

# Numbering lines in LuaTeX

Version: 0.2, 2026-06-25

Udi Fogiel, 2026

The `lualineno` module provides flexible line numbering for LuaTeX-based formats (Lua $\LaTeX$ , Op $\TeX$ , and Plain Lua $\TeX$ ).

## Table of contents

line

1	Loading <code>lualineno</code>	27
2	User Interface	32
2.1	Defining a Lineno	50
2.2	Notes For Plain Users	75
3	Examples	91
3.1	Number Lines Across the Document	92
3.2	Number Lines Per Page	122
3.3	Two Sided Document	153
3.4	Number Two Column Layout	191
3.5	Modulo	236
3.6	Reference a Line	267
3.7	Link to a Line	298
3.8	Ignore Line Numbers When Copying Text	332
3.9	The <code>offset</code> and <code>anchor</code> Keys	367
3.10	Tagging	392
4	Callbacks	420
5	Implementation	435
5.1	Lua module	436
5.2	Op $\TeX$ package	1118
5.3	$\LaTeX$ package	1123

## 1 Loading `lualineno`

To load the package you can use

```
 $\LaTeX$  \usepackage{lualineno}
Op $\TeX$  \load[lualineno]
Plain \directlua{require('lualineno')}
```

## 2 User Interface

There is only one macro, `\lualineno`, which takes a single argument consisting of a list of **key=val** pairs which are separated by spaces (or end on lines). The possible keys are

**define** Takes a list of **key=val** pairs in itself, and is used to define a new **lineno** type. See [line 51](#) for more details.

**defaults** Takes a list of **key=val** pairs in itself, and is used to set the default values of a newly defined **lineno** type. See [line 73](#) for more details.

**set** Takes a name of a defined **lineno**. Set this **lineno** as the active one.

**unset** Does not accept a value. Disables line numbers.

**label** Takes a list of tokens inside braces, e.g. `{label}`, these tokens will be fed into `\label` at a later stage. Note that like labels in  $\LaTeX$  this creates a whatsit node that can affect spacing/ligatures.

**anchor** Does not accept a value. When used after a vertical box, line numbers for lines inside that box will be positioned at the box boundaries.

**processbox** Takes an integer corresponding to a box. When using this key, numbers will be added to lines in this box. This is mainly useful for plain users who will need to use this in the output routine, but might be useful if someone wants to number lines in a strange order.

Note that a line is numbered according to the attribute of the last node in it, so it is possible to change **lineno** type in the middle of the line.

## 2.1 Defining a Lineno

When defining a **lineno** the following keys are available

**name** Accepts a string that will be used as the name for the **set** key. This key is mandatory. If the same name is used a second time, the parameters of the existing type will be modified.

**column** Takes an integer. Each **lineno** type can have different parameters in different columns, each definition only modifies the parameters for one column. The default is 1.

**toks** Takes a list of tokens inside braces. These tokens will be executed each time before line numbers are added to a line. The tokens do not have to be fully expandable, but they should not create any nodes. The default is empty.

**left** Takes a list of tokens inside braces. These tokens will be fed into an `\hbox` which will be added to the left side of a line. The default is empty.

**right** Takes a list of tokens inside braces. These tokens will be fed into an `\hbox` which will be added to the right side of a line. The default is empty.

**line** Takes two boolean keys, **number** and **recurse**. If **number** is true then hlists of subtype **line** will be numbered, default is **true**. If **recurse** is true hlists of subtype **line** will be searched recursively to check if they have lines inside of them (for example minipage inside of a line), default is **true**.

**box** The same as **line** but for hlists of subtype **box**.

**alignment** The same as **line** but for hlists of subtype **alignment**.

**equation** The same as **line** but for hlists of subtype **equation**.

**displayalignment** The same as **line** but for hlists of subtype **alignment** which are created inside a display equation.

**offset** A boolean key. If set to false, offsets will be ignored, and line numbers will be added right before or after the lines. The default is **true**.

If a key is not specified when defining a **lineno**, the default value will be used. The default values can be changed using the **defaults** key, which accepts the same keys as **define** do, except for **name**.

## 2.2 Notes For Plain Users

Plain users need to process the page box before it is shipped out. A simple example is

```
\catcode`\@=11
\def\plainoutput{%
  \setbox0\vbox{\makeheadline\pagebody\makefootline}%
  \lualineno{processbox=0}%
  \shipout\box0
  \advancepageno
  \ifnum\outputpenalty>-\@MM\else \dosupereject\fi
}
\catcode`\@=12
```

If **luatexbase** is not used, you will probably want to reallocate the attributes used by **lualineno**. You can do this with the keys **typeattr**, **markattr** and **whatsitid**, which each accept an integer. The first two set attribute numbers; **whatsitid** sets the **user\_id** of the user defined **whatsit** used for labels, which otherwise defaults to 0. Note that in this case, all alignments will be treated as regular (even those inside display equations), and the **lualineno** callbacks will not be defined.

## 3 Examples

### 3.1 Number Lines Across the Document

Since `pre_shipout_filter` is called inside the output routine, it is running inside a group, which means you should use global counters.  $\text{\LaTeX}$  already does that by default, in  $\text{OpTeX}$  we need to use the `\global` prefix.

