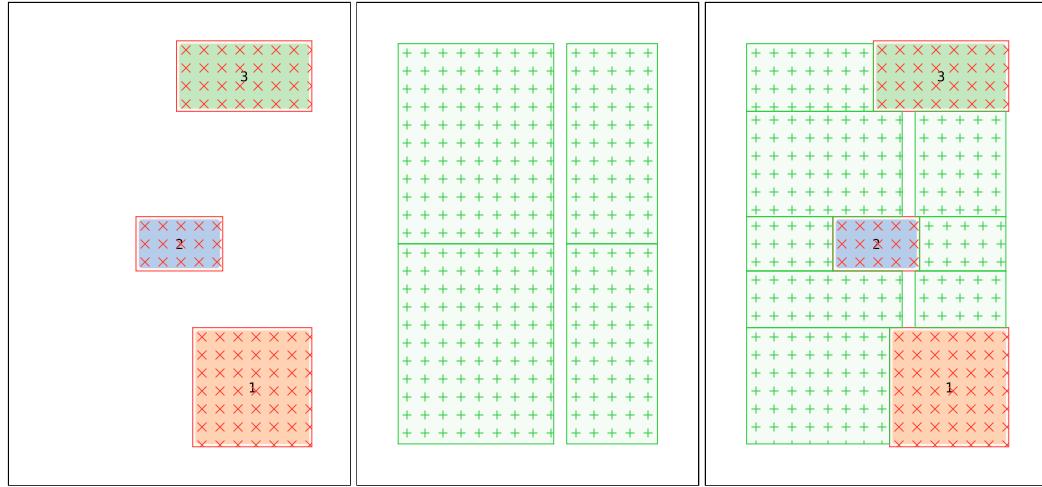


Figure 1: Left to right: the forbidden, allowed, and combined regions

## 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.

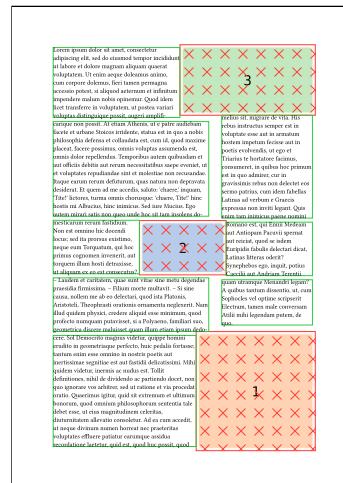


Figure 2: Debug view of the final output of `#meander.reflow.with(debug: true)`

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.

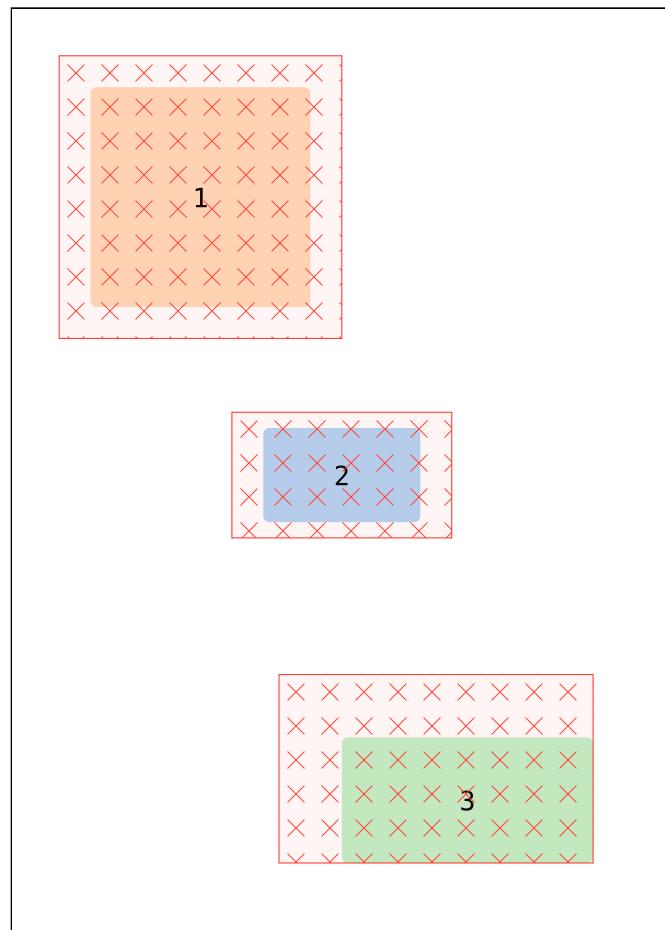
This is not a mistake. The reality is that these statements are only incompatible if we assume that 1 image = 1 obstacle. We call “contouring functions” the utilities that allow splitting one image into multiple obstacles to approximate an arbitrary shape.

