Meander
User guide

Abstract

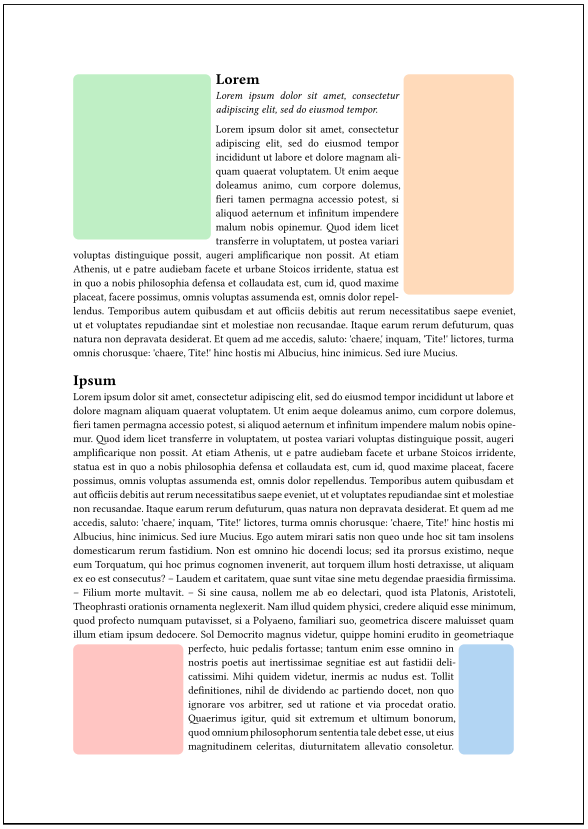
Meander implements a content layout algorithm to provide text threading (when text from one box spills into a different box if it overflows), uneven columns, and image wrap-around.

Feature requests

For as long as the feature doesn't exist natively in Typst (see issue: `github:typst/typst #5181`), feel free to submit test cases of layouts you would like to see supported by opening a new issue.

Versions

- `dev`
- `0.2.1` (`latest`)
- `0.2.0`
- `0.1.0`



Contents

I Quick start 2
II Showcase 4
III Understanding the algorithm 5
IV Advanced techniques 7
V Modularity (WIP) 14
VI Style-sensitive layout 15
VII Module details 18

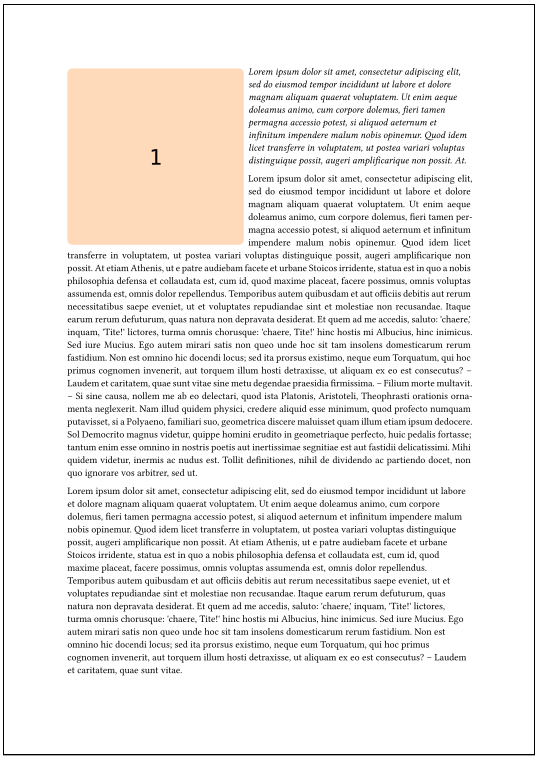
I Quick start

The main function provided 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.

I.a A simple example

Below is a single page whose layout is fully determined by Meander. Multi-page setups are also possible, see Section IV.b.

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

  // Full-page container
  container()

  // Flowing content
  content[
    _#lorem(60)_
    #[
      #set par(justify: true)
      #lorem(300)
    ]
    #lorem(200)
  ]
})
```



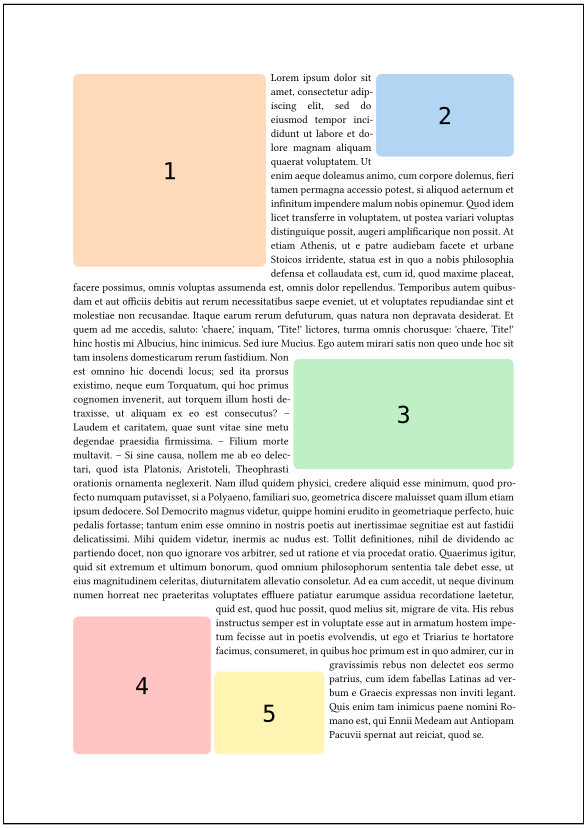
Meander is expected to respect the majority of styling options, including headings, paragraph justification, font size, etc. Notable exceptions are detailed in Section VI. If you find a discrepancy make sure to file it as a bug report if it is not already part of the known limitations.

Note: paragraph breaks may behave incorrectly. You can insert vertical spaces if needed.

I.b Multiple obstacles

`#meander.reflow` can handle as many obstacles as you provide (at the cost of potentially performance issues if there are too many, but experiments have shown that up to ~100 obstacles is no problem).

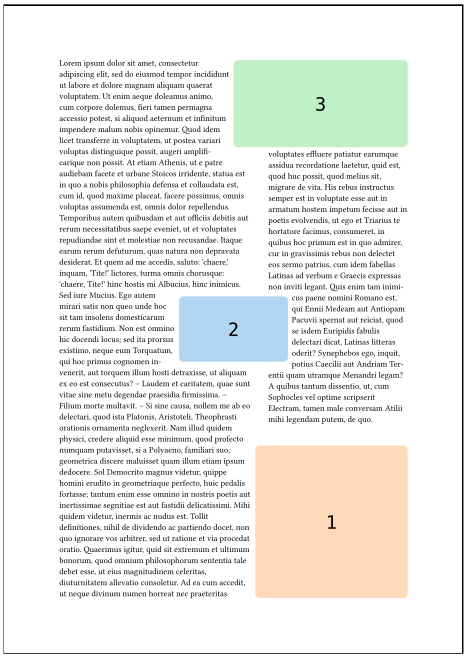
```
#meander.reflow({
  import meander: *
  // As many obstacles as you want
  placed(top + left, my-img-1)
  placed(top + right, my-img-2)
  placed(horizon + right, my-img-3)
  placed(bottom + left, my-img-4)
  placed(bottom + left, dx: 32%,
         my-img-5)