$\text{\LaTeX}$

```
\newcounter{lineno}
\lualineno
{
  define =
  {
    name = default
    toks = {\stepcounter{lineno}}
    left = {\tiny\thelineno
            \kern.8em}
  }
  set = default
}
```

$\text{OpTeX}$

```
\newcount\lineno
\lualineno
{
  define =
  {
    name = default
    toks = {\global\advance\lineno by 1}
    left = {\the\fontsize[5]
            \the\lineno\kern.8em}
  }
  set = default
}
```

### 3.2 Number Lines Per Page

Since `pre_shipout_filter` is called inside the output routine, it is running inside a group, which means you should use local counters.  $\text{OpTeX}$  already does that by default, in  $\text{\LaTeX}$  we need to reset the counter per page.

$\text{\LaTeX}$

```
\newcounter{lineno}
\counterwithin{lineno}{page}
\lualineno
{
  define =
  {
    name = default
    toks = {\stepcounter{lineno}}
    left = {\tiny\thelineno
            \kern.8em}
  }
  set = default
}
```

$\text{OpTeX}$

```
\newcount\lineno
\lualineno
{
  define =
  {
    name = default
    toks = {\advance\lineno by 1}
    left = {\the\fontsize[5]
            \the\lineno\kern.8em}
  }
  set = default
}
```

### 3.3 Two Sided Document

To print numbers in the outer margins of a two sided document you can test the value of the page counter. Usually this is problematic in  $\text{\TeX}$ , because of its asynchronous page breaking, but since this is done right before shipout we can be certain the page counter has the correct value

$\text{\LaTeX}$

```
\newcounter{lineno}
\lualineno
{
  define =
  {
    name = default
    toks = {\stepcounter{lineno}}
    left = {\unless\ifodd\value{page}%
            \tiny\thelineno
            \kern.8em\fi}
    right = {\ifodd\value{page}%
            \tiny\kern.8em
            \thelineno\fi}
  }
  set = default
}
```

$\text{OpTeX}$

```
\newcount\lineno
\lualineno
{
  define =
  {
    name = default
    toks = {\global\advance\lineno by 1}
    left = {\unless\ifodd\pageno
            \the\fontsize[5]
            \the\lineno\kern.8em\fi}
    right = {\ifodd\pageno
            \the\fontsize[5]\kern.8em
            \the\lineno\fi}
  }
  set = default
}
```

### 3.4 Number Two Column Layout

In this example, we add numbers to the right of lines in the first column and to the left in the second column.

**LaTeX**

```
\newcounter{lineno}
\lualineno
{
  define =
  {
    name = default
    toks = {\stepcounter{lineno}}
    left = {\tiny\thelineno
           \kern.8em}
  }
  define =
  {
    name = default
    column = 2
    toks = {\stepcounter{lineno}}
    right = {\kern.8em\tiny
            \thelineno}
  }

  set = default
}
```

**OpTeX**

```
\newcount\lineno
\lualineno
{
  define =
  {
    name = default
    toks = {\global\advance\lineno by 1}
    left = {\the\fontsize[5]%
           \the\lineno\kern.8em}
  }
  define =
  {
    name = default
    column = 2
    toks = {\global\advance\lineno by 1}
    right = {\kern.8em\the\fontsize[5]%
            \the\lineno}
  }

  set = default
}
```

### 3.5 Modulo

If for example you want to print only every third line after the first, you can use something like

**LaTeX**

```
\newcounter{lineno}
\lualineno
{
  define =
  {
    name = default
    toks = {\stepcounter{lineno}}
    left = {\ifnum\value{lineno}=
           \numexpr(\thelineno-1)/3*3+1\relax
           \tiny\thelineno\kern.8em\fi}
  }
  set = default
}
```

**OpTeX**

```
\newcount\lineno
\lualineno
{
  define =
  {
    name = default
    toks = {\global\advance\lineno by 1}
    left = {\ifnum\lineno=
           \numexpr(\lineno-1)/3*3+1\relax
           \the\fontsize[5]
           \the\lineno\kern.8em\fi}
  }
  set = default
}
```

### 3.6 Reference a Line

You will need to create a label target, with `\refstepcounter` or `\wlabel`. Note that `\refstepcounter` cannot be inside the number box because the `\label` will be executed at an outer group.

**LaTeX**

```
\newcounter{lineno}
\lualineno
{
  define =
  {
    name = default
    toks = {\refstepcounter{lineno}}
    left = {\tiny
           \thelineno
           \kern.8em}
  }
  set = default
}
```

**OpTeX**

```
\newcount\lineno
\lualineno
{
  define =
  {
    name = default
    toks = {\global\advance\lineno by 1}
    left = {\the\fontsize[5]%
           \wlabel{\the\lineno}%
           \the\lineno\kern.8em}
  }
}
```

Now `\lualineno{label={foo}}` will create a label named `foo` that can be referenced later.

### 3.7 Link to a Line

When hyperref is loaded, `\refstepcounter` creates a node (a pdf destination), so we need to disable the node creation temporarily and use `\MakeLinkTarget` inside the box.

In OpTeX it is the same as with normal line reference, but we also put a destination node with `\dest`.

LaTeX

```
\newcounter{lineno}
\lualineno
{
  define =
  {
    name = default
    toks = {\LinkTargetOff
            \refstepcounter{lineno}
            \LinkTargetOn}
    left = {\MakeLinkTarget{lineno}%
            \tiny\thelineno\kern.8em}
  }
  set = default
}
```

OpTeX

```
\newcount\lineno
\lualineno
{
  define =
  {
    name = default
    toks = {\global\advance\lineno by 1}
    left = {\thefontsize[5]%
            \wlabel{\the\lineno}%
            \dest[line:\the\lineno]%
            \the\lineno\kern.8em}
  }
}
```

### 3.8 Ignore Line Numbers When Copying Text

You can put the line number inside an empty ActualText span, although not all pdf readers support that.

