1. Baseline grid. This package modifies TEX's page building routine so that all the main baselines are on an equally spaced (by `\baselineskip`) grid. This not only mimicks traditional typesetting, but also results in a more pleasing document.

- In multiple colum documents, where the problem is most visible, it is not uncommon for baselines not to align accross colums, resulting in an unbalanced layout.
- In single page documents the problem is less obvious, but it shows up both in facing pages – where each page can be considered to be one column in a double-column document –, and on the same sheet of paper, if the baselines of the pages on that sheet do not align. This last problem is more visible on more "transparent" paper, but one can readily see it, for example, by looking at page 51 of the TEXbook against light: although the baselines of page 51 and 52 start aligned, towards the end of the page they are completely out of sync.

2. The solution for this problem seems to have eluded all trials so far, and this is not surprising: one needs to change the page building process to make it work. In particular, at every point when TEX is considering the addition of vertical material to the output box, we must filter it to make sure said material would end up aligned to the baseline grid. This is very hard to do (impossible?) in TEX alone, but LuaTEX has enough hooks to make this possible.

3. I will describe the algorithm in more detail later, but the idea is as follows: Firstly, we will keep track of the total height of what was already added to the output box. Let us call that value b_h. Then, we process each element in the TEX recent vertical contribution list.

- If the element is a glue with natural dimension g, we remove all stretch from it, and update $b_h \leftarrow b_h + g$.
- If the element is a horizontal list with height h_h and depth h_d, we insert enough glue g before it to make $(b_h+g+h_h) \bmod x = 0$ (in other words, to align its baseline with the grid), and update $b_h \leftarrow b_h+g+h_d+h_h$.

There are more things we need to do, but the basic idea should be clear: we want to only feed TEX's output box with things that are already aligned to the baseline grid, and give TEX no opportunity to change that by not giving it any room to stretch. One could say we want to make the output box look like an airplane's "economy class" section.

4. Using this package. This package is one part of my TEX macro set, and has to be used inside it. This means you must be willing to use Eplain instead of LaTEX, for example. Making this package portable should not be too difficult if you wish to do so – if you do, please publish your work so others can use it.

If you still want to use this, clone my `texmf` tree from GitHub, at `https://github.com/ccrusius/texmf`, make sure TEX can find it, and add the following to your document's preamble. The call to `\ccgridsetup` is to make sure everything is properly arranged: it is already called once when you include the package, but if your preamble changes things after that `ccgrid` may have to recompute some values.

⟨ Enabling the baseline grid in your document 4 ⟩ ≡

```
\input ccgrid
% ... rest of preamble ...
\ccgridsetup
```

5. The package files.

6. ⟨ `*{ccgrid.lua}` 6 ⟩ ≡

```
local exports = {}
⟨ Lua global variables 11 ⟩
⟨ Lua functions 14 ⟩
⟨ Building a gridded page 31 ⟩
return exports
```

7. ⟨ `*{ccgrid.tex}` 7 ⟩ ≡

```
⟨ TEX package preamble 8 ⟩
⟨ TEX global variables 12 ⟩
⟨ TEX macros 13 ⟩
⟨ Set up grid parameters 20 ⟩
⟨ TEX package postamble 9 ⟩
```

8. ⟨ TEX package preamble 8 ⟩ ≡

```
\input ccbase
\pragmaonce{ccgrid}
\input ccshowbox
\directlua{ccgrid = dofile(kpse.find_file("ccgrid.lua"))}
\makeatletter
```

This code is used in section 7.

9. ⟨ TEX package postamble 9 ⟩ ≡

```
\makeatother
\endinput
```

This code is used in section 7.

10. Debugging. Things did go wrong quite often with this, as baseline gridding is not something TEX was designed to do, so I had to set up a decent enough debugging infrastructure. You will probably not be debugging this package, but the code needs to be here anyway. In order to control what debugging messages are printed, you have to set `ccgrid`'s log level to a suitable value, as follows:

Level	Information
0	Nothing.
1	Adds a baseline grid to every page.
2	Prints the contents of `\box255` every page.
3	Trace page building process.

11. In Lua, the log level is stored in the global `loglevel` variable. We will keep a copy of in in TEX's `\ccgridloglevel` counter. To keep both in sync, you should always change the log level by calling the `\setccgridloglevel` TEX macro.

⟨ Lua global variables 11 ⟩ ≡
```
local loglevel = 0
```
See also sections 21, 30, and 36.

This code is used in section 6.

12. ⟨ TEX global variables 12 ⟩ ≡
```
\newcount\ccgridloglevel\ccgridloglevel=0
```
This code is used in section 7.