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.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,
    )
})
```



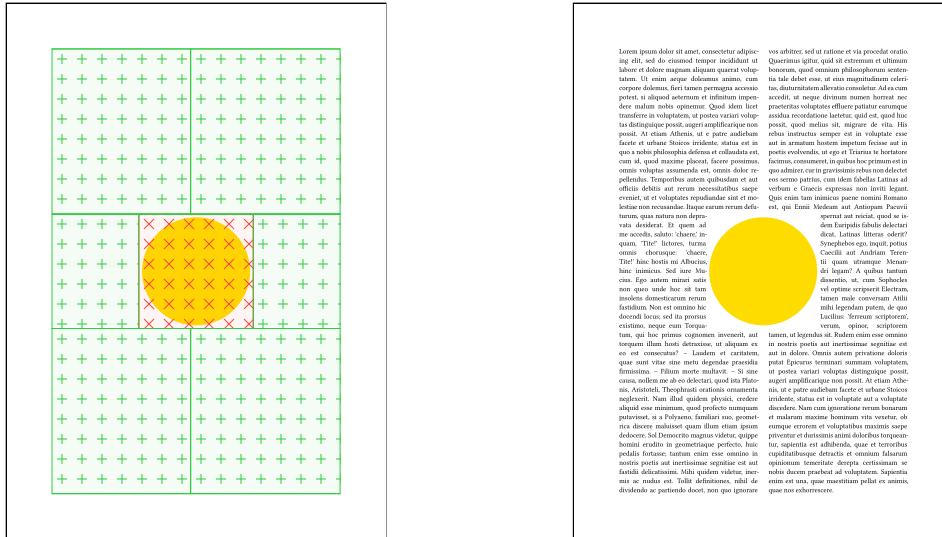
## III.2 Boundaries as equations

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()

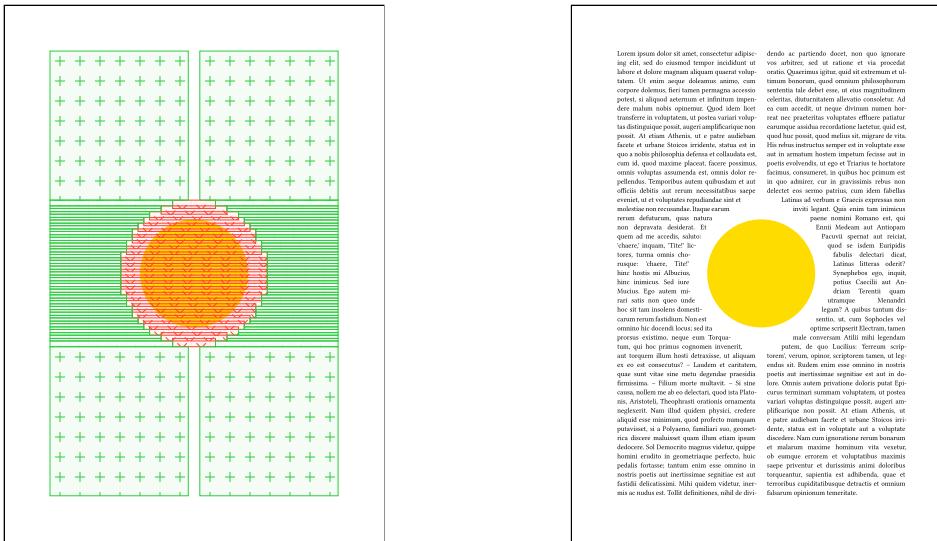
    content[
        #set par(justify: true)
        #lorem(590)
    ]
})
```



**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:

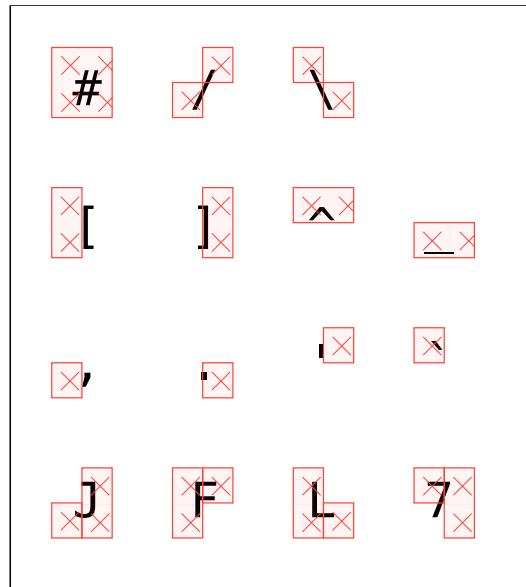
- if `(flush): horizon`, the anchor point will be the middle of the obstacle;
  - if `(flush): bottom`, the anchor point will be the bottom of the obstacle.

```
#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%

# 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.

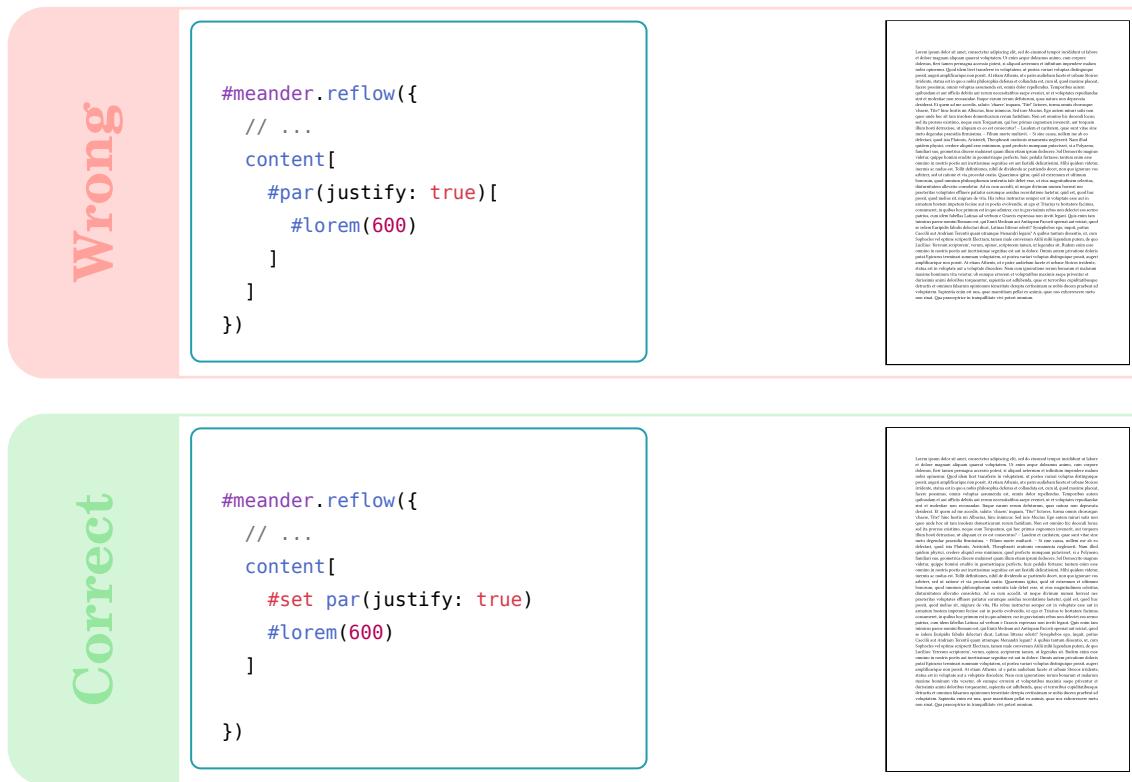
There are however styling parameters that have a consequence on layout, and some of them require special handling. Some of these restrictions may be relaxed or entirely lifted by future updates.

## 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.



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 et ea modi. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusamus et iusto odio dignissimos tsariccius.

## 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)
  ]
})
```

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 et ea modi. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusamus et iusto odio dignissimos tsariccius.