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



I.c Columns

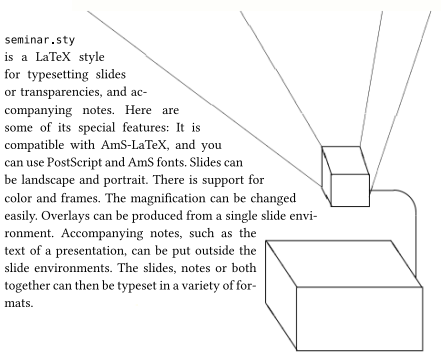
In order to simulate a multi-column layout, you can provide several `container` invocations. They will be filled in the order provided.

```
#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.
  container(width: 55%)
  container(align: right, width: 40%)
  content[#lorem(470)]
})
```



II Showcase

A selection of nontrivial examples of what is feasible.



`examples/5181-a/main.typ`

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

`examples/5181-b/main.typ`

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

`examples/talmudifier/main.typ`

From `github:subalterngames/talmudifier`

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

III Understanding the algorithm

The same page setup as the previous example I.c will internally be separated into

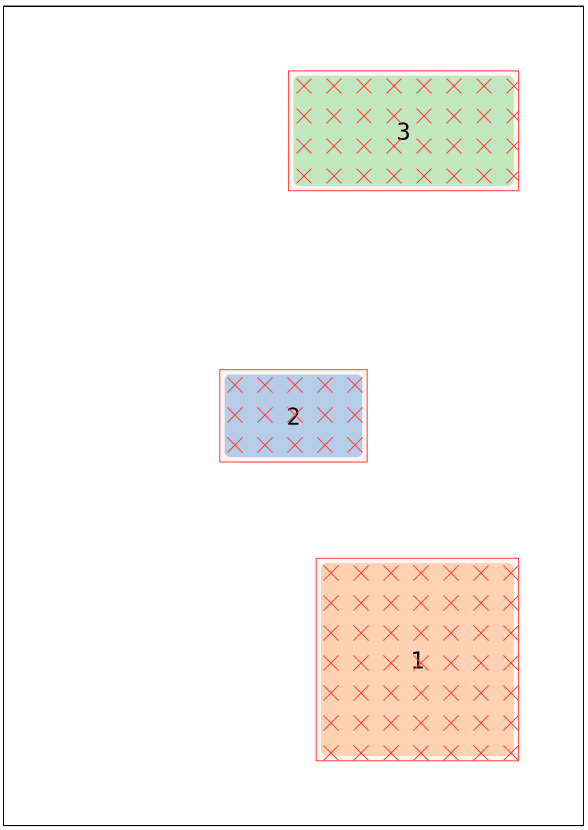
- obstacles `my-img-1`, `my-img-2`, and `my-img-3`.
- containers `#(x: 0%, y: 0%, width: 55%, height: 100%)` and `#(x: 60%, y: 0%, width: 40%, height: 100%)`
- flowing content `#lorem(470)`.

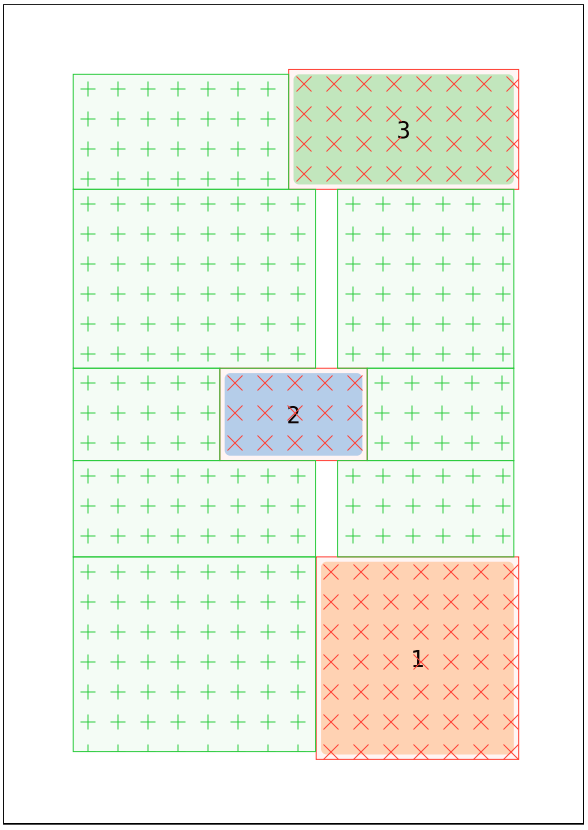
Initially obstacles are placed on the page (→). If they have a `boundary` parameter, it recomputes the exclusion zone.

