Numbering lines in LuaTEX

Version: 0.1, 2026-02-12

Udi Fogiel, 2026

The `lualineno` module provides flexible line numbering for LuaTEX-based formats (LuaLATEX, OpTEX, and Plain LuaTEX).

Table of contents

line

1 Loading `lualineno` . . . 25
2 User Interface . . . 30
2.1 Defining a Lineno . . . 48
2.2 Notes For Plain Users . . . 73
3 Examples . . . 88
3.1 Number Lines Across the Document . . . 89
3.2 Number Lines Per Page . . . 119
3.3 Number Two Column Layout . . . 150
3.4 Modulo . . . 195
3.5 Reference a Line . . . 226
3.6 Link to a Line . . . 257
3.7 Ignore Line Numbers When Copying Text . . . 294
3.8 The `offset` and `anchor` Keys . . . 329
4 Callbacks . . . 354
5 Implementation . . . 369
5.1 Lua module . . . 370
5.2 OpTEX package . . . 1078
5.3 LATEX package . . . 1085

1 Loading `lualineno`

To load the package you can use

LATEX `\usepackage{lualineno}`

OpTEX `\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 49 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 71 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. Should be used after a glyph (but it does not have to be right after it).

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`, `colattr` and `markattr`, which each accept an integer (the attribute number). 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. LaTeX already does that by default, in OpTeX we need to use the `\global` prefix.

LaTeX

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

OpTeX

```
\newcount\lineno
\lualineno
  {
    define =
      {
        name = default
        toks = {\global\advance\lineno by 1}
        left = {\thefontsize[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. OpTeX already does that by default, in LaTeX we need to reset the counter per page.

LaTeX

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

OpTeX

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

3.3 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 = {\thefontsize[5]%
                 \the\lineno\kern.8em}
      }
    define =
      {
        name = default
        column = 2
        toks = {\global\advance\lineno by 1}
        right = {\kern.8em\thefontsize[5]%
                 \the\lineno}
      }

    set = default
  }
```

3.4 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
                \thefontsize[5]
                \the\lineno\kern.8em\fi}
      }
    set = default
  }
```

3.5 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 = {\thefontsize[5]%
                  \wlabel{\the\lineno}%
                  \the\lineno\kern.8em}
      }
  }
```

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

3.6 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. There might be a better way though.

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 = {\AssignSocketPlug
                {refstepcounter/target}{noop}%
                \refstepcounter{lineno}%
                \AssignSocketPlug
                {refstepcounter/target}{hyperref}}
        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}
      }
  }
```

3.7 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
  }
```

3.8 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

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 OpTeX 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 LaTeX 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.

lualineno.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)`)

lualineno.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 tail = node.tail
48 local get_props = node.getproperty
49 local set_props = node.setproperty
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
```

```
local node_copy = node.copy
local base_kern = node.new('kern', 'user')
local line_sub, eq_sub, align_sub, box_sub
local ignored_subtypes = {}
for k,v in pairs(node.subtypes("hlist")) do
    if v == "line" then line_sub = k end
    if v == "alignment" then align_sub = k end
    if v == "box" then box_sub = k end
    if v == "equation" then eq_sub = k end
    if v == "equationnumber" then ignored_subtypes[k] = true end
    if v == "mathchar" then ignored_subtypes[k] = true end
end
local displayalign_sub = #node.subtypes("hlist") + 1
for k,v in pairs(node.subtypes("vlist")) do
    if v == "vextensible" then ignored_subtypes[k] = true end
    if v == "vdelimiter" then ignored_subtypes[k] = true end
end

local setattribute = tex.setattribute
local texerror = tex.error
local texnest = tex.nest
local format = tex.formatname

```

The module currently works with OpTEX, LATEX or Plain.

lualineno.lua