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 et ea modi. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusamus et iusto odio dignissimos tsariccius.

Correct

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

    #lorem(80)
  ]
})
```

Latin text placeholder:

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.

## 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)
  ]
})
```

Latin text placeholder:

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.

Correct

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

Latin text placeholder:

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.

Correct

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

Latin text placeholder:

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.

## IV.4 Styling containers

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

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

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

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



# Part V

## Multi-page setups

### V.1 Pagebreak

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

The layout below spans two pages:

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

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

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

    pagebreak()

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

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

### V.2 Colbreak

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

```

#meander.reflow({
  import meander: *

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


  pagebreak()
  colbreak()

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

  pagebreak()

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


})

```



## 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. This is important because `MEANDER` places all its contents, so it is by default invisible to the native layout.

## V.4.1 Default

The default, and least opinionated, mode is `(placement)`: [page](#).

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

## V.4.2 Inline

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.4.3 Full page

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 `#pagebreak`, does not consider other content.

## V.4.4 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.

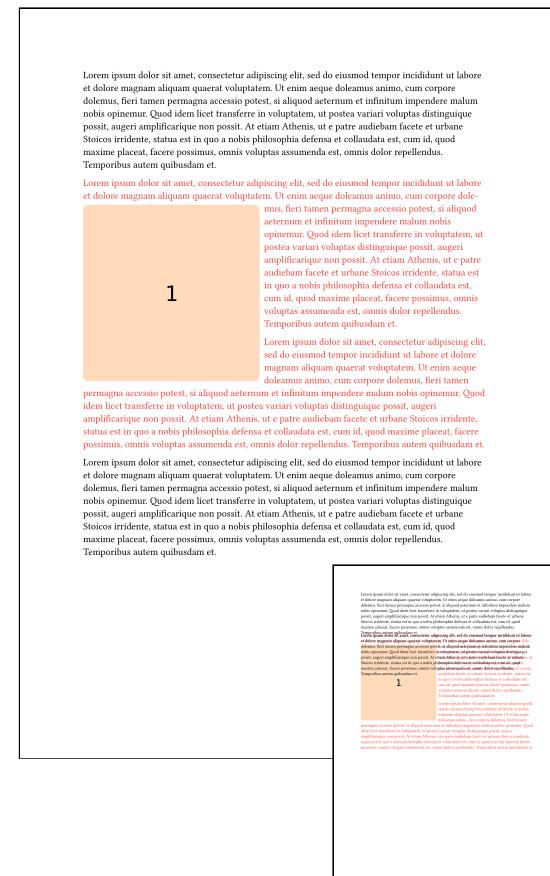
```
#lorem(100)

#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.5 Overflow

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

### V.5.1 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`

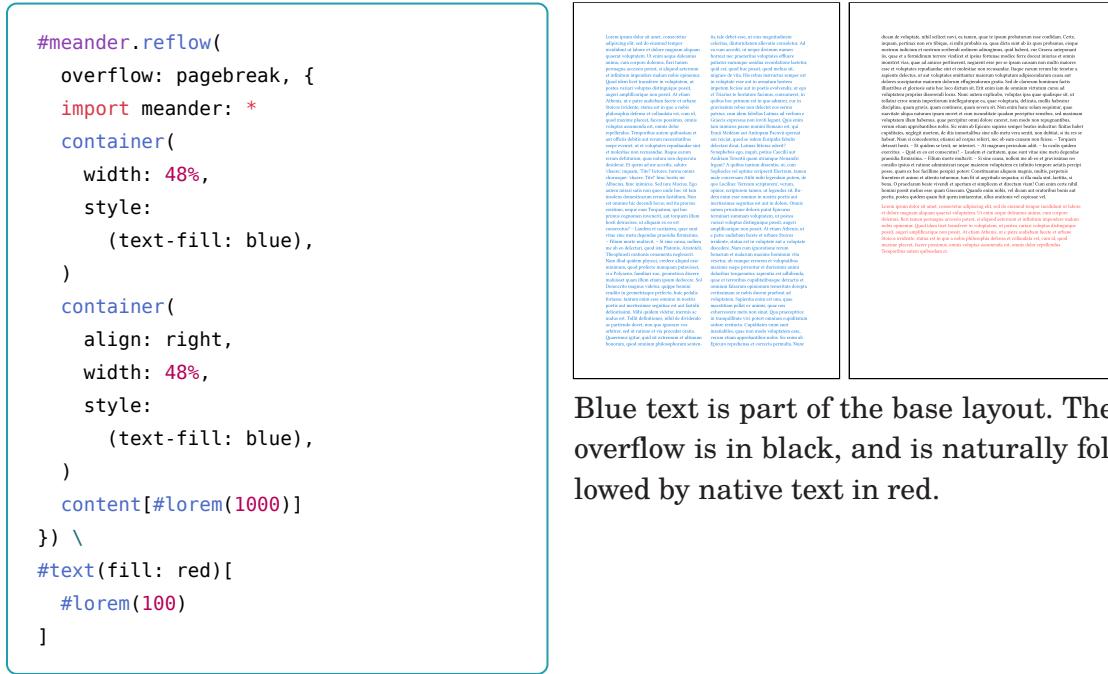
```
#meander.reflow(
  overflow: panic, {
    import meander: *
    container()
    content[#lorem(1000)]
  })
```

(panics)

## V.5.2 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 `#pagebreak`; see [Section V.4](#)).



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.

