Customised LaTeX page layout with LuaTeX

Abstract
The relationship between LaTeX's page layout parameters and the conventional desktop publishing (DTP) model of a page are explored and formulae to map between them are presented. A sample implementation of those formulae in Lua is provided, showing how to achieve customised page layouts in LuaTeX. The placement of crop marks is addressed, and a technique for preparing and adding them to typeset pages is discussed.

1 Introduction

Whilst LaTeX and its wealth of packages and facilities for typesetting are truly superb, I don't think it is unfair to say that it can be quite troublesome to achieve highly customised page layouts which need careful adjustment of the LaTeX parameters to control margins and page size. There are LaTeX packages, such as `geometry.sty`, which help you set the LaTeX layout parameters but sometimes it is nice to have access to the details to fine-tune them as you need to. In this paper, some equations which map LaTeX page layout parameters to the conventional desktop publishing (DTP) model of a page are presented and implemented through a simple Lua script for use with LuaTeX. In writing this paper I have to assume that you have a working installation of LuaTeX and that you know where to save and store the various file types discussed in this article: there is neither time nor space to address those issues here.

2 Problem definition: what are we aiming to do?

Suppose you want to produce a document which has a certain physical printed page width and page height, and you would like to achieve a layout such that your document's page(s) will be horizontally and vertically centred within the area defined by the PDF page size. Of course, the size of the PDF page is also something you want to control. Consider a typical business card which might be 85mm tall and 55mm wide. You want to create this card and have it centred within a PDF page which has, perhaps, 20mm of white space to the left and right of your card, and 10mm above and below, giving a PDF page width of $85 + (2 \times 20) = 125$mm and a height of $55 + (2 \times 10) = 75$mm. See Figure 1.

In addition, commercial printing companies may request that the pages in your PDF document are a certain size; e.g., for use with their imposition software used during preparation of "page impositions".

2.1 Setting the size of the PDF page

LuaTeX is derived from pdfTeX so you set the width and height of the PDF page using registers called `\pdfpagewidth` and `\pdfpageheight`. Returning to our business card example in Figure 1, to set the PDF page size to 125mm wide and 75mm tall we put the commands `\pdfpagewidth 125mm` and `\pdfpageheight 75mm` in our document, not forgetting that you may need to set `\pdfoutput=1`, depending on your TeX setup and distribution.

```
\documentclass[11pt,twoside]{article}
\pdfoutput=1
\pdfpagewidth 125mm
\pdfpageheight 75mm
\begin{document}
Your text here...
\end{document}
```

3 The "DTP design world" layout model

The usual way to think about layouts is, of course, as a series of enclosed boxes. Figure 2 generalises our business card example to show a typical setup for a "book" – but do remember that although I'm using the term "book", you should think of this as any document type. The outermost box is the size of the PDF page as defined by `\pdfpagewidth` and `\pdfpageheight`. Inside the PDF page box is the box defining the physical size of your "book" or document pages. Inside the box of your physical page is a non-printing area for margins which defines the boundary or enclosure for the live text area; i.e., the area in which text or other content will appear. And finally, within the live text area are the boxes that LaTeX uses to place the text on your page – such as the main text area, headers and footers.

4 LaTeX page parameters to achieve your layout

And so we reach the question: how do we define or calculate the LaTeX page parameters to achieve our desired "DTP design world" layout model? Figure 3 superim-

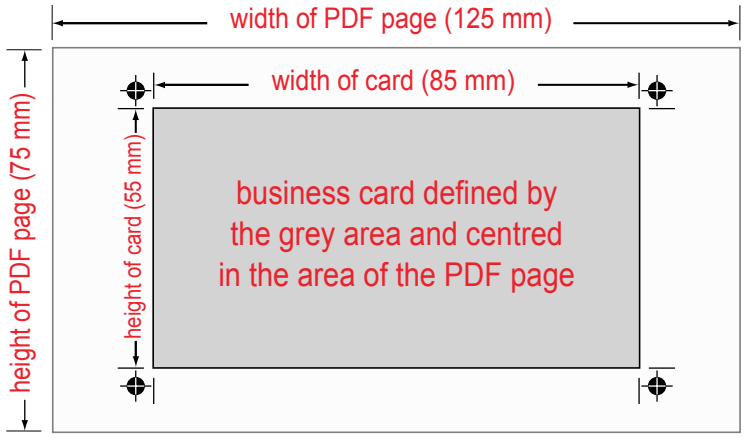

Figure 1. A business card centred within the area defined by the PDF document page. The crop marks define the boundaries of the card within the PDF document area.

poses the two world views: the LaTeX view and that of conventional desktop publishing, showing our page horizontally and vertically centred within a defined PDF document page. Separate graphics are shown for a left-hand page and a right-hand page, which, for left-to-right languages, usually implies books with even page numbers on the left-hand page and odd page numbers on the right-hand page.