LaTeX

```
\newcounter{lineno}
\lualineno
{
  define =
  {
    name = default
    toks = {\stepcounter{lineno}}
    left = {\pdfextension literal page
            {/Span<</ActualText<>>>BDC}%
            \tiny\thelineno
            \pdfextension literal page{EMC}%
            \kern.8em}
  }
  set = default
}
```

OpTeX

```
\newcount\lineno
\lualineno
{
  define =
  {
    name = default
    toks = {\global\advance\lineno by 1}
    left = {\thefontsize[5]%
            \pdfliteral page
            {/Span<</ActualText<>>>BDC}%
            \the\lineno
            \pdfliteral page{EMC}\kern.8em}
  }
  set = default
}
```

### 3.9 The offset and anchor Keys

The following example demonstrates what they do

```
\newcount\lineno
\lualineno
{
  defaults =
  {
    toks = {\advance\lineno by 1}
    left = {\the\lineno\kern.8em}
    right = {\kern.8em\the\lineno}
  }
  define = {name = offset}
  define = {name = nooffset offset = false}
  set = offset
}

\def\tmp{\noindent\hfil
\frame{\vbox{
\hbox{Text}\hbox{Spanning}\hbox{Multiple}\hbox{Lines}\hbox to 5cm{}
\vskip-\baselineskip}}}

\tmp
\medskip
\tmp\lualineno{anchor}
\medskip
\lualineno{set=nooffset}
\tmp
```

1	Text	1
2	Spanning	2
3	Multiple	3
4	Lines	4
5	Text	5
6	Spanning	6
7	Multiple	7
8	Lines	8
9	Text 9	
10	Spanning 10	
11	Multiple 11	
12	Lines 12	

### 3.10 Tagging

This concerns L<sup>A</sup>T<sub>E</sub>X's PDF tagging (enabled with `\DocumentMetadata{tagging=on}`). The **left** and **right** material is typeset during the numbering pass, which runs at shipout. So that material is not added to the structure tree where it visually appears, but at the last opened structure before shipout. A plain, unmarked number is tagged as an artifact automatically, so nothing needs to be done. But if the number material itself produces structure — `\emph`, `math`, or a picture carrying alternative text, those structures end up at the place where the numbers are typeset, which is almost never what you want.

To force the numbers to be artifacts, wrap the content of **left**/**right** in `\SuspendTagging` and `\ResumeTagging`:

```

\DocumentMetadata{tagging=on}
\documentclass{article}
\usepackage{lualineno,tikzducks,lipsum}
\newcounter{lineno}
\lualineno{
  define = {
    name = default
    toks = {\stepcounter{lineno}}
    left = {\SuspendTagging{lineno}%
           \tikz[scale=0.1,actualtext=\thelineno,transform shape]
           {\duck[signpost=\thelineno]}%
           \ResumeTagging{lineno}\quad}
  }
  set = default
}
\begin{document}
\section{Some heading}
\lipsum[1]
\end{document}

```

## 4 Callbacks

There are three callbacks defined by `lualineno`

- `lualineno.pre_numbering`: A simple callback that is called before numbers are added to a line. This is where the tokens from the `toks` keyword are executed, and labels are created if Op<sub>T</sub>E<sub>X</sub> is used.
- `lualineno.numbering`: An exclusive callback that invokes the function that adds the numbers to lines. You can replace the function if you want to modify things.
- `lualineno.post_numbering`: A simple callback that is called before numbers are added to a line. This is where labels are created if L<sup>A</sup>T<sub>E</sub>X is used.

None of the callbacks need a return value, and take as arguments the following (in order)

- `line`: The line node where numbers will be added.
- `line_type`: A table of the parameters of the type of the `lineno` (see the implementation for details)
- `offset`: The calculated horizontal offset of the line from the box margin.
- `width`: The width of the page or column (or anchored box) containing the line.
- `dir`: The direction of the vertical list containing the line.

## 5 Implementation

### 5.1 Lua module

#### Initialization

Some declarations of local functions/constants of global ones to avoid table lookups.

```
lualineneno.lua
6
7 local runtoks = tex.runtoks
8 local put_next = token.unchecked_put_next
9 local create = token.create
10 local new_tok = token.new
11 local lbrace = new_tok(string.byte('{'), token.command_id'left_brace')
12 local rbrace = new_tok(string.byte('}'), token.command_id'right_brace')
13 local b = new_tok(string.byte('b'), token.command_id'other_char')
14 local d = new_tok(string.byte('d'), token.command_id'other_char')
15 local i = new_tok(string.byte('i'), token.command_id'other_char')
16 local r = new_tok(string.byte('r'), token.command_id'other_char')
17 local zero = new_tok(string.byte('0'), token.command_id'other_char')
18 local get_next = token.get_next
19 local scan_toks = token.scan_toks
20 local scan_string = token.scan_string
21 local scan_list = token.scan_list
22 local scan_int = token.scan_int
23
```