13. ⟨ TEX macros 13 ⟩ ≡
```
\def\setccgridloglevel#1{%
  \directlua{ccgrid.setloglevel(#1)}%
  \global\ccgridloglevel=#1}
```
See also sections 16, 17, and 24.

This code is used in section 7.

14. ⟨ Lua functions 14 ⟩ ≡
```
local function setloglevel(x) loglevel = x end
exports["setloglevel"] = setloglevel
```
See also sections 15, 22, 23, 29, 38, 39, and 40.

This code is used in section 6.

15. Most of the debugging messages are printed by the Lua module via a call to the `typeout` function below:

⟨ Lua functions 14 ⟩ +≡
```
local function typeout(lvl,str)
  if loglevel >= lvl then texio.write_nl(str) end
end
```

16. Displaying a baseline grid. To enable the display of a baseline grid at every page, use the macro below. Doing this before a final build is a good idea, as it will quickly tell you whether things are working or not. The macro works by setting the appropriate log level, as described previously.

⟨ TEX macros 13 ⟩ +≡
```
\def\ccgriddraft{\setccgridloglevel{1}}
```

17. To print a baseline grid at every page, we redefine the `\output` routine.

⟨ TEX macros 13 ⟩ +≡
```
\newtoks\ccgrid@prevoutput
\ccgrid@prevoutput=\expandafter{\the\output}
\output={%
  \ifnum\ccgridloglevel>0%
    \setbox0=⟨ Baseline grid box 18 ⟩
    \setbox255=\vbox to\vsize{\vtop to0pt{\box0\vss}\hrule height 0pt\box255}
  \fi
  \the\ccgrid@prevoutput}
```

18. ⟨ Baseline grid box 18 ⟩ ≡
```
\vbox to\vsize{
  ⟨ Baseline grid rule 19 ⟩
  \vskip\topskip
  \cleaders\vbox to\baselineskip{
    ⟨ Baseline grid rule 19 ⟩
    \vfil%
    ⟨ Baseline grid rule 19 ⟩}
  \vfill}
```
This code is used in section 17.

19. ⟨ Baseline grid rule 19 ⟩ ≡
```
\kern-0.2pt\hrule height0.2pt depth0.2pt width\hsize\kern -0.2pt
```
This code is used in section 18.

20. Parameter initialization. A gridded TEX run must have some parameters properly initialized. This includes removing stretch from all known skips and sizing things such as `\vsize` properly. All of this is done in the `\ccgridsetup` macro, which is called automatically when `ccgrid.tex` is read.

⟨ Set up grid parameters 20 ⟩ ≡
```
\def\ccgridsetup{
  ⟨ Remove glue from \baselineskip and set the Lua grid 25 ⟩
  ⟨ Remove glue from other TEX skips 26 ⟩
  ⟨ Set \vsize to a multiple of \baselineskip 27 ⟩
  ⟨ Set \lineskip and \lineskiplimit 28 ⟩
  \raggedbottom% We do our own thing, but let's tell others this is the intent
}
\ccgridsetup
```
This code is used in section 7.

21. The spacing for our grid should be essentially `\baselineskip`, but we need to fix this value at the beginning of the document in a separate variable, since TEX may change it mid-course. In Lua, we keep the value in the `baselineskip` variable, and the user can only change it by first setting `\baselineskip` accordingly, and then calling the TEX `\ccgridseput` macro.

⟨ Lua global variables 11 ⟩ +≡
```
local baselineskip = 0
```

22. ⟨ Lua functions 14 ⟩ +≡
```
local function setgrid(x) baselineskip=x end
exports["setgrid"] = setgrid
```

23. Some basic math has to be performed when setting these parameters, and that is done on the Lua side of things.

⟨ Lua functions 14 ⟩ +≡
```
local function snapdown(x)
  return baselineskip*math.floor(x/baselineskip)
end
exports["snapdown"] = snapdown
local function freeze(x)
  return (ccbase.spstr(ccbase.tosp(x)))..." plus 0pt minus 0pt"
end
exports["freeze"] = freeze
```

24. ⟨ TEX macros 13 ⟩ +≡
```
\def\ccgrid@freeze#1{\directlua{tex.print(ccgrid.freeze("#1"))}}
```

25. ⟨ Remove glue from `\baselineskip` and set the Lua grid 25 ⟩ ≡
```
\global\baselineskip=\ccgrid@freeze{\the\baselineskip}
\directlua{ccgrid.setgrid(ccbase.tosp("\the\baselineskip"))}
\typeout{ccgrid: baselineskip=\the\baselineskip}
```
This code is used in section 20.