```

local optex, latex, plain
if format:find("optex") then
    optex = true
elseif format:find("latex") then
    latex = true
elseif format == "luatex" or
       format == "luahbtex" or
       format:find("plain")
then
    plain = true
end
if not (optex or latex or plain) then
    error("lualineno: The format " .. format .. " is not supported\n\n" ..
             "Use OpTeX, LuaLaTeX or Plain.")
end

local lineno_types = { }
local lineno_attrs = { }
local LINENO_NUMBER = 0x1
local LINENO_RECURSE = 0x2
local lineno_marks = {
    anchor = 1,
    processed = 2,
    display = 3,
}
local type_attr = luatexbase and luatexbase.new_attribute('lualineno_type') or 0
local col_attr = luatexbase and luatexbase.new_attribute('lualineno_col') or 1
local mark_attr = luatexbase and luatexbase.new_attribute('lualineno_mark') or 2
local unset_attr = -0x7FFFFFFF

```

We use the `luakeyval` module for the user interface

lualineno.lua

```

local keyval = require('luakeyval')
local scan_choice = keyval.choices
local scan_bool = keyval.bool
local process_keys = keyval.process
local messages = {
    error1 = "lualineno: Wrong syntax in \\lualineno",
    value_forbidden = 'lualineno: The key "%s" does not accept a value',
    value_required = 'lualineno: The key "%s" requires a value',
}

```

Numbering Lines

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

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

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

lualineno.lua

```
local function number_line(line, line_type, offset, width, dir)

   local head = line.head
   local is_offset = line_type['offset']

```

In case LaTeX is used without the luacolor package, we add an additional group to make the boxes color safe. Since `token.scan_string` is ran in horizontal mode, the default box direction is `\textdirection`, and this can be pretty random at shipout time, so an explicit direction is specified. Currently LTR is used, if someone ask an option can be added, but you can always use `\hbox bidr 1 {}` inside the box.

lualineno.lua

```
    put_next({rbrace, rbrace})
    put_next(line_type.left)
    put_next({hbox,b,d,i,r,zero,lbrace,lbrace})
    put_next({rbrace, rbrace})
    put_next(line_type.right)
    put_next({hbox,b,d,i,r,zero,lbrace,lbrace})

```

to make sure "right" always means right, we check the line direction.

lualineno.lua

```
    local end_box, start_box
    if line.dir == "TLT" then
        end_box = scan_list()
        start_box = scan_list()
    else
        start_box = scan_list()
        end_box = scan_list()
    end

```

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

lualineno.lua

```
    local start_kern_width = 0
    local end_kern_width = 0
    if is_offset then
        if line.dir == dir then
            start_kern_width = offset
            end_kern_width = width - line.width - offset
        else
            start_kern_width = width - line.width - offset
            end_kern_width = offset
        end
    end

    if start_box.head then
        if start_kern_width ~= 0 then
            local start_kern = node_copy(base_kern)
            start_kern.kern = start_kern_width
            head = insert_before(head,head,start_kern)
        end
        head = insert_before(head,head,start_box)
```

```
        start_kern_width = -start_box.width - start_kern_width
        if start_kern_width ~= 0 then
            local start_kern = node_copy(base_kern)
            start_kern.kern = start_kern_width
            head = insert_before(head,head,start_kern)
        end
    else
        node_flush(start_box)
    end
    if end_box.head then
        if end_kern_width ~= 0 then
            local end_kern = node_copy(base_kern)
            end_kern.kern = end_kern_width
            head = insert_after(head,tail(head),end_kern)
        end
        head = insert_after(head,tail(head),end_box)
    else
        node_flush(end_box)
    end
    line.head = head
end

local lineno_callbacks
if luatexbase then
    luatexbase.create_callback('lualineno.pre_numbering', 'simple', false)
    luatexbase.create_callback('lualineno.numbering', 'exclusive', number_line)
    luatexbase.create_callback('lualineno.post_numbering', 'simple', false)
    luatexbase.add_to_callback('lualineno.pre_numbering', function(_, line_type)
        runtoks(function() put_next(line_type['toks']) end)
    end, 'lualineno.runtoks')
    local call_callback = luatexbase.call_callback
    lineno_callbacks = function(line, line_type, offset, width, dir)
        call_callback('lualineno.pre_numbering', line, line_type, offset, width, dir)
        call_callback('lualineno.numbering', line, line_type, offset, width, dir)
        call_callback('lualineno.post_numbering', line, line_type, offset, width, dir)
    end
else
    lineno_callbacks = function(line, line_type, offset, width, dir)
        runtoks(function() put_next(line_type['toks']) end)
        number_line(line, line_type, offset, width, dir)
    end
end

```

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.

lualineno.lua