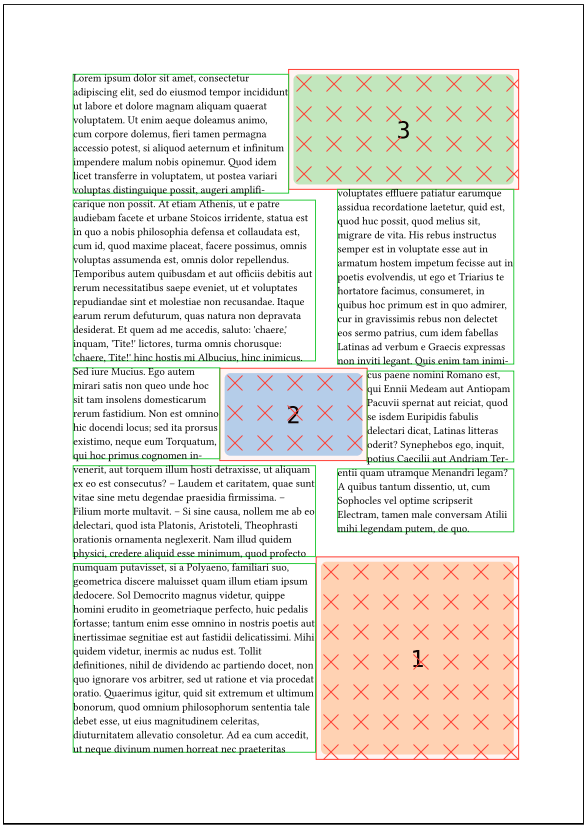
Then the containers are placed on the page and segmented into rectangles to avoid the exclusion zones (↓).

Finally the flowing content is threaded through those boxes (↘), which may be resized vertically a bit compared to the initial segmentation.

The debug views on this page are accessible via `#meander.regions` and `#meander.reflow.with(debug: true)`





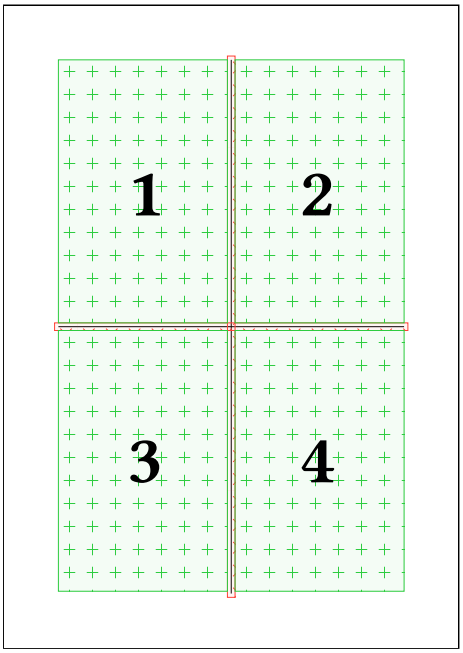


The order in which the boxes are filled is in the priority of

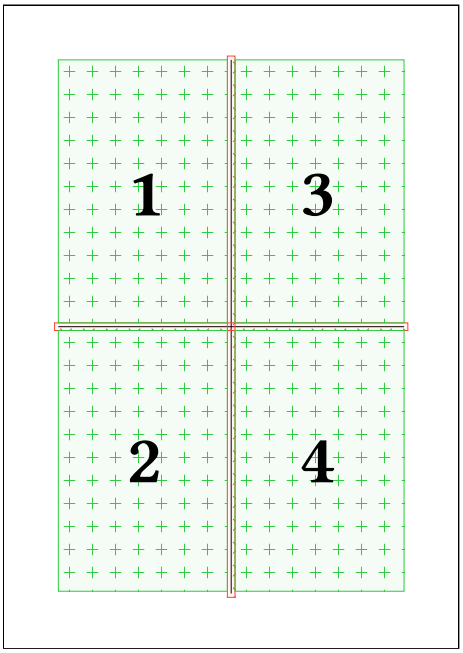
- container order
- top → bottom
- left → right

which has implications for how your text will be laid out. Indeed compare the following situations that result in the same boxes but in different orders:

```
#meander.regions({
  import meander: *
  placed(center + horizon,
    line(end: (100%, 0%)))
  placed(center + horizon,
    line(end: (0%, 100%)))

  container(width: 100%)

})
```




```
#meander.regions({
  import meander: *
  placed(center + horizon,
    line(end: (100%, 0%)))
  placed(center + horizon,
    line(end: (0%, 100%)))

  container(width: 50%)
  container(align: right, width: 50%)
})
```



And even in the example above, the box **1** will be filled before the first line of **2** is used. In short, Meander **does not "guess" columns**. If you want columns rather than a top-bottom and left-right layout, you need to specify them.

IV Advanced techniques

IV.a Obstacle contouring

Although Meander started as only a text threading engine, the ability to place text in boxes of unequal width has direct applications in more advanced paragraph shapes. This has been a desired feature since at least issue #5181.

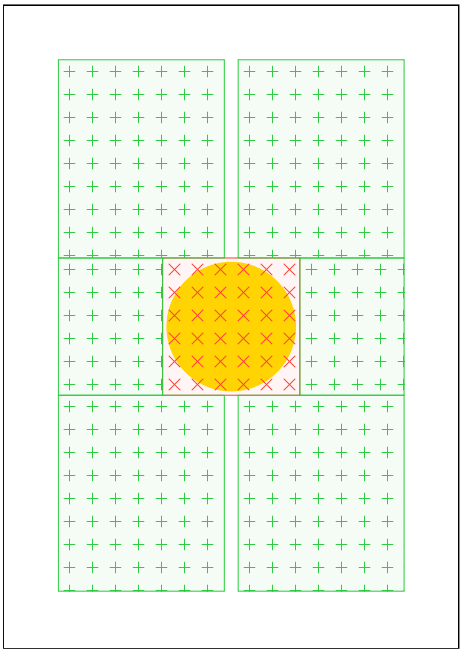
Even though this is somewhat outside of the original feature roadmap, Meander makes an effort for this application to be more user-friendly, by providing functions to redraw the boundaries of an obstacle. Here we walk through these steps.