```

{text(fill: green)[
  #lorem(200)
]

#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(700)]
)

{text(fill: red)[
  #lorem(200)
]
}

```

Learn more about overflow handling at [V.4.1 Content layout behavior](#).  
 This section describes how to handle overflowing text in a page or a page's header/footer. It also describes how to handle overflowing text in a page's sidebar or in a page's sidebar's footer.  
 If you want to handle overflowing text in a page's sidebar's header, see [V.4.2 Header and footer behavior](#).  
 If you want to handle overflowing text in a page's sidebar's sidebar, see [V.4.3 Sidebar behavior](#).

Learn more about overflow handling at [V.4.1 Content layout behavior](#).  
 This section describes how to handle overflowing text in a page or a page's header/footer. It also describes how to handle overflowing text in a page's sidebar or in a page's sidebar's footer.  
 If you want to handle overflowing text in a page's sidebar's header, see [V.4.2 Header and footer behavior](#).  
 If you want to handle overflowing text in a page's sidebar's sidebar, see [V.4.3 Sidebar behavior](#).

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

### 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` 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,

## V Multi-page setups

### V.5 Overflow

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

**Overflow** is a CSS property that defines how content is handled when it exceeds the dimensions of its container. It can be applied to all block-level elements. The most common overflow values are visible, hidden, scroll, and auto.

**visible**: This is the default value. Content is displayed as normal, and no scrollbars are shown.

**hidden**: Content is clipped at the boundaries of its container, and no scrollbars are shown.

**scroll**: Content is clipped at the boundaries of its container, and scrollbars are always displayed, even if there is no content to scroll.

**auto**: Content is clipped at the boundaries of its container, and scrollbars are displayed only if there is content to scroll.

**Text Overflow** is a CSS property that specifies how text is handled when it exceeds the width of its container. It can be applied to all inline elements. The most common text-overflow values are ellipsis and none.

**ellipsis**: This value adds an ellipsis (...) at the end of the text if it exceeds the width of its container.

**none**: This value does not add any ellipsis or other markers to the text if it exceeds the width of its container.

**Image Overflow** is a CSS property that defines how images are handled when they exceed the dimensions of their container. It can be applied to all block-level elements containing images. The most common image-overflow values are visible, hidden, and scroll.

**visible**: The image is displayed as normal, and no scrollbars are shown.

**hidden**: The image is clipped at the boundaries of its container, and no scrollbars are shown.

**scroll**: The image is clipped at the boundaries of its container, and scrollbars are always displayed, even if there is no content to scroll.

**Flexbox Overflow** is a CSS property that defines how flex items are handled when they exceed the dimensions of their container. It can be applied to all flex containers. The most common flex-overflow values are visible, hidden, and scroll.

**visible**: Flex items are displayed as normal, and no scrollbars are shown.

**hidden**: Flex items are clipped at the boundaries of their container, and no scrollbars are shown.

**scroll**: Flex items are clipped at the boundaries of their container, and scrollbars are always displayed, even if there is no content to scroll.

**Grid Overflow** is a CSS property that defines how grid items are handled when they exceed the dimensions of their container. It can be applied to all grid containers. The most common grid-overflow values are visible, hidden, and scroll.

**visible**: Grid items are displayed as normal, and no scrollbars are shown.

**hidden**: Grid items are clipped at the boundaries of their container, and no scrollbars are shown.

**scroll**: Grid items are clipped at the boundaries of their container, and scrollbars are always displayed, even if there is no content to scroll.

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

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

#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)]
})
```

Latinum, ut neque divinum nomen horret nec  
peracteris voluptatis officere patitur eurymus  
assidua recitatione hæcatur, quid est, quod hæc  
positi, quod metus sit, migrare de vita His  
rebus, et in secessu, ut in secessu, et in secessu,  
et in secessu hæc impetrat, ut in secessu, et in secessu  
potius evolvetur, ut ego et Tauris te hortatore  
facimus, consumetur, in quibus hoc primum est in  
quibus, et in quibus gravissimis rebus non detectet  
se sensu patitur, id est, Latini, et Laticini, et  
verbum a Graecis expressum non invia legant.  
Quis enim tam imitius paucis nominis Romanus est,  
qui Ennius Medeum aut Antiochum Faucii  
spem aut reicit, quod se idem Euripides  
in secessu, et in secessu, et in secessu, et in secessu  
Synephobus ego, inquit, potius Cœcilius aut An-  
dreas Terentius quam oratione Menandri legant?  
A quibus res ipsas dissident, ut, cum Sophocles vel  
optime scriptor Eleusinis, et quibus res ipsas  
legimus, et quibus res ipsas potius deo Lucifer  
'terram scriptorem', verum, opinor, scriptorem  
tamen, ut legendus sit. Rudem enim esse omnino  
in secessu, et in secessu, et in secessu, et in secessu  
potius in dolo. Quis autem, in secessu, et in secessu,  
potius Epicurus terminus summae voluptatem  
ut postea variata voluptas distinguenda possit,  
angari amplificarique non possit. At etiam Athene,  
ut per puerum, et per adolescentem, et per virginem  
status est in voluptate, et in voluptate  
discovere. Nam enim ignorantes rerum bonarum  
et malarum maxime hominis vita vexatur, ob  
eumque eamque et voluptatibus omnes saepè  
potius in dolo, et in secessu, et in secessu, et in secessu  
supponit, et abducit, quas et terribilis  
equidistantibus detracit, et omnium falarum  
opiniorum temeritate derupta certissimam se  
potius in dolo, et in secessu, et in secessu, et in secessu  
discovere. Non illud quidam physici,  
crederé aliquid esse minimum, quod profecto  
numeroso putavisset, si a Polyceno, familiari suo,  
genitrix dilectione, et amori, quam illam  
ipsum detinere. Sol Democritus minus volenter  
quippe homini erudit in prometragine perfecto  
haic modis fortasse: tantum enim esse omnino  
in nostris poëtis aut ineritissimae segnitiae est aut  
fastidii delicatissimi. Mithi quidem videtur, iner-