Sadly there isn't a nice way in LuaTeX to get a primitive token without using a csname. To be sure `\hbox` has the correct meaning we can use `tex.enableprimitives` to create a new csname with the meaning of the primitive, then create a token with the same `.mode` and `.command` fields so we won't need the csname anymore. All of this is to avoid to use some implementation details (`local hbox = new_tok(141, 21)`)

```
lualineneno.lua
33
34 local hbox do
35   local prefix = '@lua~line&no_'
36   while token.is_defined(prefix .. 'hbox') do
37     prefix = prefix .. '@lua~line&no_'
38   end
39   tex.enableprimitives(prefix,{'hbox'})
40   local tok = create(prefix .. 'hbox')
41   hbox = new_tok(tok.mode, tok.command)
42 end
43
44 local hlist_id = node.id('hlist')
45 local vlist_id = node.id('vlist')
46 local glyph_id = node.id('glyph')
47 local whatsit_id = node.id('whatsit')
48 local node_write = node.write
49 local tail = node.tail
50 local get_attribute = node.get_attribute
51 local set_attribute = node.set_attribute
52 local node_flush = node.flush_node
53 local insert_before = node.insert_before
54 local insert_after = node.insert_after
55 local traverse = node.traverse
56 local rangedimensions = node.rangedimensions
57 local node_copy = node.copy
58 local base_kern = node.new('kern', 'user')
59 local line_sub, eq_sub, align_sub, box_sub
60 local ignored_subtypes = {}
61 for k,v in pairs(node.subtypes("hlist")) do
62   if v == "line" then line_sub = k end
63   if v == "alignment" then align_sub = k end
64   if v == "box" then box_sub = k end
65   if v == "equation" then eq_sub = k end
66   if v == "equationnumber" then ignored_subtypes[k] = true end
67   if v == "mathchar" then ignored_subtypes[k] = true end

```

```

68 end
69 local displayalign_sub = #node.subtypes("hlist") + 1
70 for k,v in pairs(node.subtypes("vlist")) do
71     if v == "vextensible" then ignored_subtypes[k] = true end
72     if v == "vdelimiter" then ignored_subtypes[k] = true end
73 end
74
75 local setattribute = tex.setattribute
76 local texerror = tex.error
77 local texnest = tex.nest
78 local format = tex.formatname
79

```

509

The module currently works with OpTeX, L<sup>A</sup>T<sub>E</sub>X or Plain.

lualineno.lua

```

82
83 local optex, latex, plain
84 if format:find("optex") then
85     optex = true
86 elseif format:find("latex") then
87     latex = true
88 elseif format == "luatex" or
89     format == "luahbtex" or
90     format:find("plain")
91 then
92     plain = true
93 end
94 if not (optex or latex or plain) then
95     error("lualineno: The format " .. format .. " is not supported\n\n" ..
96         "Use OpTeX, LuaLaTeX or Plain.")
97 end
98
99 local lineno_types = { }
100 local lineno_attrs = { }
101 local LINENO_NUMBER = 0x1
102 local LINENO_RECURSE = 0x2
103 local lineno_marks = {
104     anchor = 1,
105     processed = 2,
106     display = 3,
107 }
108 local type_attr = luatexbase and luatexbase.new_attribute('lualineno_type') or 0
109 local mark_attr = luatexbase and luatexbase.new_attribute('lualineno_mark') or 1

```

538

Labels are carried by a user defined whatsit (see the labels section). A real allocator is used when available, otherwise the id defaults to 0; it can be reassigned with the `whatsitid` key, like the attributes.

539

lualineno.lua

```

113 local label_whatsit = luatexbase and luatexbase.new_whatsit
114     and luatexbase.new_whatsit('lualineno') or 0
115 local user_defined_sub = node.subtype("user_defined")
116 local base_whatsit = node.new('whatsit', user_defined_sub)
117 base_whatsit.user_id = label_whatsit
118 base_whatsit.type = string.byte('l')
119
120 local unset_attr = -0x7FFFFFFF
121

```

549

We use the `luakeyval` module for the user interface

lualineno.lua

```

123
124 local keyval = require('luakeyval')
125 local scan_choice = keyval.choices
126 local scan_bool = keyval.bool
127 local process_keys = keyval.process
128 local messages = {
129     error1 = "lualineno: Wrong syntax in \\lualineno",
130     value_forbidden = "lualineno: The key \"%s\" does not accept a value",
131     value_required = "lualineno: The key \"%s\" requires a value",
132 }
133

```



## 561 Numbering Lines

562 In here we define the main functions of the module, the functions that find and number the lines in a  
563 page.

564 The following function is used to number a line that is considered “real” (i.e. has some glyphs in it  
565 that are not equation number or a big math delimiter). This function is used in the `lualineno.numbering`  
566 callback, so it can be replaced if desired.

567 `line` is the hlist node representing the line, `line_type` is a lua table with the parameters defined in  
568 the define key according to the attribute and the column, `offset` is the total shift calculated from the  
569 start of the line and `width` is the width of the column containing the lines.

```
lualineno.lua
147
148 local function number_line(line, line_type, offset, width, dir)
149
150     local head = line.head
151     local is_offset = line_type['offset']
152
```

576 In case L<sup>A</sup>T<sub>E</sub>X is used without the luacolor package, we add an additional group to make the  
577 boxes color safe. Since `token.scan_list` is ran in horizontal mode, the default box direction is  
578 `\textdirection`, and this can be pretty random at shipout time, so an explicit direction is specified.  
579 Currently LTR is used, if someone ask an option can be added, but you can always use `\hbox bidr 1 {}`  
580 inside the box.

```
lualineno.lua
160
161     put_next({rbrace, rbrace})
162     put_next(line_type.left)
163     put_next({\hbox,b,d,i,r,zero,lbrace,lbrace})
164     put_next({rbrace, rbrace})
165     put_next(line_type.right)
166     put_next({\hbox,b,d,i,r,zero,lbrace,lbrace})
167
```