Basic contouring is simply the ability to customize the margin around obstacles by passing to `#placed` the argument `boundary: contour.margin(1cm)`. `#contour.margin` also accepts parameters `x`, `y`, `top`, `bottom`, `left`, `right`, with the precedence you would expect, to customize the margins precisely.

As equations

Here is our 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: 48%)
  container(align: right, width: 48%)

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



Meander sees all obstacles as rectangular, so the circle leaves a big ugly square hole in our page.

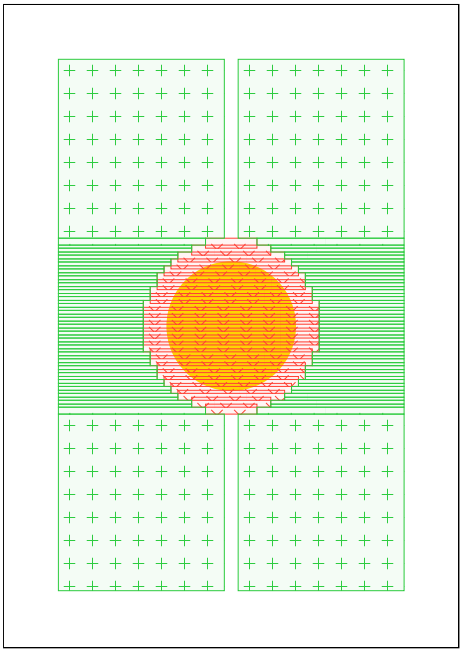
Fear not! We can redraw the boundaries. `#meander.placed` accepts as parameter `boundary` a sequence of box transformers to change the way the object affects the layout. These transformations are normalized to the interval $[0, 1]$ for convenience. The default `boundary` value is `#contour.margin(5pt)`.

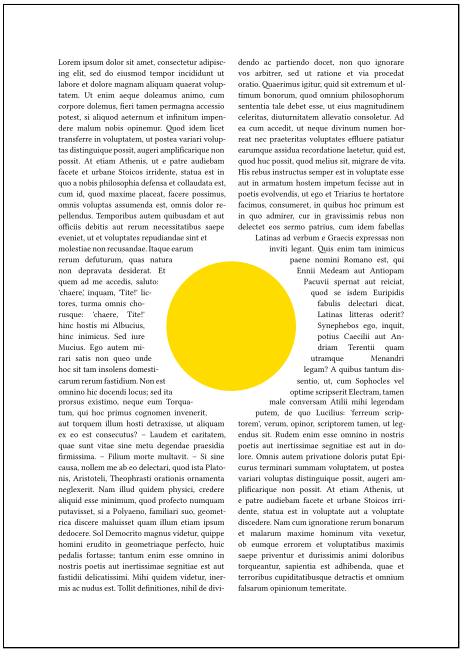
`#meander.contour.grid` is one such redrawing function, from $[0, 1] \times [0, 1]$ to `bool`, returning for each normalized coordinate `(x, y)` whether it belongs to the obstacle.

So instead of placing directly the circle, we write:

```
#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 (choose whatever looks good)
        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. Note the high density of obstacles on the debug view above: this works here but we are getting close to the resolution limit of meander, so don't try to draw obstacles with a resolution too much lower than the normal line height.

As stacked rectangles

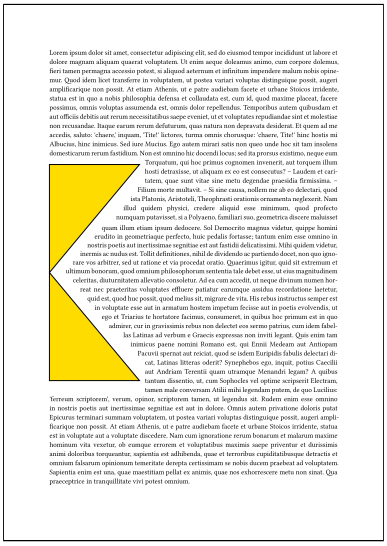
If your shape is not convenient to express through a grid function, but has some horizontal or vertical regularity, here are some other suggestions:

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

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

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

```typ
#meander.reflow({
  import meander: *
  placed(left + horizon,
    boundary:
      contour.horiz(
        div: 25,
        y => if y <= 0.5 {
          (0, 2 * (0.5 - y))
        } else {
          (0, 2 * (y - 0.5))
        },
      ) +
      contour.margin(5mm)
  )[...]
  // ...
})
```



To summarize,

- `vert(div: _, fun)`: subdivide vertically in `div` sections, then `fun(x) = (top, bottom)` produces an obstacle between `top` and `bottom`.
- `height(div: _, flush: _, fun)`: subdivide vertically in `div` sections, then `fun(x) = (anchor, height)` produces an obstacle of height `height`, with the interpretation of `anchor` depending on the value of `flush`:
  - if `flush = top` then `anchor` will be the top of the obstacle;
  - if `flush = bottom` then `anchor` will be the bottom of the obstacle;
  - if `flush = horizon` then `anchor` will be the center of the obstacle.
- `horiz`: a horizontal version of `vert`.
- `width`: a horizontal version of `height`.

All of these functions operate on values normalized to $[0, 1]$.

As ASCII art

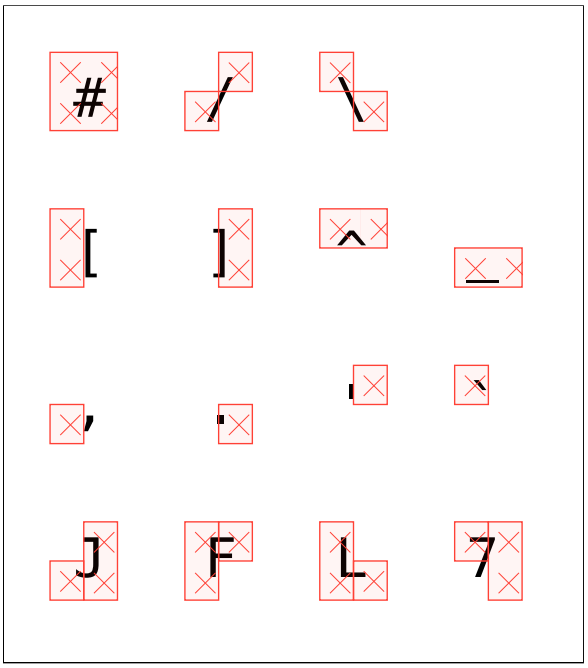
Another method of specifying contours is by drawing a rough shape of the obstacle in ASCII art. Pass a code block made of the characters `# /\[]^_,.'` `JFL7` to the contouring function `#contour.ascii-art`, and you can easily draw the shape of your image.