26. ⟨ Remove glue from other TEX skips 26 ⟩ ≡

```
\global\topskip=\ccgrid@freeze{\the\topskip}
\global\parskip=\ccgrid@freeze{\the\parskip}
\global\abovedisplayskip=\ccgrid@freeze{\the\abovedisplayskip}
\global\belowdisplayskip=\ccgrid@freeze{\the\belowdisplayskip}
\global\abovedisplayshortskip=\ccgrid@freeze{\the\abovedisplayshortskip}
\global\belowdisplayshortskip=\ccgrid@freeze{\the\belowdisplayshortskip}
```

This code is used in section 20.

27. ⟨ Set `\vsize` to a multiple of `\baselineskip` 27 ⟩ ≡

```
\global\advance\vsize by-\topskip
\global\vsize=\directlua{tex.print(%
  ccbase.spstr(ccgrid.snapdown(ccbase.tosp("\the\vsize"))))}
\global\advance\vsize by\topskip
\typeout{ccgrid: vsize=\the\vsize}
```

This code is used in section 20.

28. ⟨ Set `\lineskip` and `\lineskiplimit` 28 ⟩ ≡

```
\global\lineskip=0pt
\global\lineskiplimit=-0.5\baselineskip
```

This code is used in section 20.

29. There is one last thing to do, which is to make sure our ragged bottom is ragged. TEX `\raggedbottom` command inserts a rather small, for our purposes, "`1fil`" glue at the end of the page. This is easily undone by other infinite glues that may be sprinkled around.

We register a LuaTEX `pre_output_filter` callback and insert a very large glue at the end of every page, hopefully smothering any other glue that may be present. (We also take the opportunity and print the log level 2 output box in the same function.)

⟨ Lua functions 14 ⟩ +≡

```
local function output(head)
  node.insert_after(head,node.slide(head),ccbase.mkglue(0,10000*2^16,3,0,0))
  typeout(2,"OUTPUT BOX:")
  if loglevel >= 2 then ccshowbox.showheadlist(head,0,".") end
  return true
end
callback.register("pre_output_filter",output)
```

30. Building a gridded page. Right before LuaTEX moves material into the main vertical list, it calls a user function registered as a `buildpage_filter` callback. We will use this to massage the material so it will be aligned to the grid when LuaTEX finally inserts it into the vertical list.

To do that, we must keep track of all that was already inserted into that list. Hopefully it was all aligned, but if it was not (for some reason), we must try to start aligning things as soon as possible.

The only thing we need to know about what was already inserted is the total natural height. We will store this in the `haccum` global variable, and refer to it as h_{acc} in the documentation.

⟨ Lua global variables 11 ⟩ +≡
```
local haccum = 0
```

31. When LuaTEX calls the buildpage filter callback, it gives the function the reason why it is being called, as a string. We will build a map from the reasons to the functions that will be called and store it in `buildpage_actions`. We will describe what each means, and what each function does, as we implement them.

⟨ Building a gridded page 31 ⟩ ≡
```
local buildpage_actions = {
  ⟨ Build page actions 33 ⟩
}
```
See also section 32.

This code is used in section 6.

32. Once we have that map, the callback is simple: apart from a debugging statement, it simply executes the appropriate function on the list of potential contributions. Those are found in the global LuaTEX variable `tex.lists.contrib_head`.

⟨ Building a gridded page 31 ⟩ +≡
```
local function buildpage(reason)
  local head = tex.lists.contrib_head
  typeout(3,string.format("BUILDPAGE("..reason..") haccum=%fpt",haccum/2^16))
  local action = buildpage_actions[reason]
  if action then action(head) end
end
callback.register("buildpage_filter",buildpage)
```

33. Let us start with the simplest action. The calling reason is `after_output` when LuaTEX has just finished shipping a page. When this happens, the main vertical material list becomes empty, and all have to do is to reset h_{acc}.

⟨ Build page actions 33 ⟩ ≡
```
after_output = function(head) haccum=0 end,
```
See also sections 35, 37, and 41.

This code is used in section 31.

34. Next, let us define a few no-op actions for debugging purposes:

⟨ Build page no-op action 34 ⟩ ≡
```
function(head)
  if loglevel >= 3 then
    ccshowbox.showheadlist(head,0,"  == .")
  end
end
```
This code is used in section 35.