```

local function real_box(list)
    for n, id, sb in traverse(list) do
        if id == glyph_id then
            return true
        elseif (id == hlist_id or id == vlist_id) and not ignored_subtypes[sb] then
            if real_box(n.list) then
                return true
            end
        end
    end
    return false
end

```

If the first thing (that we care about) in a line is a glyph we simply number it, if it is an hlist we keep looking inside for glyphs and if it is a vlist we add the shift to the offset and go back to finding lines in that vlist.

lualineno.lua

```

```

```lua
local function real_line(list, parent, offset)
    for n, id, sb in traverse(list) do
        if id == glyph_id then
            return true
        elseif id == vlist_id and not ignored_subtypes[sb] and real_box(n.list) then
            return n, offset + rangedimensions(parent, list, n)
        elseif id == hlist_id and not ignored_subtypes[sb] and real_box(n.list) then
           offset = offset + rangedimensions(parent, list, n)
           return real_line(n.list, n, offset)
        end
    end
    return false
end

```

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.

lualineno.lua

```lua
local find_line
find_line = function(parent, list, column, offset, width)

    if get_attribute(parent, mark_attr) == lineno_marks.processed then return end
    set_attribute(parent, mark_attr, lineno_marks.processed)

```

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

lualineno.lua

```lua

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

```

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

lualineno.lua

```lua

        if id ~= hlist_id then
            if not n.list then goto continue end
            local new_offset, new_width = offset, width
            local new_col = get_attribute(n, col_attr)
            if get_attribute(n, mark_attr) == lineno_marks.anchor or new_col then
                new_offset, new_width = 0, n.width
            elseif parent_is_vlist then
                new_offset = new_offset + n.shift
            end
            find_line(n, n.list, new_col or column, new_offset, new_width)
            goto continue
        end

```

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.

lualineno.lua

```lua

        local line_attr = n.head and get_attribute(tail(n.head), type_attr)
        local line_type = line_attr and lineno_types[line_attr] and lineno_types[line_attr][column]
        local flag
        if sb == align_sub and get_attribute(n,mark_attr) == lineno_marks.display then
            flag = line_type and line_type[displayalign_sub]
        else
            flag = line_type and line_type[sb]
        end

```

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

lualineno.lua

```lua

```

```
        local should_number = flag and (flag & LINENO_NUMBER) ~= 0 or false
        local should_recurse = flag and (flag & LINENO_RECURSE) ~= 0 or true
        if not (should_number or should_recurse) then
            goto continue
        end

```

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.

lualineno.lua

```

        if should_number and not should_recurse then
            if real_box(n.list) then
                local new_offset = parent_is_vlist and (offset + n.shift) or offset
                lineno_callbacks(n, line_type, new_offset, width, parent.dir)
            end
            goto continue
        end

```

If `real_line` returned a `new_offset`, the first thing encountered in the line is a `vlist`, so we need to find lines inside of that `vlist` as well. As before offset and width might need to be updated. If we encounter a column, maybe this line contains more columns, so we number the first one and keep looking for more.

lualineno.lua

```

        local m, new_offset = real_line(n.head, n, offset)
        local new_width = width
        if new_offset then
            local new_col = get_attribute(m, col_attr)
            if get_attribute(m, mark_attr) == lineno_marks.anchor or new_col then
                new_offset, new_width = 0, m.width
            elseif parent_is_vlist then
                new_offset = new_offset + n.shift
            end
            find_line(m, m.head, new_col or column, new_offset, new_width)
            if new_col then
                find_line(n, n.head, new_col, new_offset, width)
            end
            goto continue
        end

        if not (m and should_number) then goto continue end

```

A line is found! update the offset and number it.

lualineno.lua