589 To make sure “right” always means right, we check the line direction.

```
lualineno.lua
169
170     local end_box, start_box
171     if line.dir == "TLT" then
172         end_box = scan_list()
173         start_box = scan_list()
174     else
175         start_box = scan_list()
176         end_box = scan_list()
177     end
178
```

600 If the vertical list containing the line and the line has different directions we need to mirror the  
601 kerns as the kern means opposite direction then the shift, or alignment (similar to how `\shapemode` is  
602 working).

```
lualineno.lua
182
183     local start_kern_width = 0
184     local end_kern_width = 0
185     if is_offset then
186         if line.dir == dir then
187             start_kern_width = offset
188             end_kern_width = width - line.width - offset
189         else
190             start_kern_width = width - line.width - offset
191             end_kern_width = offset
192         end
193     end
194
195     if start_box.head then
196         if start_kern_width ~= 0 then
197             local start_kern = node_copy(base_kern)
198             start_kern.kern = start_kern_width
199             head = insert_before(head, head, start_kern)
200         end
201         head = insert_before(head, head, start_box)

```

```

202     start_kern_width = -start_box.width - start_kern_width
203     if start_kern_width ~= 0 then
204         local start_kern = node_copy(base_kern)
205         start_kern.kern = start_kern_width
206         head = insert_before(head, head, start_kern)
207     end
208 else
209     node_flush(start_box)
210 end
211 if end_box.head then
212     if end_kern_width ~= 0 then
213         local end_kern = node_copy(base_kern)
214         end_kern.kern = end_kern_width
215         head = insert_after(head, tail(head), end_kern)
216     end
217     head = insert_after(head, tail(head), end_box)
218 else
219     node_flush(end_box)
220 end
221 line.head = head
222 end
223
224 local lineno_callbacks
225 if luatexbase then
226     luatexbase.create_callback('lualineno.pre_numbering', 'simple', false)
227     luatexbase.create_callback('lualineno.numbering', 'exclusive', number_line)
228     luatexbase.create_callback('lualineno.post_numbering', 'simple', false)
229     luatexbase.add_to_callback('lualineno.pre_numbering', function(_, line_type)
230         runtoks(function() put_next(line_type['toks']) end)
231     end, 'lualineno.runtoks')
232     local call_callback = luatexbase.call_callback
233     lineno_callbacks = function(line, line_type, offset, width, dir)
234         call_callback('lualineno.pre_numbering', line, line_type, offset, width, dir)
235         call_callback('lualineno.numbering', line, line_type, offset, width, dir)
236         call_callback('lualineno.post_numbering', line, line_type, offset, width, dir)
237     end
238 else
239     lineno_callbacks = function(line, line_type, offset, width, dir)
240         runtoks(function() put_next(line_type['toks']) end)
241         number_line(line, line_type, offset, width, dir)
242     end
243 end
244

```

Not every object that would be considered a line from LuaTeX's point of view would be considered a line from a human perspective. For example, a line containing only an indent box, or an alignment containing only rules, so we use the following two functions to search for a glyph node recursively, while ignoring boxes for equation number, for big delimiters (i.e. in `cases` environment) or dummy boxes for null delimiters.

```

252
253 local function real_box(list)
254     for n, id, sb in traverse(list) do
255         if id == glyph_id then
256             return true
257         elseif (id == hlist_id or id == vlist_id) and not ignored_subtypes[sb] then
258             if real_box(n.list) then
259                 return true
260             end
261         end
262     end
263     return false
264 end
265

```

If the first thing (that we care about) in a line is a glyph we simply number it, in which case `true` is returned. Otherwise the line is just a wrapper around boxes, so we collect every `vlist` reachable through box-only chains into `found` (each as a `{node, hoffset, voffset}` triple) and return that list, so the caller can recurse into all of them. We can't assume there is only a single box to recurse into: an `hbox` can

hold several numbering targets, either stacked vertically (pgf stacks the title and the body of a tcolorbox with `\raise/\lower`) or side by side (the columns of `multicol`, `paracol`, two-column layouts, ...).

For each box we record both its horizontal offset (for placing the number) and its vertical offset `voffset`. Inside an `hlist` the `shift` field is a *vertical* displacement, so we accumulate it on every hop. The caller uses `voffset` to tell columns (boxes side by side, sharing a `voffset`) from a vertical stack (boxes at distinct `voffsets`).

The one structure where the boxes of an `hbox` form a *single* line that should be numbered once is the cells of a table row. Those are told apart by the caller using the subtype of the line (`alignment`), not here.

```

286
287 local function real_line(list, parent, offset, voffset, found)
288     found = found or {}
289     voffset = voffset or 0
290     for n, id, sb in traverse(list) do
291         if id == glyph_id then
292

```

A leading glyph means this really is a text line, so number it as a whole. A glyph encountered after we already collected boxes is just part of the wrapper and is ignored.

```

296
297     if #found == 0 then return true end
298     elseif id == vlist_id and not ignored_subtypes[sb] and real_box(n.list) then
299         found[#found+1] = {n, offset + rangedimensions(parent, list, n), voffset + n.shift}
300     elseif id == hlist_id and not ignored_subtypes[sb] and real_box(n.list) then
301         if real_line(n.list, n, offset + rangedimensions(parent, list, n),
302             voffset + n.shift, found) == true then
303             return true
304         end
305     end
306 end
307 return found
308 end
309