```
#meander.reflow(debug: true, {
  import meander: *
  placed(
    top + left,
    boundary:
      contour.margin(6mm) +
      contour.ascii-art(
        ```
        # / \

        [ ] ^ _

        , . ' `

        J F L 7
        ```
      )
  )[#image]
})
```



See `examples/5181-b/main.typ` for a nontrivial use-case.

Remarks

The contouring functions available should already cover a reasonable range of use-cases, but if you have other ideas you could always try to submit one as a new issue.

There are of course limits to this technique, and in particular increasing the number of obstacles will in turn increase the number of boxes that the layout is segmented into. This means

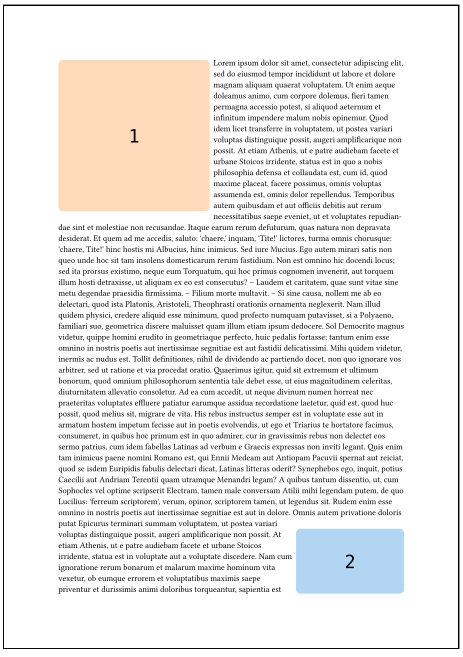
- performance issues if you get too wild (though notice that having 20+ obstacles in the previous examples went completely fine, and I have test cases with up to ~100)
- text may not fit in the boxes, and the vertical stretching of boxes still needs improvements.

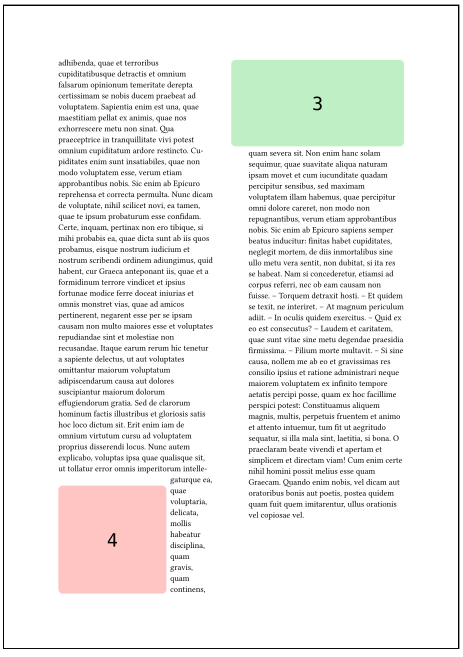
In the meantime it is highly discouraged to use a subdivision that results in obstacles much smaller than the font height.

IV.b Multi-page setups

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

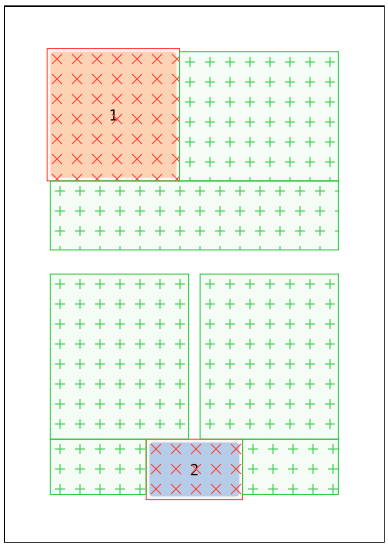
```
#meander.reflow({
  import meander: *

  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)]
})
```

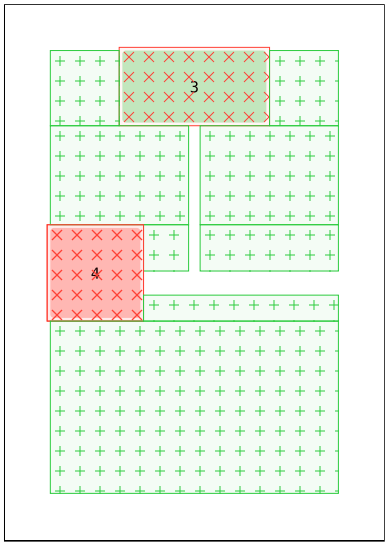




If you run into performance issues, consider finding spots where you can break the `#reflow` invocation. As long as you don't insert a `#pagebreak` explicitly, several `#reflow`s can coexist on the same page. A `#set block(spacing: 0em)` can help with the vertical alignment of invocations.