```

        local new_offset = parent_is_vlist and (offset + n.shift) or offset
        lineno_callbacks(n, line_type, new_offset, width, parent.dir)

        ::continue::
    end
end

if not plain then
    luatexbase.add_to_callback('pre_shipout_filter', function(box)
        find_line(box, box.list, 1, 0, box.width)
        return true
    end, 'lualineno.shipout')
end

```

Check if inside display for display alignment

lualineno.lua

```

if luatexbase then
    luatexbase.add_to_callback("buildpage_filter", function(info)
        if info == "before_display" then
            setattribute(mark_attr,lineno_marks.display)
        elseif info == "after_display" then
            setattribute(mark_attr,unset_attr)
```

```
        end
    end, "lualineno.indisplay")
end

```

Anchoring numbers to a box

lualineno.lua

```

local function mark_last_vlist(n)
    local current = n
    while current do
        if current.id == vlist_id then
            set_attribute(current, mark_attr, lineno_marks.anchor)
            return true
        elseif current.id == hlist_id then
            if mark_last_vlist(tail(current.list)) then return true end
        end
        current = current.prev
    end
    return false
end

```

labels

lualineno.lua

```

local make_label
local function find_label(line)
    local list = line.list
    for n in traverse(list) do
        if n.id == glyph_id then
            local props = get_props(n)
            if props then
                local label = props.lualineno
                if label then
                    make_label(label, list, n)
                end
            end
        elseif n.list then
            find_label(n)
        end
    end
end

local function label_last_glyph(m, tokens)
    if optex then
        luatexbase.add_to_callback('lualineno.pre_numbering', find_label, 'lualineno.labels')
    elseif latex then
        luatexbase.add_to_callback('lualineno.post_numbering', find_label, 'lualineno.labels')
    end
    label_last_glyph = function(n, toks)
        local current = n
        while current do
            if current.id == glyph_id then
                local props = get_props(current)
                if not props then
                    props = { }
                    set_props(current,props)
                end
                props.lualineno = toks
                return true
            elseif current.list then
                if label_last_glyph(tail(current.list), toks) then
                    return true
                end
            end
            current = current.prev
        end
        return false
    end
```

```
457     return label_last_glyph(m, tokens)
458 end
459
```

809 ## User Interface

810 This section describes the definition of the one macro exposed to the end user. It is based on the luakeyval
811 module.

lualineno.lua

```
464
465 local defaults = {
466     toks = { },
467     left = { },
468     right = { },
469     box = {number = true, recurse = true},
470     alignment = {number = true, recurse = true},
471     displayalignment = {number = true, recurse = true},
472     equation = {number = true, recurse = true},
473     line = {number = true, recurse = true},
474     offset = true,
475     column = 1,
476 }
477
478 local inner_keys = {
479     number = {scanner = scan_bool, default = true},
480     recurse = {scanner = scan_bool, default = true}
481 }
482
483 local defaults_keys = {
484     toks = {scanner = scan_toks},
485     left = {scanner = scan_toks},
486     right = {scanner = scan_toks},
487     box = {scanner = process_keys, args = {inner_keys, messages}},
488     alignment = {scanner = process_keys, args = {inner_keys, messages}},
489     displayalignment = {scanner = process_keys, args = {inner_keys, messages}},
490     equation = {scanner = process_keys, args = {inner_keys, messages}},
491     line = {scanner = process_keys, args = {inner_keys, messages}},
492     offset = {scanner = scan_bool},
493     column = {scanner = scan_int}
494 }
495
496 local function set_defaults()
497     local vals = process_keys(defaults_keys,messages)
498     for k,v in pairs(vals) do
499         defaults[k] = v
500     end
501 end
502
503 local define_keys = { }
504 for k,v in pairs(defaults_keys) do
505     define_keys[k] = v
506 end
507 define_keys.name = {scanner = scan_string}
508
509 local function define_lineno()
510     local vals = process_keys(define_keys, messages)
511     local name = vals['name']
512     if not name then
513         texerror("lualineno: Missing name when defining a lineno")
514         return
515     end
516
517     local col = vals['column'] or defaults.column
518     lineno_attrs[name] = lineno_attrs[name] or #lineno_types + 1
519     local i = lineno_attrs[name]
520     lineno_types[i] = lineno_types[i] or {}
521     lineno_types[i][col] = lineno_types[i][col] or {}
522
523     local c = lineno_types[i][col]
524
525     local function store_type(key, subtype_id)
```