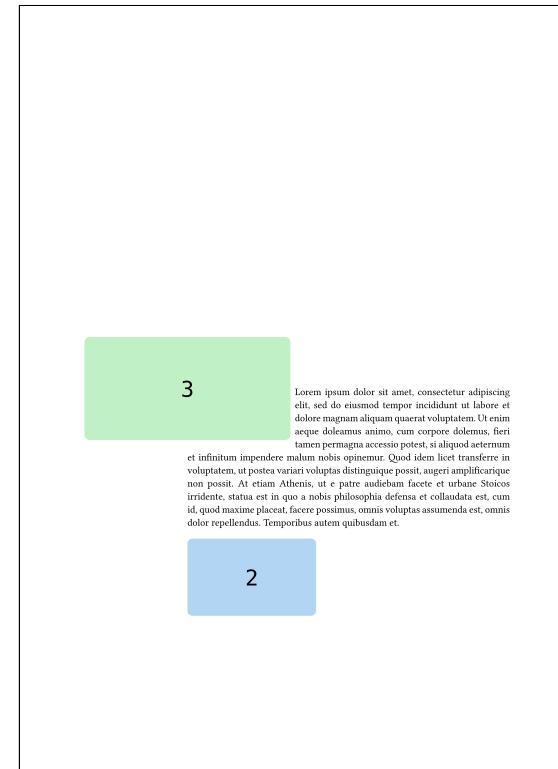
This is already doable globally by setting (boundary) to `#contour.phantom`. The innovation of (invisible) is that this can be done on a per-container basis.