35. ⟨ Build page actions 33 ⟩ +≡

```
before_display = ⟨ Build page no-op action 34 ⟩,
hmode_par      = ⟨ Build page no-op action 34 ⟩,
new_graf       = ⟨ Build page no-op action 34 ⟩,
vmode_par      = ⟨ Build page no-op action 34 ⟩,
```

36. The `pre_box` call signals that new material will be added via a `box` call. The reason that is important for us is because the `box` call will refer to both old and new material, so we have to use the `pre_box` call to identify where the old material ends, and the new starts. The list seen in a `pre_box` call is the old material, and the list `box` sees is (some) of the old, plus the new. We keep track of where the old material ends in the `lastprebox` variable, which we update in the `pre_box` call.

⟨ Lua global variables 11 ⟩ +≡

```
local lastprebox = nil
```

37. ⟨ Build page actions 33 ⟩ +≡

```
pre_box = function(head) lastprebox=node.slide(head) end,
```

38. So far, we have only done book-keeping activities. Let's start modifying the contents of the contributions list, starting with glues. When we encounter glue, we simply remove its stretch so it does not affect any material on the page, and update h_{acc} to reflect the new height of the page once the glue is inserted.

The `head` argument represents the head of the contributions list, and the `glue` parameter is a pointer to the glue, in that contribution list, we want to modify. The function returns the new head and glue. As an aside, I could probably have replaced the glue with a kern, but either works.

⟨ Lua functions 14 ⟩ +≡

```
local function freeze_glue(head,glue)
  local spec = glue.spec
  local width = spec.width
  haccum = haccum + width
  if spec.stretch == 0 and spec.shrink == 0 then return head, glue end

  local noglue = ccbase.mkglue(glue.spec.width,0,0,0,0)
  head, noglue = node.insert_after(head,glue,noglue)
  head, noglue = node.remove(head,glue)

  -- I think we should free "glue" now, but that makes things crash.
  -- if spec.writable then node.free(glue) end
  return head, noglue
end
```

39. Next, we deal with a horizontal box, or `hlist` in LuaTEX parlance. This is where the real work happens: we want to align all `hlist`s baselines to our grid. In order to do that, we insert enough glue to make that happen.

- First, we update $h_{\text{acc}} \leftarrow h_{\text{acc}} + h$, where h is the box height. This is where the box baseline would be if we left it to TEX alone.
- Then, we compute how much we have to add in skip to make h_{acc} a multiple of the baseline grid. If s is our skip, we want $h_{\text{acc}} + s = b\lceil \frac{h_{\text{acc}}}{b} \rceil$, which gives us a simple expression for s. If the skip is non-zero, we insert the glue and update $h_{\text{acc}} \leftarrow h_{\text{acc}} + s$.
- Now the box is placed correctly, but the material may still continue past the baseline, so we update $h_{\text{acc}} \leftarrow h_{\text{acc}} + d$, where d is the box depth.

The function inputs and outputs are similar to those of `freeze_glue`.

⟨ Lua functions 14 ⟩ +≡
```
local function align_hlist(head,hlist)
  haccum = haccum + hlist.height
  local skip = baselineskip*math.ceil(haccum/baselineskip) - haccum
  if skip > 0 then
    typeout(3,string.format("  ( shifting %fpt )",skip/2^16))
    head = node.insert_before(head,hlist,ccbase.mkglue(skip,0,0,0,0))
    haccum = haccum + skip
  else
    typeout(3,"  ( nop )")
  end
  haccum = haccum + hlist.depth
  return head, hlist
end
```

40. With the last two functions we can go back to building the page. When the callback calling reason is `box`, LuaTEX is inserting horizontal material. When it is `after_display`, LuaTEX has inserted display-mode math. They are both processed by the same function, which goes through the contribution list, aligning any new material – you should recall that the contribution list may contain already processed material, so we have to start from `lastprebox`, which was updated when the callback was called with a `pre_box` reason.

⟨ Lua functions 14 ⟩ +≡
```
local function align_box(head)
  if lastprebox then
    head = lastprebox.next
    lastprebox = nil
  end
  if loglevel>= 3 then ccshowbox.showheadlist(head,0,"  << .") end
  local cur = head
  while cur do
    local actions = { [ccbase.GLUE_TYPE]  = freeze_glue,
                      [ccbase.HLIST_TYPE] = align_hlist }
    local fn = actions[cur.id]
    if fn then head, cur = fn(head,cur) end
    cur = cur.next
  end
  if loglevel >= 3 then ccshowbox.showheadlist(head,0,"  >> .") end
end
```

41. ⟨ Build page actions 33 ⟩ +≡

```
after_display  = align_box,
box            = align_box,
```

CCGRID