```
// Omitting this leads to a vertical
// discrepancy on obstacle 4
#set block(spacing: 0em)

// First half-page
#meander.regions({
  import meander: *
  placed(top + left, my-img-1)
  container(height: 45%)
})

// Overflows on the second page
#meander.regions({
  import meander: *
  placed(bottom + center, my-img-2)
  container(align: bottom, height: 50%, width: 48%)
  container(align: bottom + right,
            height: 50%, width: 48%)

  pagebreak()

  placed(top + center, my-img-3)
  container(height: 50%, width: 48%)
  container(align: right, height: 50%, width: 48%)

  placed(horizon, my-img-4)
})

// Takes over for the last half-page
#meander.regions({
  import meander: *
  // This obstacle is already placed by the previous
  // invocation. We just restate it without displaying
  // it so that it appears only once yet still gets
  // counted as an obstacle for both invocations.
  placed(display: false, horizon, my-img-4)
  container(align: bottom, height: 45%)
})
```





Note: the default behavior if the content provided overflows the available containers is **not** to put in on the next page, but rather to show a warning. You can manually enable the overflow going to the next page by passing to `#reflow` the parameter `overflow: pagebreak`. Other options include `overflow: panic` if you want accidental overflows to trigger an immediate panic.

V Modularity (WIP)

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

- 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.

VI Style-sensitive layout

Meander respects most styling options through a dedicated content segmentation algorithm. 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.

VI.a Paragraph justification

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

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

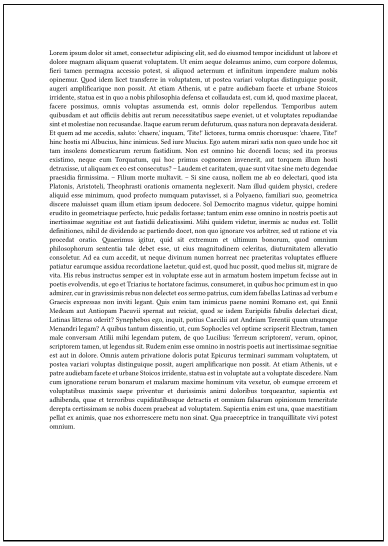
Instead prefer `#[#set par(justify: true); ...]`, or put the `#set` rule outside of the invocation of `#meander.reflow` altogether.

Wrong

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

Correct

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

})
```



VI.b Font size

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.

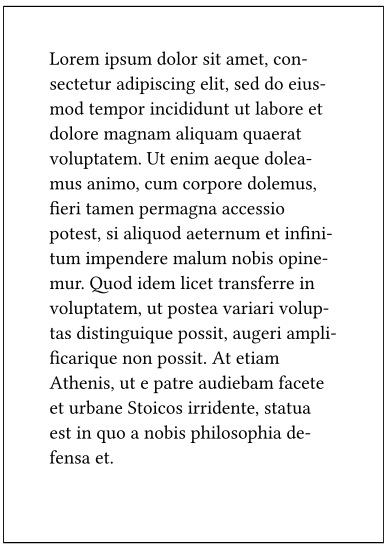
As such, it is currently discouraged to do large changes of font size in highly segmented regions from within the invocation. A future update will provide a way to do this in a more well-behaved manner.

Wrong

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

Correct

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



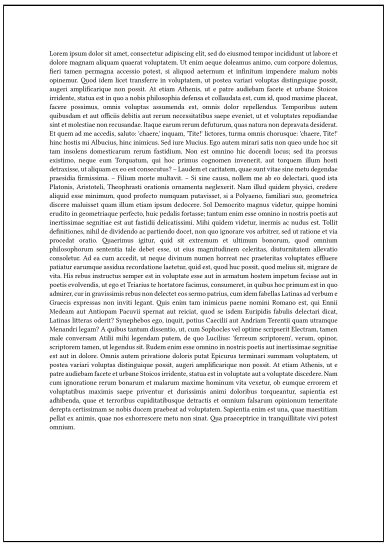
VI.c Hyphenation and language

The language is not yet configurable. This feature will come soon.

Hyphenation can only be fetched contextually, and highly influences how text is split between boxes. Thus hyphenation can currently only be enabled or disabled outside of the `#meander.reflow` invocation. A future update will provide a means to change it more locally.

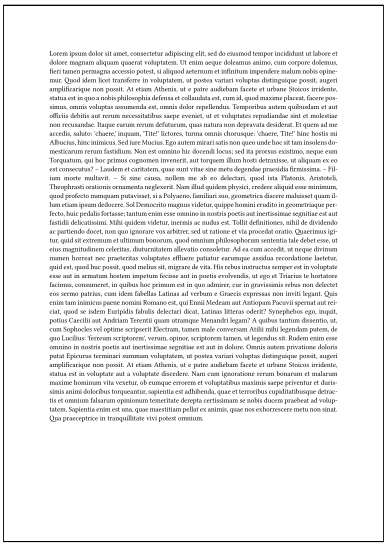
Wrong

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



Correct

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



VII Module details

VII.a Geometry (`geometry.typ`)

Generalist functions for 1D and 2D geometry.

- clamp()
- between()
- intersects()
- resolve()
- align()

clamp

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

Parameters

```
clamp(
  val: any,
  min: any none,
  max: any none
) -> any
```

val `any`

Base value.

min `any` or `none`

Lower bound.

Default: `none`

max `any` or `none`

Upper bound.

Default: `none`

between

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

Parameters

```
between(
  a: length,
  b: length,
  c: length
) -> bool
```

a `length`

Lower bound.

b `length`

Tested value.

c `length`

Upper bound. Asserted to be `>= c`.

intersects

Tests if two intervals intersect.

Parameters

```
intersects(
  i1: (length, length),
  i2: (length, length),
  tolerance: length
)
```

i1 `(length, length)`

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

i2 `(length, length)`

Second interval.

tolerance `length`

Set to nonzero to ignore small intersections.

Default: `0pt`

resolve

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

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

```
(x: 10pt, y: 101pt, width: 50pt, height: 5pt)
```

Parameters

```
resolve(
  size: (width: length, height: length),
  ..args: dictionary
) -> dictionary
```

size `(width: length, height: length)`

Size of the container as given by the `layout` function.

align

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

Parameters

```
align(
  alignment: alignment,
  dx: relative,
  dy: relative,
  width: relative,
  height: relative
) -> (x: relative, y: relative)
```

alignment `alignment`

Absolute alignment.

dx `relative`

Horizontal displacement.