## VI.2 Position and length queries

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

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

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



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

## VI.3 A nontrivial example

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

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

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

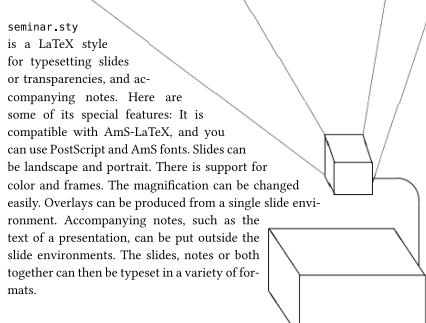
  // The actual content occupies
  // the complement of the
  // complement of the obstacles.
  container(invisible: <x>)
  content[
    #set par(justify: true)
    #lorem(225)
  ]
})
```

Lorem ipsum dolor sit amet, consecetur adipiscing elit, sed ei dominus tempor incidit ut labore et dolore magna aliquam quaerat voluptatem. Ut enim acque doleamus animo, cum corpore dolens, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod ideo licet transference in voluptatem, ut postea variari voluntas distinguere possit, augeri amplificarique non possit. At etiam Athemis, ut e patre audiebam facetus et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placet, facere possimus, omnis voluntas assumenda est, omnis dolor repellendum. Teneribus autem quibusdam et a officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluntates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedit, saluto: 'chaere', inquam, 'Tite!' **3**Dores, turma omnis choruscus: 'chaere. Tite!' hinc hostis mi Albusci, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non quo queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum **4**Torquatum qui hoc primus cognomen invenerit, aut torquem illum hosti destraxisse, ut aliam ex eo est consecutus? - Laudem et caritatem, quae sunt vitae sine metu degendae praesidia firmissima. Filium morte multavit. - Si sine causa, nollem me ab eo delectari, quod ista Platonis, Aristotelis, **5**Heophrasti orationis ornamenta.

# Part VII

# Showcase

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



examples/5181-a/main.typ

Motivated by [github:typst/typst #5181](https://github.com/typst/typst/pull/5181) (a)

Loren ipsum dolor sit amet, consecetur adipiscing elit, sed duis euismod tempor incididunt ut labore et dolore magna aliquam querat volupatem. Ut enim adeo doleamus animo, cum corpore dolendo, fieri item perniciosa auctoritate potest, si aliquod aeternum et infinitum imperium madu nobis opinemur. Quod item licet transferre in voluptam, ut postea variari voluptas distinguatur. Potes augeri amplificare non potest. At etiam Athenis, ut patre audebam et urbano Stoicos irridente, statua est in quo ubi philosophia defensa et collaudata est, cum id, quod maxime placet, faciem possimus, quam voluntate assumenda est, omnis dolor repellendus. Temporibus autem quiibusdam et omnibus debitis aut rerum necessitatibus sapie evenit, ut voluptates repudiandae sunt et molestias non recusandae. Itaque earum rerum deficimus, quas natura non depravata debeat. Et quem ad me accedit, saluto: "Haec, inquam, "Tite" factus, tunc omnis choruscus: "Haec, Tite!" hoc festus mihi Albus; hinc minutus. Sed ure Mucius. Ego autem mirari sat non sive quo unde hic tam insolens domesticarum rerum fastidum. Non est omnino hic docendi locus, sed etiam porsus exstitit, neque eum Togatuum, qui hunc primus cognovit inventum, ut tantum illorum hosti detrahebatur, ut aliquam ex eo est consecutus? - Laudem et caritatem, quae sunt vitae sine metu deprendere presidia firmissima. - Filium morte multavit. - Si sine causa, nequem vel ab eo decidi, quid platonis, Aristotelei, Theophrasti orationis ornamenta negligenter. Nam illud quidem physici, credere aliquod esse minimum, quod profecto nunquam putatus, si a Polyaeno, familiari suo, geesse metrica discere maluerunt quam illum etiam ipsius deducere. Deinde Mocinus videtur, quippe homini eruditio in geometria perfecto, hinc pedalibus fortior, tantum enim esse omnino in nostra poetae sunt. Intermissione segnitius est ad fastu deliciatissim. Mithi quidem videtur, nemisis ac nudit est. Tollit definitiones, nihil de dividendo a parte docet, non quo ignorare vos arbitriis, sed ut ratione et vno procedat oratio. Quarierimus igit, quid est extremum et summum bonum, quod omnium philosophorum sententia debet esse, ut eius magnitudinem celeriter, distuturatum alveo consuleat. Ad ea cum accedit, ut neque duminum horret ne praeteritas voluptates efficiere patitur, caruecum assida recordatione laetetur, quid est, quod huius possit, quod minus sit, migrare de vita. His rebus inceptori semper est in voluptate esse ac in auctuam hostem impetu lescisse aut in potu evolvens, ut ego et Triarius te hortatore facimus, consumetur, in quibus hoc primus est in quo admir, cur in gravissimis rebus non detecto eos sermo patrus, cum idem fabella Latinas ad verbum et Graecis expressus non inviti legant. Quis enim tam innatus paene nomini Romani est, qui Ennius Medeum aut Antipompa Pacuvii sparet aut reicit, quod si idem Euripidis fabulus detecter dicit, Latinas litteras edet? Symphodus ergo inquit, potus Caccilii aut Andram "Trenti" quoniam utrumque Menandi legem? A quibus tantum dissentio, ut quod Sophocli vel

examples/5181-b/main.typ

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

## VII Showcase

## Talmudifier Test Page

Quisque nisi  
nec faci-  
mentum hoc. Pe-  
titi sagittis  
laesit legumen-  
tibus. Aliquam feugiat  
id maledicatur.  
hodier eros  
pudinaria. Cui-  
ciliis ut tincidunt  
ta eu libero. A  
incipit nisi. Li-  
tus lacris, tristis  
non, tempus id  
lum ante ipsum  
cibus orci luctu-  
sus cubilia  
porta vel, qui  
erit. Et R. Quisque  
sit amet justo

Ali-  
quiam feugiat  
huius turpis id maledicatur. Su-  
spendisse sit amet tempor puden-  
tia. Duis velit mauris, facilis ut tincidunt, pharetra  
eu libero. Aenanc lobortis tincidunt nisi. Praesent  
met lucus, tristique sed porta non, tempus id  
lum. Et I use these red Hebrew letters in my own  
WIP project along with various other font changes. You  
might find this functionality useful. R. Alter: I  
strongly disagree with you, and future readers of this  
will have to comb through pages upon pages of what  
might as well be lorem ipsum to figure out why we're  
dunking on each other so much. As it is written: "Sed  
ut eros id arcu tincidunt accusam. Vestibulum ut  
Suspenside pharetra lorem vitae ex tincidunt ornare

Aenean sed dolor suscipit, dignissimus ligat, blandi  
eros. Pelletentesque scelerisque viverra nisl, at blandi  
congue lorem semper non. Nulla nec convallis  
neque Donec ut velit. Donec felis odio, tincidunt  
non vestibulum ut, consetetur  
egestet lectus. Donec varus finibus  
scelerisque. Aliquam eget sem non, tincidunt  
sollicitudin felis. Curabitur et ullam-  
corporis lacus. Cras eu ante quis  
aut dictum ultrices. Proin vel atque  
ut id sollicitudin autor. Nullam  
ultricies tempor neque, varius  
scelerisque neque. Nam et arcu  
ut odio maximus tempor et sit  
amet elit. *Ror Mörbi fermentum*  
dabipsum elementum. Proin id metus  
ipsum. Aenean pescus nunc quis  
lacus varius, erat molestie mauris.  
Aliquam erat volutpat. Quisque  
ac venenatis a tincidunt vel, dignissimus  
ut mi. Morbi id velit ut turpis alli-  
mentum. Curabitur hacida. *Habendum tel-*  
lus viae cras rutrum scelerisque.  
Vivamus et palepellente est, non  
imperedit massa. Cumbar dictum  
nisi sollicitudin luctus malesuada.

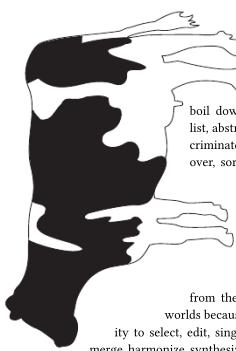
Vestibulum ac nunc dolor, sit amet ornare elec-  
tum. Etiam id dolor. *RN* In ac velit maximus elec-  
tum ex, blandi massa. Aliquam vehicula  
at neque sit amet ultrices. Integre id justo est  
Quisque luctus erat etiam aliquam faucibus. Vivamus  
et mi ac odio pretium dictum. Vestibulum vivamus  
congue risus, et acegas est dapibus eti. Aenean  
ut ore leo. Nulla dignissim erat pulvinar elit  
facilisis, ac venenatis leo tincidunt. Quisque ut  
lorem tortor. Quisque nec portitor elit. Ut finibus  
ullancorpora odio, in portitor lorem suscipit ut.

et nisl blandit, efficiunt odio vittus, dictum commu-  
nione, efficiunt effector istribo libero, eget commu-  
nione, consequent etem magna: *aptus=6&89;6240*  
get purus lectus. > Checks out how nice that little  
vianum venenatis nisi net tempus fringilla. Vivamus  
libero. Integre ac ut urna ut lectus fringilla mattis ac id  
portitor quis. Donec ut ante, semper non quam  
finibus si 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](https://github.com/typst/typst/pull/5181) (c)