Working from Figures 2 and 3 we can now write down some simple formulae to calculate LaTeX page parameters that will achieve our layout. Firstly, some definitions:

$$\begin{aligned} \mathrm{B_{PW}} &= \text{width of the book page} \\ \mathrm{B_{PH}} &= \text{height of the book page} \\ \mathrm{B_{OM}} &= \text{the Book Outer Margin} \\ \mathrm{B_{IM}} &= \text{the Book Inner Margin} \\ \mathrm{B_{TM}} &= \text{the Book Top Margin} \\ \mathrm{B_{BM}} &= \text{the Book Inner Margin} \end{aligned}$$

The following values control centring the book page within the PDF document page size:

$$\Delta\mathrm{X} = \frac{1}{2}(\mathrm{pdfpagewidth} - \mathrm{B_{PW}})$$

$$\Delta\mathrm{Y} = \frac{1}{2}(\mathrm{pdfpageheight} - \mathrm{B_{PH}})$$

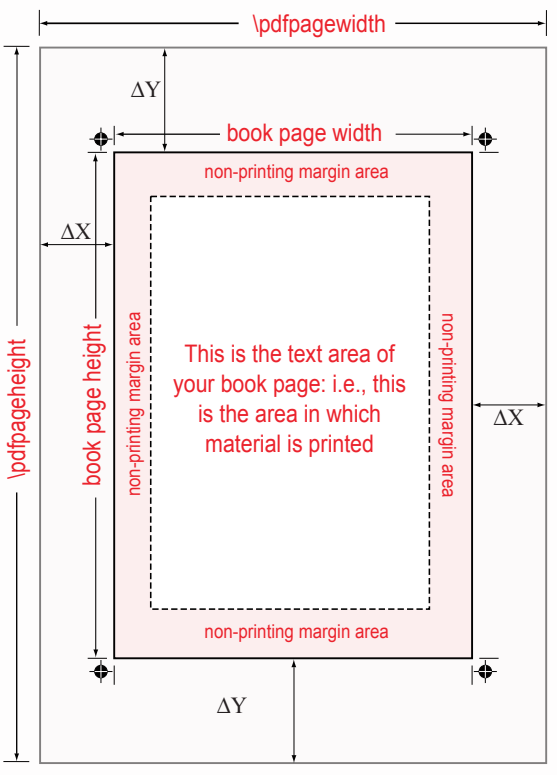

Figure 2. A generalised view of a "book" page: horizontally and vertically centred within the enclosing PDF document page area.