```
        local setting = vals[key] or defaults[key]
        local flags = 0
        if setting.number then flags = flags | LINENO_NUMBER end
        if setting.recurse then flags = flags | LINENO_RECURSE end
        c[subtype_id] = flags
    end

    store_type('box', box_sub)
    store_type('alignment', align_sub)
    store_type('equation', eq_sub)
    store_type('line', line_sub)
    store_type('displayalignment', displayalign_sub)

    c.toks = vals.toks or defaults.toks
    c.left = vals.left or defaults.left
    c.right = vals.right or defaults.right
    if vals.offset ~= nil then
        c.offset = vals.offset
    else
        c.offset = defaults.offset
    end
end

local lualineno_keys = {
    set = {scanner = scan_string},
    unset = { default = true },
    define = {scanner = function() return true end, func = define_lineno},
    defaults = {scanner = function() return true end, func = set_defaults},
    anchor = { default = true },
    label = {scanner = scan_toks, args = {false, true}},
    typeattr = {scanner = scan_int},
    colattr = {scanner = scan_int},
    markattr = {scanner = scan_int},
    processbox = {scanner = scan_int},
}

local function lualineno()
    local saved_endlinechar = tex.endlinechar
    tex.endlinechar = 32
    local vals = process_keys(lualineno_keys,messages)
    tex.endlinechar = saved_endlinechar
    if vals.set then
        local attr = lineno_attrs[vals.set]
        if attr then
          setattribute(type_attr, attr)
        else
          texerror("lualineno: type '" .. vals.set .. "' undefined")
        end
    end
    if vals.unset then
        setattribute(type_attr, unset_attr)
    end
    if vals.anchor then
        for i=texnest.ptr,0,-1 do
            if mark_last_vlist(texnest[i].tail) then return end
        end
    end
    if vals.label then
        for i=texnest.ptr,0,-1 do
            if label_last_glyph(texnest[i].tail, vals.label) then return end
        end
    end
    type_attr = vals.typeattr or type_attr
    col_attr = vals.colattr or col_attr
    mark_attr = vals.markattr or mark_attr
    if vals.processbox then
        local box = tex.box[vals.processbox]
        local col = get_attribute(box, col_attr)
        find_line(box, box.head, col or 1, 0, box.width)
    end
```

```
end

do
  if token.is_defined('lualineno') then
      texio.write_nl('log', "lualineno: redefining \\lualineno")
  end
  local function_table = lua.get_functions_table()
  local luafnalloc = luatexbase and luatexbase.new_luafunction
    and luatexbase.new_luafunction('lualineno') or #function_table + 1
  token.set_lua('lualineno', luafnalloc, 'protected')
  function_table[luafnalloc] = lualineno
end

```

Format Specific Code

lualineno.lua

```

if format == 'optex' then

```

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.

lualineno.lua

```

    local colorize = callback.remove_from_callback('pre_shipout_filter', '_colors')
    callback.add_to_callback('pre_shipout_filter', colorize, '_colors')

```

This is the patch for `\beginmulti` in order mark the columns boxes. For each box we assign an attribute with a value according to the column number.

lualineno.lua

```

    local replace = table.concat({
      "\\_directlua{",
      "local column = tex.splitbox(6, tex.dimen[1], 'exactly') ",
      "local num = tex.count['_tmpnum'] ",
      "local attr = luatexbase.attributes['lualineno_col'] ",
      "node.set_attribute(column, attr, num) ",
      "node.write(column) ",
      "}",
    })
    local find = [[\_vsplit 6 to\_dimen 1 ]]
    local patch, success = token.get_macro("_createcolumns"):gsub(find, replace)

```

Log the success or failure of the patch

lualineno.lua