examples/cow/main.typ

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

# Part VIII

## Public API

These are the user-facing functions of MEANDER.

### VIII.1 Elements

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

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

↑ Since 0.2.2

#### `#colbreak` → `elem`

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

↑ Since 0.2.3

#### `#colfill` → `elem`

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

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

Core function to create a container.

— Argument —

`{align}: top + left`

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

— Argument —

`{dx}: 0% + 0pt`

relative

Horizontal displacement.

— Argument —

`(dy): 0% + 0pt`

relative

Vertical displacement.

— Argument —

`(width): 100%`

relative

Width of the container.

— Argument —

`(height): 100%`

relative

Height of the container.

— Argument —

`(style): ()`

dictionary

↑ Since 0.2.2 Styling options for the content that ends up inside this container. If you don't find the option you want here, check if it might be in the `(style)` parameter of `#content` instead.

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

— Argument —

`(margin): 5mm`

length | dictionary

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

— Argument —

`(invisible): ()`

label | array(label)

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

— Argument —

`(tags): ()`

label | array(label)

↑ Since 0.2.3 Optional set of tags so that future element can refer to this one and others with the same tag.

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

→ flowing

Core function to add flowing content.

— Argument —

`(data)`

content

Inner content.

↑ Since 0.2.2

— Argument —

`(size): auto`

`length`

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

↑ Since 0.2.2

— Argument —

`(lang): auto`

`str`

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

↑ Since 0.2.2

— Argument —

`(hyphenate): auto`

`bool`

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

↑ Since 0.2.2

— Argument —

`(leading): auto`

`length`

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

↑ Since 0.2.1

## #pagebreak → obstacle

Continue layout to next page.

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

Core function to create an obstacle.

— Argument —

`(align)`

`align` | `position`

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

— Argument —

`(dx): 0% + 0pt`

`relative`

Horizontal displacement.

— Argument —

`(dy): 0% + 0pt`

`relative`

Vertical displacement.

— Argument —

`(boundary): (auto,)`

`contour`

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

Argument —

`(display): true`

`bool`

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

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

Argument —

`(content)`

`content`

Inner content.

Argument —

`(tags): ()`

`label | array(label)`

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

Argument —

`(anchor): auto`

`auto | align`

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

## VIII.2 Layouts

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

`#meander.reflow`      `#meander.regions`

`#meander.reflow((seq), (debug): false, (overflow): false, (placement): page) → content`

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

Argument —

`(seq)`

`array(elem)`

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

Argument —

`(debug): false`

bool

Whether to show the boundaries of boxes.

↑ Since 0.2.1

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 placed on the next page
- `#text` → the text that overflows is placed on the same page
- `#panic` → refuses to compile the document
- a `state` → stores the overflow in the state. You can then `_.get()` it later.
- any function (`overflow`) → `content` → uses that for formatting

↑ Since 0.2.2

Argument —

`(placement): page`

any

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.

`#meander.regions({seq}, {display}: true, {placement}: page, {overflow}: none) → content`

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

Argument —

`(seq)`

array(elem)

Input sequence to segment.

Argument —

`(display): true`

bool

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

↑ Since 0.2.2

Argument —

`(placement): page`

any

Controls relation to other content on the page. See analogous `(placement)` option on `#meander.reflow`.

<sup>↑</sup> Since 0.2.1

— Argument —  
**{overflow}**: `none`  
 Ignored.

## VIII.3 Contouring

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

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

<sup>↑</sup> Since 0.2.1

**`#contour.ascii-art({ascii}) → contour`**

Allows drawing the shape of the image as ascii art.

Blocks

- "#": full
- " ": empty

Half blocks

- "[": left
- "]": right
- "^": top
- "\_": bottom

Quarter blocks

- "\\"": top left
- "\)": top right
- ",": bottom left
- ".": bottom right

Anti-quarter blocks

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

Diagonals

- "/": positive
- "\": negative

— Argument —

`{ascii}`

`code` | `str`

Draw the shape of the image in ascii art.

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

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

Argument —

{div}: 5

int | (x: int, y: int)

Number of subdivisions.

Argument —

{fun}

function

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

(fraction, fraction) → bool

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

Vertical segmentation as (anchor, height).

Argument —

{div}: 5

int

Number of subdivisions.

Argument —

{flush}: horizon

align

Relative vertical alignment of the anchor.

Argument —

{fun}

function

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

(fraction) → (fraction, fraction)

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

Horizontal segmentation as (left, right)

Argument —

{div}: 5

int

Number of subdivisions.

Argument —

{fun}

function

For each location, returns the left and right bounds.

(ratio) → (ratio, ratio)

**#contour.margin(..{args}) → contour**

Contouring function that pads the inner image.

Argument

..{args}

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

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

**#contour.phantom → contour**

Drops all boundaries. Having as (boundary) a #contour.phantom will let other content flow over this object.

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

Vertical segmentation as (top, bottom)

Argument

{div}: 5

int

Number of subdivisions.

Argument

{fun}

function

For each location, returns the top and bottom bounds.

( fraction )→( fraction , fraction )

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

Horizontal segmentation as (anchor, width).

Argument

{div}: 5

int

Number of subdivisions.

Argument

{flush}: center

align

Relative horizontal alignment of the anchor.

Argument

{fun}

function

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

( fraction )→( fraction , fraction )

## VIII.4 Queries

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

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

↑ Since 0.2.3

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

↑ Since 0.2.3

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

— Argument —

`(tag)`

`label`

Reference a previous element by its tag.

— Argument —

`{at}: center`

`align`

Anchor point relative to the box in question.

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

↑ Since 0.2.3

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 Std

Some builtin functions or types are shadowed by `MEANDER`'s definitions. This module allows you to still access them.

### #pagebreak

content

Use `#std.pagebreak()` if an `#import meander: *` has shadowed the builtin `#pagebreak`.

### #colbreak

content

Use `#std.colbreak()` if an `#import meander: *` has shadowed the builtin `#colbreak`.

### #content

type

Use `std.content` if an `#import meander: *` has shadowed the builtin `content`.

### #grid

function

Use `#std.grid(...)` if an `#import meander.contour: *` has shadowed the builtin `#grid`.

### #query

function

Use `#std.query(...)` if an `#import meander: *` has shadowed the builtin `#query`.

## VIII.6 Public internals

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

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