```

This function finds the lines that needs to be numbered in a page. It should be used right before shipout, but can be used on individual boxes using the `processbox` key if needed (maybe special numbering order is desired). When a line found, `lineno_callbacks` is called to number it.

```

315 local find_line
316 find_line = function(parent, list, column, offset, width)
317
318     if get_attribute(parent, mark_attr) == lineno_marks.processed then return end
319     set_attribute(parent, mark_attr, lineno_marks.processed)
320

```

We need to keep track of the parent id to know if the `.shift` field represent horizontal or vertical displacement.

```

323
324     local parent_is_vlist = parent.id == vlist_id
325     for n, id, sb in traverse(list) do
326

```

Lines are `hlists`, so if a node is not one we dig deeper, while calculating the offset and the width. If a box marked with the `anchor` key is found then the offset is reset and the width is updated.

```

331
332     if id ~= hlist_id then
333         if not n.list then goto continue end
334         local new_offset, new_width = offset, width
335         if get_attribute(n, mark_attr) == lineno_marks.anchor then
336             new_offset, new_width = 0, n.width
337         elseif parent_is_vlist then
338             new_offset = new_offset + n.shift
339         end
340         find_line(n, n.list, column, new_offset, new_width)
341         goto continue
342     end
343

```

A line type is determined by the attribute of its last node so that line types can be switched from within the line (but maybe this should be configurable). The flag is a bitset that determines whether to number or recurse further.

```

347
348     local line_attr = n.head and get_attribute(tail(n.head), type_attr)
349     local line_type = line_attr and lineno_types[line_attr] and lineno_types[line_attr][column]
350     local flag
351     if sb == align_sub and get_attribute(n, mark_attr) == lineno_marks.display then
352         flag = line_type and line_type[displayalign_sub]
353     else
354         flag = line_type and line_type[sb]
355     end
356

```

If a line does not have any attribute we don't number it, but we do recurse further.

```

358
359     local should_number = flag and (flag & LINENO_NUMBER) ~= 0 or false
360     local should_recurse = flag and (flag & LINENO_RECURSE) ~= 0 or true
361     if not (should_number or should_recurse) then
362         goto continue
363     end
364

```

This is the case where a line should be numbered only once. Maybe someone would like to number alignment once, regardless of the fact the first column contains cells with paragraphs.

```

368
369     if should_number and not should_recurse then
370         if real_box(n.list) then
371             local new_offset = parent_is_vlist and (offset + n.shift) or offset
372             lineno_callbacks(n, line_type, new_offset, width, parent.dir)
373         end
374         goto continue
375     end
376

```

If `real_line` did not return `true`, the line is a wrapper around boxes, so we need to find lines inside of each collected box as well. As before offset and width might need to be updated.

The boxes of a table row are the exception: they are the cells of a single line and must be numbered once, so for an alignment we only keep the first box (this is what `real_line` used to return before it learned to collect several boxes).

Otherwise we look at the vertical offsets of the collected boxes: boxes sharing a `voffset` sit side by side and are columns, so each is processed relative to its own left edge and width, and gets a column number assigned in the order they appear (LuaTeX keeps the columns in logical order). Boxes alone at their `voffset` are a vertical stack (or just a single box) and keep the inherited offset, width and column.

```

392
393     local found = real_line(n.head, n, offset)
394     if found ~= true then
395         local last = sb == align_sub and math.min(#found, 1) or #found
396

```

A `voffset` shared by more than one box marks a row of columns; we count the boxes per `voffset` to recognize them and number them within the row.

```

399
400     local per_voffset = {}
401     for i = 1, last do
402         local v = found[i][3]
403         per_voffset[v] = (per_voffset[v] or 0) + 1
404     end
405     local col_number = {}
406     for i = 1, last do
407         local box = found[i]
408         local m, v = box[1], box[3]
409         local new_offset, new_width, new_col = box[2], width
410         if per_voffset[v] > 1 then
411             col_number[v] = (col_number[v] or 0) + 1

```

```

412         new_col = col_number[v]
413         new_offset, new_width = 0, m.width
414         elseif get_attribute(m, mark_attr) == lineno_marks.anchor then
415             new_offset, new_width = 0, m.width
416         elseif parent_is_vlist then
417             new_offset = new_offset + n.shift
418         end
419         find_line(m, m.head, new_col or column, new_offset, new_width)
420     end
421     goto continue
422 end
423
424     if not should_number then goto continue end
425

```

826 A line is found! Update the offset and number it.

lualineneno.lua

```

427
428     local new_offset = parent_is_vlist and (offset + n.shift) or offset
429     lineno_callbacks(n, line_type, new_offset, width, parent.dir)
430
431     ::continue::
432 end
433 end
434
435 if not plain then
436     luatexbase.add_to_callback('pre_shipout_filter', function(box)
437         find_line(box, box.list, 1, 0, box.width)
438         return true
439     end, 'lualineneno.shipout')
440 end
441

```

842 Check if inside display for display alignment.

lualineneno.lua

```

443
444 if luatexbase then
445     luatexbase.add_to_callback("buildpage_filter", function(info)
446         if info == "before_display" then
447             setattribute(mark_attr, lineno_marks.display)
448         elseif info == "after_display" then
449             setattribute(mark_attr, unset_attr)
450         end
451     end, "lualineneno.indisplay")
452 end
453