Default: `0pt`

dy `relative`

Vertical displacement.

Default: `0pt`

width `relative`

Object width.

Default: `0pt`

height `relative`

Object height.

Default: `0pt`

VII.b Tiling (`tiling.typ`)

Page splitting algorithm.

- placed()
- container()
- content()
- separate()
- pat-forbidden()
- pat-allowed()
- forbidden-rectangles()
- tolerable-rectangles()
- regions()

placed

Core function to create an obstacle.

Parameters

```
placed(
  align: alignment,
  dx: relative,
  dy: relative,
  boundary: (..function,),
  display: bool,
  content: content
) -> obstacle
```

align `alignment`

Reference position on the page (or in the parent container).

dx `relative`

Horizontal displacement.

Default: `0% + 0pt`

dy `relative`

Vertical displacement.

Default: `0% + 0pt`

boundary `(..function,)`

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

Default: `(auto,)`

display `bool`

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

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

Default: `true`

content `content`

Inner content.

container

Core function to create a container.

Parameters

```
container(
  align: alignment,
  dx: relative,
  dy: relative,
  width: relative,
  height: relative
) -> container
```

align `alignment`

Location on the page.

Default: `top + left`

dx `relative`

Horizontal displacement.

Default: `0% + 0pt`

dy `relative`

Vertical displacement.

Default: `0% + 0pt`

width `relative`

Width of the container.

Default: `100%`

height `relative`

Height of the container.

Default: `100%`

content

Core function to add flowing content.

Parameters

```
content(data: content) -> flowing
```

data `content`

Inner content.

separate

Splits the input sequence into obstacles, containers, and flowing content.

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

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

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

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

Parameters

```
separate(seq: content) -> (containers: (..box,), obstacles: (..box,), flow: (..content,))
```

pat-forbidden

Pattern with red crosses to display forbidden zones.

Parameters

```
pat-forbidden(sz: length) -> pattern
```

sz `length`

Size of the tiling.

pat-allowed

Pattern with green pluses to display allowed zones.

Parameters

```
pat-allowed(sz: length) -> pattern
```

sz `length`

Size of the tiling.

forbidden-rectangles

From a set of obstacles (see `separate`: an obstacle is any `placed` content) construct the blocks `(x: length, y: length, width: length, height: length)` that surround the obstacles.

The return value is as follows:

- `rects`, a list of `blocks (x: length, y: length, width: length, height: length)`
- `display`, show this to include the placed content in the final output
- `debug`, show this to include helper boxes to visualize the layout

Parameters

```
forbidden-rectangles(
  obstacles: (..box,),
  size: (width: length, height: length)
) -> (rects: (..box,), display: content, debug: content)
```

obstacles `(..box,)`

Array of all the obstacles that are placed on this document.

size `(width: length, height: length)`

Dimensions of the parent container, as provided by `layout`.

Default: `none`

tolerable-rectangles

Partition the complement of `avoid` into `containers` as a series of rectangles.

The algorithm is roughly as follows:

```
for container in containers {
  horizontal-cuts = sorted(top and bottom of zone for zone in avoid)
  for (top, bottom) in horizontal-cuts.windows(2) {
    vertical-cuts = sorted(
      left and right of zone for zone in avoid
      if zone intersects (top, bottom)
    )
    new zone (top, bottom, left, right)
  }
}
```

The main difficulty is in bookkeeping and handling edge cases (weird intersections, margins of error, containers that overflow the page, etc.) There are no heuristics to exclude zones that are too small, and no worries about zones that intersect vertically. That would be the threading algorithm's job.

Blocks are given an additional field `bounds` that dictate the upper limit of how much this block is allowed to stretch vertically, set to the dimensions of the container that produced this block.

Parameters

```
tolerable-rectangles(
  containers: (..box,),
  avoid: (..box,),
  size: (width: length, height: length)
) -> (rects: (..box,), debug: content)
```

containers `(..box,)`

Array of the containers in which content can be placed.

avoid `(..box,)`

Array of all the obstacles that are placed on this document. Will be subtracted from `containers`.

Default: `()`

size `(width: length, height: length)`

Dimensions of the parent container, as provided by `layout`.

Default: `none`

regions

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

Parameters

```
regions(
  ct: content,
  display: bool
) -> content
```

ct `content`

Content to be segmented and have its layout displayed.

display `bool`

Whether to show the placed objects.

Default: `true`

VII.c Contouring (`contour.typ`)

Image boundary transformers.

- margin()
- frac-rect()
- horiz()
- vert()
- width()
- height()
- grid()
- ascii-art()

Variables

- phantom

margin

Contouring function that pads the inner image.

Parameters

```
margin(..args) -> function
```

..args

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

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

frac-rect

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

Parameters

```
frac-rect(
  frac: (x: fraction, y: fraction, width: fraction, height: fraction),
  abs: (x: length, y: length, width: length, height: length),
  ..style
) -> (x: length, y: length, width: length, height: length)
```

frac `(x: fraction, y: fraction, width: fraction, height: fraction)`

Child dimensions as fractions.

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

Parent dimensions as absolute lengths.

..style

Currently ignored.

horiz

Horizontal segmentation as `(left, right)`

Parameters

```
horiz(
  div: int,
  fun: function(fraction) => (fraction, fraction)
) -> function
```

div `int`

Number of subdivisions.

Default: 5

fun `function(fraction) => (fraction, fraction)`

For each location, returns the left and right bounds.

vert

Vertical segmentation as `(top, bottom)`

Parameters

```
vert(
  div: int,
  fun: function(fraction) => (fraction, fraction)
) -> function
```

div `int`

Number of subdivisions.

Default: 5

fun `function(fraction) => (fraction, fraction)`

For each location, returns the top and bottom bounds.

width

Horizontal segmentation as `(anchor, width)`.

Parameters

```
width(
  div: int,
  flush: alignment,
  fun: function(fraction) => (fraction, fraction)
) -> function
```

div `int`

Number of subdivisions.

Default: 5

flush `alignment`

Relative horizontal alignment of the anchor.