```
#import "@preview/meander:0.2.4": internals.fill-box
```

↑ Since 0.2.4

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

```
#import "@preview/meander:0.2.4": internals.geometry
```

↑ Since 0.2.4

This grants you access to all the functions in the `geometry` module, which implement interesting 1D and 2D primitives. See [Section IX.2](#) 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 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}): 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  $a \leq b \leq c$ , 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.frac-rect({frac}, {abs}, ...{style}): block(length)**

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

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

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

Resolves an anchor point relative to a region.

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

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

Tests if two intervals intersect.

— Argument —	<code>{i1}</code>	<code>(length, length)</code>
First interval as a tuple of (low, high) in absolute lengths.		
— Argument —	<code>{i2}</code>	<code>(length, length)</code>
Second interval.		
— Argument —	<code>{tolerance}: Opt</code>	<code>length</code>
Set to nonzero to ignore small intersections.		

**#geometry.resolve({size}, ..{args}) → dictionary**

Converts relative and contextual lengths to absolute. The return value will contain each of the arguments once converted, with arguments that begin or end with "x"

or start with "w" being interpreted as horizontal, and arguments that begin or end with "y" or start with "h" being interpreted as vertical.

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

— Argument —

(size)

size

Size of the container as given by the layout function.

— Argument —

..(args)

dictionary

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

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

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

— Argument —

(obj)

dictionary

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

— Argument —

{regions}: (:)

dictionary(block)

Regions delimited by items already placed on the page.

## IX.3 Tiling

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

### #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, (elems): ()) → opaque**

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

Argument —

(size): none

size

Dimensions of the page

Argument —

(elems): ()

(..elem, )

Elements to dispense in order

**#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.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.placement-mode((placement)) → function`

Determines the appropriate layout invocation based on the placement mode. 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))`

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

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

Argument —

(seq)

array(elem)

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

## IX.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.split-word</code>
<code>#bisect.fits-inside</code>	<code>#bisect.has-text</code>	<code>#bisect.take-it-or-leave-it</code>

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

**#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 —

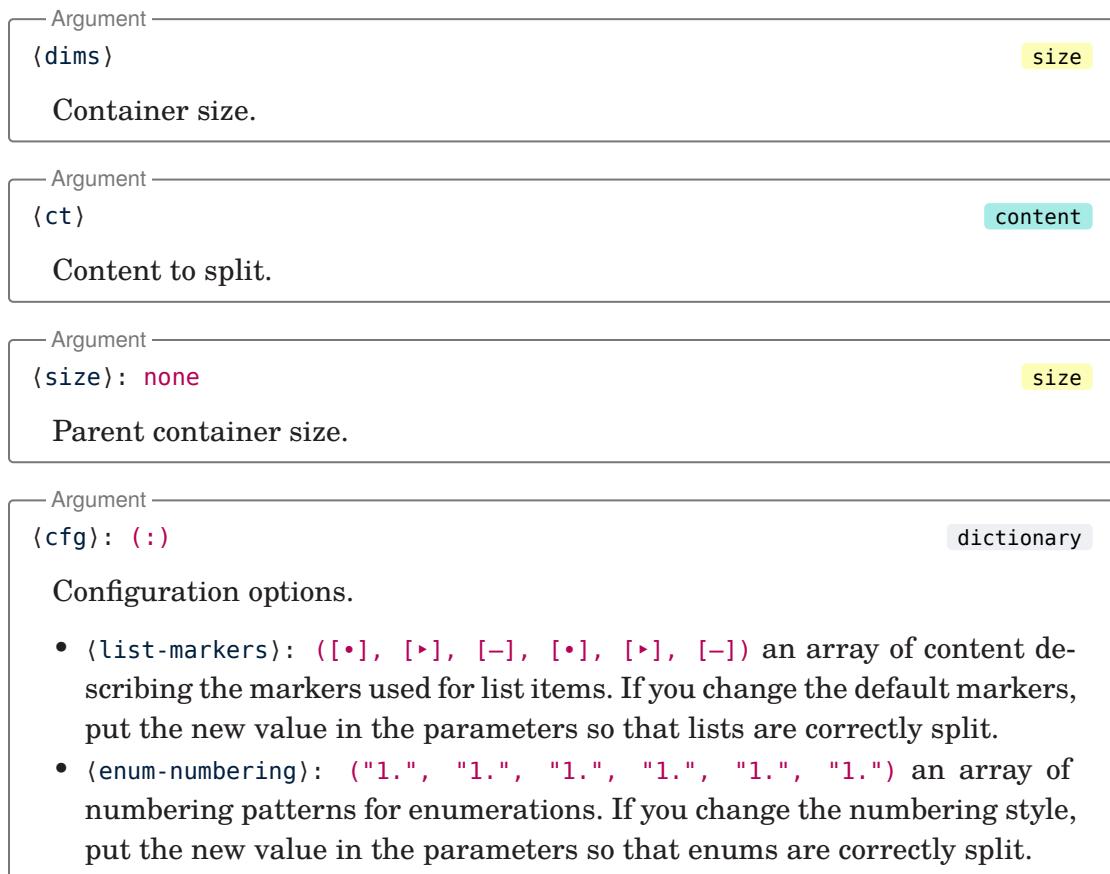
{cfg}

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.

**↖ context****#bisection.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 `#bisection.fits-inside` on them.

```

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

```

```

1 #let dims = (width: 100%, height: 50%)
2 #box(width: 7cm, height: 3cm)[#layout(size => context {
3   let words = [#lorem(15)]
4   [#fits-inside(dims, words, size: size)]
5   linebreak()
6   box(..dims, stroke: 0.1pt, words)
7 })]

```

— Argument —

(dims)

(width: relative, height: relative)

Maximum container dimensions. Relative lengths are allowed.

— Argument —

(ct)

content

Content to fit in.

— Argument —

(size): none

(width: length, height: length)

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

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

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

— Argument —

(ct)

content

Content to split.

— Argument —

(split-dispatch)

function

Recursively passed around (see **#bisect.dispatch**).

— Argument —

(fits-inside)

function

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

— Argument —

(cfg)

dictionary

Extra configuration options.

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

Split content with a "child" main field.

Strategy: recursively split the child.

Argument —

(ct)

content

Content to split.

Argument —

(split-dispatch)

function

Recursively passed around (see #bisect.dispatch).

Argument —

(fits-inside)

function

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

Argument —

(cfg)

dictionary

Extra configuration options.

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

Split content with a "children" main field.

Strategy: take all children that fit.

Argument —

(ct)

content

Content to split.

Argument —

(split-dispatch)

function

Recursively passed around (see #bisect.dispatch).

Argument —

(fits-inside)

function

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

Argument —

(cfg)

dictionary

Extra configuration options.

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

Split content with a "text" main field.

Strategy: split by " " and take all words that fit. Then if hyphenation is enabled, split by syllables and take all syllables that fit. End the block with a #linebreak that has the justification of the paragraph.

Argument –  
 (ct) content  
 Content to split.

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

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

Argument –  
 (cfg) dictionary  
 Extra configuration options.

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

Split an enum.item.

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

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

Argument –  
 (ct) content  
 Content to split.

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

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

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.5 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

## 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 XI

## About

### XI.1 Related works

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

#### In Typst:

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

#### In L<sup>A</sup>T<sub>E</sub>X:

- [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.

### XI.2 Dependencies

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

### XI.3 Acknowledgements

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

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