```

854 Anchoring numbers to a box

lualineneno.lua

```

455
456 local function mark_last_vlist(n)
457     local current = n
458     while current do
459         if current.id == vlist_id then
460             set_attribute(current, mark_attr, lineno_marks.anchor)
461             return true
462         elseif current.id == hlist_id then
463             if mark_last_vlist(tail(current.list)) then return true end
464         end
465         current = current.prev
466     end
467     return false
468 end
469

```

870 labels

lualineneno.lua

```

471

```

872 A label is attached by inserting a user\_defined whatsit carrying the label tokens as a lua table at  
873 the point of the \lualineneno{label=...} call.

```

475
476 local make_label
477 local function find_label(line)
478     local head = line.list
479     for n, id, sb in traverse(head) do
480         if n.list then
481             find_label(n)
482         elseif id == whatsit_id and sb == user_defined_sub and n.user_id == label_whatsit then
483             make_label(n.value, head, n)
484         end
485     end
486 end
487

```

887 The carrier stores the label tokens directly in the `whatsit value` field (a lua table, type 1). On its  
888 first use the function registers `find_label` with the numbering callback and then redefines itself to the  
889 bare inserter.

```

492
493 local add_label
494 add_label = function(tokens)
495     if optex then
496         luatexbase.add_to_callback('lualineno.pre_numbering', find_label, 'lualineno.labels')
497     elseif latex then
498         luatexbase.add_to_callback('lualineno.post_numbering', find_label, 'lualineno.labels')
499     end
500     add_label = function(toks)
501         local w = node_copy(base_whatsit)
502         w.value = toks
503         w.attr = node.current_attr()
504         node_write(w)
505     end
506     return add_label(tokens)
507 end
508

```

## 907 User Interface

908 This section describes the definition of the one macro exposed to the end user. It is based on the `luakeyval`  
909 module.

```

513
514 local defaults = {
515     toks = { },
516     left = { },
517     right = { },
518     box = {number = true, recurse = true},
519     alignment = {number = true, recurse = true},
520     displayalignment = {number = true, recurse = true},
521     equation = {number = true, recurse = true},
522     line = {number = true, recurse = true},
523     offset = true,
524     column = 1,
525 }
526
527 local inner_keys = {
528     number = {scanner = scan_bool, default = true},
529     recurse = {scanner = scan_bool, default = true}
530 }
531
532 local defaults_keys = {
533     toks = {scanner = scan_toks},
534     left = {scanner = scan_toks},
535     right = {scanner = scan_toks},
536     box = {scanner = process_keys, args = {inner_keys, messages}},
537     alignment = {scanner = process_keys, args = {inner_keys, messages}},
538     displayalignment = {scanner = process_keys, args = {inner_keys, messages}},
539     equation = {scanner = process_keys, args = {inner_keys, messages}},
540     line = {scanner = process_keys, args = {inner_keys, messages}},
541     offset = {scanner = scan_bool},
542     column = {scanner = scan_int}

```

```

543 }
544
545 local function set_defaults()
546     local vals = process_keys(defaults_keys, messages)
547     for k,v in pairs(vals) do
548         defaults[k] = v
549     end
550 end
551
552 local define_keys = { }
553 for k,v in pairs(defaults_keys) do
554     define_keys[k] = v
555 end
556 define_keys.name = {scanner = scan_string}
557
558 local function define_lineno()
559     local vals = process_keys(define_keys, messages)
560     local name = vals['name']
561     if not name then
562         texerror("lualineno: Missing name when defining a lineno")
563     return
564     end
565
566     local col = vals['column'] or defaults.column
567     lineno_attrs[name] = lineno_attrs[name] or #lineno_types + 1
568     local i = lineno_attrs[name]
569     lineno_types[i] = lineno_types[i] or {}
570     lineno_types[i][col] = lineno_types[i][col] or {}
571
572     local c = lineno_types[i][col]
573
574     local function store_type(key, subtype_id)
575         local setting = vals[key] or defaults[key]
576         local flags = 0
577         if setting.number then flags = flags | LINENO_NUMBER end
578         if setting.recurse then flags = flags | LINENO_RECURSE end
579         c[subtype_id] = flags
580     end
581
582     store_type('box', box_sub)
583     store_type('alignment', align_sub)
584     store_type('equation', eq_sub)
585     store_type('line', line_sub)
586     store_type('displayalignment', displayalign_sub)
587
588     c.toks = vals.toks or defaults.toks
589     c.left = vals.left or defaults.left
590     c.right = vals.right or defaults.right
591     if vals.offset ~= nil then
592         c.offset = vals.offset
593     else
594         c.offset = defaults.offset
595     end
596 end
597
598 local lualineno_keys = {
599     set = {scanner = scan_string},
600     unset = { default = true },
601     define = {scanner = function() return true end, func = define_lineno},
602     defaults = {scanner = function() return true end, func = set_defaults},
603     anchor = { default = true },
604     label = {scanner = scan_toks, args = {false, true}},
605     typeattr = {scanner = scan_int},
606     markattr = {scanner = scan_int},
607     whatsitid = {scanner = scan_int},
608     processbox = {scanner = scan_int},
609 }
610
611 local function lualineno()
612     local saved_endlinechar = tex.endlinechar