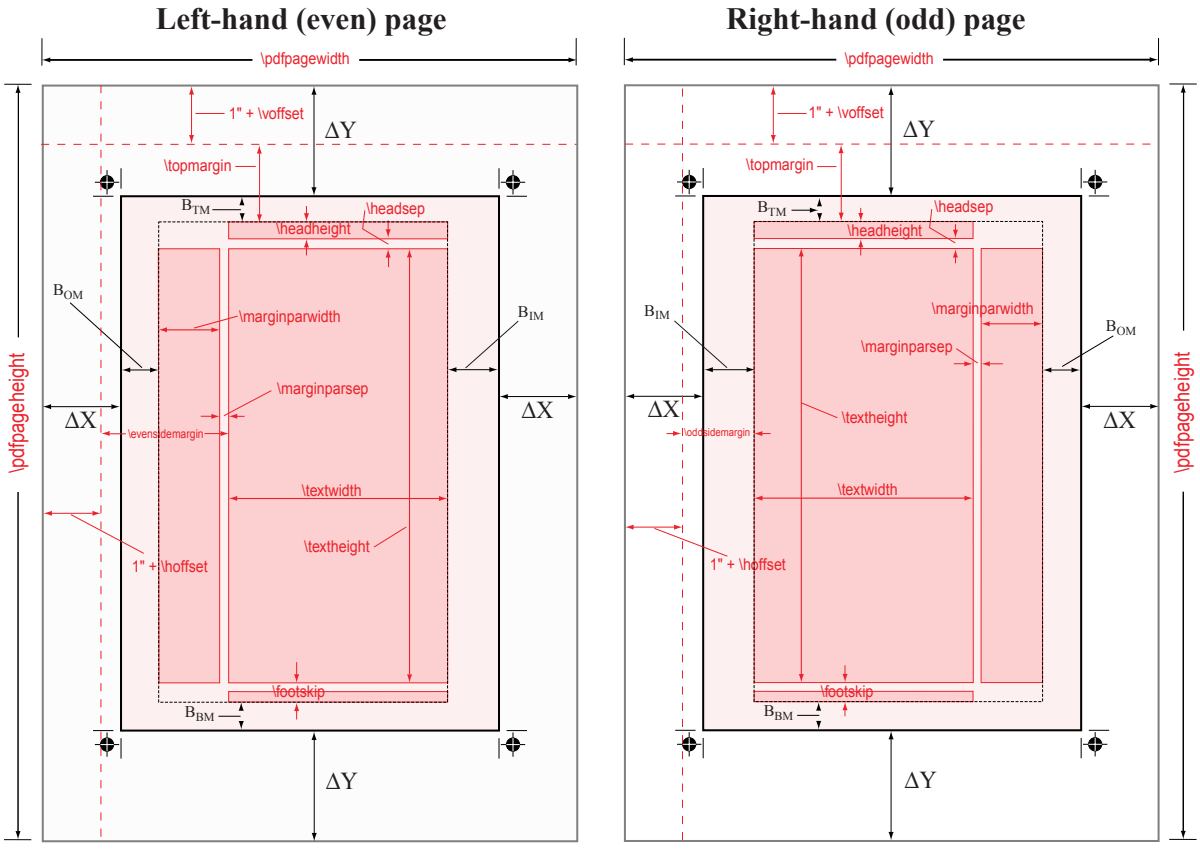

Figure 3. The LaTeX view of a page superimposed onto the conventional "desktop publishing" model to achieve a "book page" which is horizontally and vertically centred within the PDF document area. A separate graphic is shown for left- and right-hand pages.

4.1 Formulae for the width of the PDF document page

Starting with the left-hand (even-numbered) page shown in Figure 3:

$$\begin{aligned}\text{pdfpagewidth} = {} & \Delta X + B_{OM} \\ & + \text{marginparwidth} \\ & + \text{marginparsep} \\ & + \text{textwidth} \\ & + B_{IM} + \Delta X\end{aligned}$$

and

$$\begin{aligned}\text{pdfpagewidth} = {} & 1\text{inch} + \text{hoffset} \\ & + \text{evensidemargin} \\ & + \text{textwidth} \\ & + B_{IM} + \Delta X\end{aligned}$$

For the right-hand (odd-numbered) page shown in Figure 3:

$$\begin{aligned}\text{pdfpagewidth} = {} & \Delta X + B_{IM} \\ & + \text{textwidth} \\ & + \text{marginparsep} \\ & + \text{marginparwidth} \\ & + B_{OM} + \Delta X\end{aligned}$$

and

$$\begin{aligned}\text{pdfpagewidth} = {} & 1\text{inch} + \text{hoffset} \\ & + \text{oddsidemargin} \\ & + \text{textwidth} \\ & + \text{marginparsep} \\ & + \text{marginparwidth} \\ & + B_{OM} + \Delta X\end{aligned}$$

4.2 Formulae for the height of the PDF page

Clearly, the following holds true for left- and right-hand pages:

$$\begin{aligned}\text{pdfpageheight} = {} & 1\text{inch} + \text{voffset} \\ & + \text{topmargin} \\ & + \text{headheight} \\ & + \text{headsep} \\ & + \text{textheight} \\ & + \text{footskip} \\ & + \mathrm{B_{BM}} + \Delta\mathrm{Y}\end{aligned}$$

and

$$1\text{inch} + \text{voffset} + \text{topmargin} = \Delta\mathrm{Y} + \mathrm{B_{TM}}$$

5 Implementation using Lua and LuaTEX

The details above can be implemented in any TEX engine, whether they output PDF directly or you have to go through a DVI-to-PostScript driver. Of course, the direct use of `\pdfpagewidth` and `\pdfpageheight` would be ruled out for non-pdfTEX-based engines and for DVI–PostScript–PDF workflows you'd need to tell your DVI-to-PostScript driver how to set the paper size to the values we are specifying with `\pdfpagewidth` and `\pdfpageheight`. For example, the formulae above are easily programmed into, say, a Perl script or any other scripting language, to output a TEX file which sets the various LaTEX parameters. But for the purposes of this paper the implementation will be in Lua and LuaTEX.

5.1 Implementation overview

The goal is quite straightforward: the formulae above will be turned into a simple Lua script, say `pagecalcs.lua`, which will be run or called using LuaTEX's `\directlua{ }` primitive. `pagecalcs.lua` outputs a LaTEX file (`margins.tex`) which contains some code to define various LaTEX parameters such that your physical page area is centred within the PDF document page area.