```

    if success > 0 then
        token.set_macro("_createcolumns", patch)
    else
        texio.write_nl('log', "lualineno: failed to patch \\_createcolumns")
    end

```

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

lualineno.lua

```

    local lbracket = new_tok(string.byte('['), token.command_id'other_char')
    local rbracket = new_tok(string.byte(']'), token.command_id'other_char')
    local label_tok = create('_label')
    make_label = function(label)
        runtoks(function()
            put_next({rbracket})
            put_next(label)
            put_next({label_tok,lbracket})
        end)
    end

elseif latex then

```

Here we mark the columns according to `\if@firstcolumn`

lualineno.lua

```
660
661   local true_mode do
662     local prefix = '@lua^line&no_'
663     while token.is_defined(prefix .. 'iftrue') do
664       prefix = prefix .. '@lua^line&no_'
665     end
666     tex.enableprimitives(prefix,{'iftrue'})
667     local tok = create(prefix .. 'iftrue')
668     true_mode  = tok.mode
669   end
670
671
672   luatexbase.add_to_callback('pre_output_filter', function()
673       if create('if@firstcolumn').mode == true_mode then
674          setattribute(col_attr, 1)
675       else
676          setattribute(col_attr, 2)
677       end
678       return true
679   end, 'lualineno.mark_columns')
680   luatexbase.add_to_callback('buildpage_filter', function(info)
681       if info == 'after_output' then
682           setattribute(col_attr, unset_attr)
683       end
684   end, 'lualineno.mark_columns')
685
```

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

lualineno.lua

```
689
690   luatexbase.declare_callback_rule('pre_shipout_filter',
691        'lualineno.shipout', 'before', 'luacolor.process')
692
```

Since LaTeX 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`.

lualineno.lua

```
697
698   local attr_num = luatexbase.attributes['lualineno_mark']
699   local replace = string.format([[\moveright \@themargin \vbox attr %d = -1]], attr_num)
700   local find = [[\moveright \@themargin \vbox]]
701   local patch, num_subs = token.get_macro("@outputpage"):gsub(find, replace)
702
```

space controls are changed as well to avoid a bug in `token.set_macro`

lualineno.lua

```
704
705   find = [[\catcode `\ 10\relax \catcode `\   10\relax]]
706   replace = [[\catcode 32=10\relax \catcode 9=10\relax]]
707   patch, num_subs = patch:gsub(find, replace)
708
```

Log the success or failure of the patch

lualineno.lua

```
710
711   if num_subs > 0 then
712       token.set_macro("@outputpage", patch)
713   else
714       texio.write_nl('log', "lualineno: failed to patch \\@outputpage")
715   end
716
```

LaTeX's `\label`'s creates a whatsit node (`\write`), so we temporarily box the label to fetch this node, and add it to the list.

lualineno.lua

```
719
720   local label_tok = create('label')
721   make_label = function(label, list, n)
```

```
        runtoks(function()
            put_next({rbrace,rbrace})
            put_next(label)
            put_next({hbox, lbrace, label_tok,lbrace})
            local label_node = scan_list()
            list = insert_after(list,n,node_copy(label_node.head))
            node_flush(label_node)
        end)
    end

end
```

5.2 OpTEX package

The OpTEX 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')}`.

lualineno.opm

```
\_codedecl \lualineno {Line numbering <0.1, 2026-02-12>}

\_directlua{require('lualineno')}
\addto\_resetattrs{\lualineno{unset}}
```

5.3 LATEX package

The LATEX package mostly contains patches to other packages to mark the column boxes.

lualineno.sty

```
\ProvidesPackage
  {lualineno} [2026-02-12 v0.1
  Line numbering in LuaTeX]