```

```

613 tex.endlinechar = 32
614 local vals = process_keys(lualineneno_keys,messages)
615 tex.endlinechar = saved_endlinechar
616 if vals.set then
617     local attr = lineneno_attrs[vals.set]
618     if attr then
619         setattribute(type_attr, attr)
620     else
621         texerror("lualineneno: type '" .. vals.set .. "' undefined")
622     end
623 end
624 if vals.unset then
625     setattribute(type_attr, unset_attr)
626 end
627 if vals.anchor then
628     for i=texnest.ptr,0,-1 do
629         if mark_last_vlist(texnest[i].tail) then return end
630     end
631 end
632 if vals.label then
633     add_label(vals.label)
634 end
635 type_attr = vals.typeattr or type_attr
636 mark_attr = vals.markattr or mark_attr
637 label_whatsit = vals.whatsitid or label_whatsit
638 if vals.processbox then
639     local box = tex.box[vals.processbox]
640     find_line(box, box.head, 1, 0, box.width)
641 end
642 end
643
644 do
645     if token.is_defined('lualineneno') then
646         texio.write_nl('log', "lualineneno: redefining \\lualineneno")
647     end
648     local function_table = lua.get_functions_table()
649     local luafnalloc = luatexbase and luatexbase.new_luafunction
650     and luatexbase.new_luafunction('lualineneno') or #function_table + 1
651     token.set_lua('lualineneno', luafnalloc, 'protected')
652     function_table[luafnalloc] = lualineneno
653 end
654

```

## Format Specific Code

```

656
657 if format == 'optex' then
658

```

To be able to use OpTeX's color mechanism in line numbers the colorizing needs to happen after line numbers are added, so we remove and insert back again the colorizing function from the `pre_shipout_filter` callback.

```

662
663 local colorize = callback.remove_from_callback('pre_shipout_filter', '_colors')
664 callback.add_to_callback('pre_shipout_filter', colorize, '_colors')
665

```

OpTeX only needs to run `\label[toks]` before a destination to label it.

```

667
668 local lbracket = new_tok(string.byte('{'), token.command_id'other_char')
669 local rbracket = new_tok(string.byte('}'), token.command_id'other_char')
670 local label_tok = create('_label')
671 make_label = function(label)
672     runtoks(function()
673         put_next({rbracket})
674         put_next(label)
675         put_next({label_tok,lbracket})
676     end)
677 end

```



```

678
679 elseif latex then
680

```

If the luacolor package is loaded, colorizing must happen after line numbers are added to be able to color them.

```

684
685     luatexbase.declare_callback_rule('pre_shipout_filter',
686         'lualineno.shipout', 'before', 'luacolor.process')
687

```

Since L<sup>A</sup>T<sub>E</sub>X isn't really shipping out the page box, but a box containing the `\topmargin` and the page box which is shifted with `\moveright`, so it adds an undesired offset in `find_line`, so we mark the page box as anchor.

```

692
693     local attr_num = luatexbase.attributes['lualineno_mark']
694     local replace = string.format([[ \moveright \@themargin \vbox attr %d = %d]],
695         attr_num, lineno_marks.anchor)
696     local find = [[ \moveright \@themargin \vbox]]
697     local patch, num_subs = token.get_macro("@outputpage"):gsub(find, replace)
698

```

Log the success or failure of the patch.

```

700
701     if num_subs > 0 then
702         token.set_macro("@outputpage", patch)
703     else
704         texio.write_nl('log', "lualineno: failed to patch \@outputpage")
705     end
706

```

L<sup>A</sup>T<sub>E</sub>X's `\label`'s creates a whatsit node (`\write`), so we temporarily box the label to fetch this node, and add it to the list.

```

709
710     local label_tok = create('label')
711     make_label = function(label, list, n)
712         runtoks(function()
713             put_next({rbrace,rbrace})
714             put_next(label)
715             put_next({hbox, lbrace, label_tok,lbrace})
716             local label_node = scan_list()
717             list = insert_after(list,n,node_copy(label_node.head))
718             node_flush(label_node)
719         end)
720     end
721
722 end

```

## 5.2 OpT<sub>E</sub>X package

The OpT<sub>E</sub>X package does not contain much. It is mainly here for the documentation, or in case someone prefers to type `\load[lualineno]` instead of `\directlua{require('lualineno')}`.

```

3 \directlua{require('lualineno')}
4 \addto\resetattns{\lualineno{unset}}

```

## 5.3 L<sup>A</sup>T<sub>E</sub>X package

The L<sup>A</sup>T<sub>E</sub>X package mostly contains a patch to `breqn` so its equation numbers are recognized.

```

1 \ProvidesPackage
2 {lualineno} [2026-06-25 v0.2
3     Line numbering in LuaTeX]
4
5 \directlua{require('lualineno')}

```

```

6 \AddToHook{build/page/reset}{\lualineno{unset}}
7
8 \AddToHook{package/breqn/after}{%
9   \directlua{
10     local replace = "\csstring\\csstring\\directlua {
11       local box = tex.getbox('EQ@numbox')
12       if box then
13         local n = node.copy_list(box)
14         n.subtype = 7
15         node.write(n)
16       end}"
17     local find = "\csstring\\csstring\\copy \csstring\\csstring\\EQ@numbox "
18     local function patch_breqn(macro)
19       local patch, success = token.get_macro(macro):gsub(find, replace)
20       if success > 0 then
21         token.set_macro(macro, patch)
22       else
23         texio.write_nl('log', "lualineno: failed to patch
24           \csstring\\csstring\\" .. macro .. " (breqn)")
25       end
26     end
27     patch_breqn("eq@typeset@RShifted")
28     patch_breqn("eq@typeset@LShifted")
29     patch_breqn("eq@typeset@rightnumber")
30     patch_breqn("eq@typeset@leftnumber")
31   }}
32
33
34 \endinput

```