Default: `center`

fun `function(fraction) => (fraction, fraction)`

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

height

Vertical segmentation as `(anchor, height)`.

Parameters

```
height(
  div: int,
  flush: alignment.,
  fun: function(fraction) => (fraction, fraction)
) -> function
```

div `int`

Number of subdivisions.

Default: 5

flush `alignment.`

Relative vertical alignment of the anchor.

Default: `horizon`

fun `function(fraction) => (fraction, fraction)`

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

grid

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

Parameters

```
grid(
  div: int (x: int, y: int),
  fun: function(fraction, fraction) => bool
) -> function
```

div `int` or `(x: int, y: int)`

Number of subdivisions.

Default: 5

fun `function(fraction, fraction) => bool`

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

ascii-art

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

Parameters

`ascii-art(ascii: code)`

ascii `code`

Draw the shape of the image in ascii art.

phantom `function`

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

VII.d Bisection (`bisect.typ`)

Content splitting algorithm.

- fits-inside()
- default-rebuild()
- take-it-or-leave-it()
- has-text()
- has-child()
- has-children()
- is-list-item()
- is-enum-item()
- has-body()
- dispatch()

- fill-box()

fits-inside

Tests if content fits inside a box.

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

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

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

```
true
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor.
```

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

```
false
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor
incididunt ut labore.
```

Parameters

```
fits-inside(
  dims: (width: relative, height: relative),
  ct: content,
  size: (width: length, height: length)
) -> bool
```

dims `(width: relative, height: relative)`

Maximum container dimensions. Relative lengths are allowed.

ct `content`

Content to fit in.

size `(width: length, height: length)`

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

Default: `none`

default-rebuild

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

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

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

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

Content: Initial
Inner: Initial
Rebuild: foo

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

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

Content: ***Initial***
Inner: *Initial*
Rebuild: **foo**

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

Content: #content \
Inner: #inner \
Rebuild: #rebuild(([x], [y]))
```

Content: a:b
Inner: `([a], [:], [b])`
Rebuild: xy

Parameters

```
default-rebuild(
  ct: content,
  inner-field: string
) -> (dictionnary, function)
```

inner-field `string`

What "inner" field to fetch (e.g. `"body"`, `"text"`, `"children"`, etc.)

take-it-or-leave-it

"Split" opaque content.

Parameters

```
take-it-or-leave-it(
  ct: content,
  fits-inside: function
) -> (content?, content?)
```

ct `content`

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

fits-inside `function`

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

has-text

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.

Parameters

```
has-text(
  ct: content,
  split-dispatch: function,
  fits-inside: function,
  cfg: dictionary
)
```

ct `content`

Content to split.

split-dispatch `function`

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

fits-inside `function`

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

cfg `dictionary`

Extra configuration options.

has-child

Split content with a `"child"` main field. Strategy: recursively split the child.

Parameters

```
has-child(
  ct: content,
  split-dispatch: function,
  fits-inside: function,
  cfg: dictionary
)
```

ct `content`

Content to split.

split-dispatch `function`

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

fits-inside `function`

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

cfg `dictionary`

Extra configuration options.

has-children

Split content with a `"children"` main field. Strategy: take all children that fit.

Parameters

```
has-children(
  ct: content,
  split-dispatch: function,
  fits-inside: function,
  cfg: dictionary
)
```

ct `content`

Content to split.

split-dispatch `function`

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

fits-inside `function`

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

cfg `dictionary`

Extra configuration options.

is-list-item

Split a `list.item`. Strategy: recursively split the `body`, and do some magic to simulate a bullet point indent.

Parameters

```
is-list-item(
  ct: content,
  split-dispatch: function,
  fits-inside: function,
  cfg: dictionary
)
```

ct `content`

Content to split.

split-dispatch `function`

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

fits-inside `function`

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

cfg `dictionary`

Extra configuration options.

is-enum-item

Split an `enum.item`. Strategy: recursively split the `body`, and do some magic to simulate a numbering indent.

Parameters

```
is-enum-item(
  ct: content,
  split-dispatch: function,
  fits-inside: function,
  cfg: dictionary
)
```

ct `content`

Content to split.

split-dispatch `function`

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

fits-inside `function`

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

cfg `dictionary`

Extra configuration options.

has-body

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

Parameters

```
has-body(
  ct: content,
  split-dispatch: function,
  fits-inside: function,
  cfg: dictionary
)
```

ct `content`

Content to split.

split-dispatch `function`

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

fits-inside `function`

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

cfg `dictionary`

Extra configuration options.

dispatch

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.

Parameters

```
dispatch(
  ct: content,
  fits-inside: function,
  cfg: dictionary
)
```

ct `content`

Content to split.

fits-inside `function`

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

cfg `dictionary`

Extra configuration options.

fill-box

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.

Parameters

```
fill-box(
  dims: (width: length, height: length),
  ct: content,
  size: (width: length, height: length),
  cfg: dictionary
) -> (content, content)
```

dims `(width: length, height: length)`

Container size.

ct `content`

Content to split.

size `(width: length, height: length)`

Parent container size.

Default: `none`

cfg `dictionary`

Configuration options.

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

Default: `(:)`

VII.e Threading (`threading.typ`)

Filling and stretches boxes iteratively.

- smart-fill-boxes()
- reflow()

smart-fill-boxes

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

Parameters

```
smart-fill-boxes(
  body: content,
  avoid: (..block,),
  boxes: (..block,),
  extend: length,
  size: (width: length, height: length)
) -> (..content,)
```

body `content`

Flowing text.

avoid `(..block,)`

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

Default: `()`

boxes `(..block,)`

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

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

Default: `()`

extend `length`

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

Default: `1em`

size `(width: length, height: length)`

Dimensions of the container as given by `layout`.

Default: `none`

reflow

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

Parameters

```
reflow(
  ct: content,
  debug: bool,
  overflow: any
) -> content
```

ct `content`

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

debug `bool`

Whether to show the boundaries of boxes.

Default: `false`

overflow `any`

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

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

Default: `false`