\directlua{require('lualineno')}
\AddToHook{build/page/reset}{\lualineno{unset}}

\AddToHook{package/multicol/after}{%
  \directlua{
      local replace = "\csstring\\\csstring\\directlua {
        local right_box = token.create('mult@rightbox').index
        local col_attr = luatexbase.attributes['lualineno_col']
        local i = right_box
        local last = tex.count['doublecol@number'] - 2
        local column = 0
        while i < last do
            i = i + 2
            column = column + 1
            tex.box[i][col_attr] = column
        end
        tex.box[right_box][col_attr] = column + 1}\csstring\\\csstring\\mc@align@columns"
      local find = "\csstring\\\csstring\\mc@align@columns"
      local patch, success = token.get_macro("page@sofar"):gsub(find, replace)
      if success > 0 then
          token.set_macro("page@sofar", patch)
      else
          texio.write_nl('log', "lualineno: failed to patch
               \csstring\\\csstring\\page@sofar (multicol)")
      end}}

\AddToHook{package/balance/after}{%
  \expandafter\renewcommand\expandafter
  \@BAlancecol\expandafter{\@BAlancecol\directlua{
      local sec_col = token.create('@outputbox').index
      local first_col = token.create('@leftcolumn').index
      local col_attr = luatexbase.attributes['lualineno_col']
      tex.box[sec_col][col_attr] = 2
      tex.box[first_col][col_attr] = 1}}}

\AddToHook{package/flushend/after}{%
    \directlua{
```

```
      local replace = "\csstring\\\csstring\\set@outputbox@with@footnote@and@float
        \csstring\\\csstring\\fi \csstring\\\csstring\\directlua {
          local sec_col = token.create('@outputbox').index
          local first_col = token.create('@leftcolumn').index
          local col_attr = luatexbase.attributes['lualineno_col']
          tex.box[sec_col][col_attr] = 2
          tex.box[first_col][col_attr] = 1}"
      local find = "\csstring\\\csstring\\set@outputbox@with@footnote@and@float
        \csstring\\\csstring\\fi "
      local patch, success = token.get_macro("last@outputdblcol"):gsub(find, replace)
      if success > 0 then
          token.set_macro("last@outputdblcol", patch)
      else
          texio.write_nl('log', "lualineno: failed to patch
            \csstring\\\csstring\\last@outputdblcol (flushend)")
      end}}

\AddToHook{package/ltxgrid/after}{%
\def\box@column#1{%
 \ltxgrid@info@sw{\class@info{\string\box@column\string#1}}{}%
 \raise\topskip
 \hb@xt@\columnwidth\bgroup
  \dimen@\ht#1\@ifdim{\dimen@>\@colht}{\dimen@\@colht}{}%
  \count@\vbadness\vbadness\@M
  \dimen@ii\vfuzz\vfuzz\maxdimen
  \ltxgrid@info@sw{\saythe\@colht\saythe\dimen@}{}%
  \vtop attr \directlua{tex.print(luatexbase.attributes['lualineno_col'])}
        = \pagegrid@cur to\dimen@\bgroup
   \hrule\@height\z@
   \unvbox#1%
   \raggedcolumn@skip
  \egroup
  \vfuzz\dimen@ii
  \vbadness\count@
  \hss
 \egroup
}}

\AddToHook{package/breqn/after}{%
    \directlua{
      local replace = "\csstring\\\csstring\\directlua {
        local box = tex.getbox('EQ@numbox')
        if box then
          local n = node.copy_list(box)
          n.subtype = 7
          node.write(n)
        end}"
      local find = "\csstring\\\csstring\\copy \csstring\\\csstring\\EQ@numbox "
      local function patch_breqn(macro)
        local patch, success = token.get_macro(macro):gsub(find, replace)
        if success > 0 then
            token.set_macro(macro, patch)
        else
            texio.write_nl('log', "lualineno: failed to patch
              \csstring\\\csstring\\" .. macro .. " (breqn)")
        end
      end
      patch_breqn("eq@typeset@RShifted")
      patch_breqn("eq@typeset@LShifted")
      patch_breqn("eq@typeset@rightnumber")
      patch_breqn("eq@typeset@leftnumber")
      }}


\endinput
```