5.2 Many choices

A quick inspection of the page formulae shows that the various parameters are all interrelated so the immediate question is which ones do you specify and which ones do you calculate? The answer really depends on your starting point; i.e., what type of document do you want to produce, which parameters do you want to define (assign values to) and which ones you want or need to calculate. The sample implementation, discussed below, is just *one* of the many possible variations. I have chosen the following setup which could be used for a typical business card or journal/book cover:

- set the following LaTEX values to zero: `\hoffset`, `\voffset`, `\marginparsep`, `\headheight`, `\marginparwidth`, `\headsep` and `\footskip`;
- calculate the following LaTEX parameters: `\textwidth`, `\topmargin`, `\oddsidemargin`, `\evensidemargin` and `\textheight`;
- input the "DTP design world" values for the paper (PDF page) size, the size of our physical "book" pages and the various white space margins surrounding the live text area (see Figure 2);
- generalise the margins to consider different values for the white space at the top, bottom, left and right of our live text area, which is quite typical for book designs. Of course, you can set them to be all the same, if you wish.

Clearly, these assumptions are hard-coded into `pagecalcs.lua` and a much more sophisticated implementation would support far more flexibility; but the purpose here is to demonstrate the basic ideas.

5.3 Using Lua

For our Lua script, `pagecalcs.lua`, we will use the following variable names which are rather long but have the benefit of being descriptive.

- `PaperWidth` = the value of `\pdfpagewidth`
- `PaperHeight` = the value of `\pdfpageheight`
- `BookPageWidth` = your document's page width
- `BookPageHeight` = your document's page height
- `BookOuterMargin` = $\mathrm{B_{OM}}$ as defined above
- `BookInnerMargin` = $\mathrm{B_{IM}}$ as defined above
- `BookTopMargin` = $\mathrm{B_{TM}}$ as defined above
- `BookBottomMargin` = $\mathrm{B_{BM}}$ as defined above

In the Lua code below, the variables names for the LaTEX parameters follow their LaTEX counterparts minus the leading '\'. For example,`\textwidth` will be referred to as `textwidth` and so forth. The following listing is `pagecalcs.lua`, just one of many calculation scenarios for one document type based on values we define and values we choose to calculate.

`pagecalcs.lua` defines a Lua function called `calcvals( )` which takes a single argument called `arg`. In Lua-speak `arg` will be a table so that when we call the function `calcvals({...})`, the data in braces `{...}` is passed in as the value of `arg` and will contain a number of key–value pairs. The values we will pass to `calcvals( )` are `PaperWidth`, `PaperHeight`, `BookPageWidth`, `BookPageHeight`, `BookOuterMargin`, `BookInnerMargin`, `BookTopMargin` and `BookBottomMargin`. Each of these values is accessed

pagecalcs.lua

```
function calcvals(arg)
    local OneInch=25.4
    local hoffset=0
    local voffset=0
    local marginparsep=0
    local headheight=0
    local marginparwidth=0
    local headsep=0
    local footskip=0
    local DeltaX = 0.5*(arg.PaperWidth - arg.BookPageWidth)
    local DeltaY = 0.5*(arg.PaperHeight - arg.BookPageHeight)
    local textwidth = arg.PaperWidth -(2*DeltaX) -
                          (arg.BookOuterMargin + arg.BookInnerMargin) -
                          (marginparsep + marginparwidth)
    local topmargin = DeltaY + arg.BookTopMargin -(OneInch + voffset)
    local oddsidemargin =  arg.PaperWidth - (OneInch + hoffset + textwidth +
                          marginparsep +  marginparwidth +
                          arg.BookOuterMargin+ DeltaX)
    local evensidemargin  =  arg.PaperWidth -(OneInch + hoffset + textwidth +
                         arg.BookInnerMargin + DeltaX)
    local textheight =  arg.PaperHeight -(OneInch + voffset + topmargin +
                          headheight + headsep + footskip +
                          arg.BookBottomMargin + DeltaY)
    local marg = assert(io.open("path_to_your_tex_setup/margins.tex","w"))
    marg:write("\\pdfpagewidth="..arg.PaperWidth.."mm\n")
    marg:write("\\pdfpageheight="..arg.PaperHeight.."mm\n")
    marg:write("\\newlength{\\deltax}\n")
    marg:write("\\setlength{\\deltax}{"..DeltaX.."mm}\n")
    marg:write("\\newlength{\\deltay}\n")
    marg:write("\\setlength{\\deltay}{"..DeltaY.."mm}\n")
    marg:write("\\newlength{\\bookpagetopmargin}\n")
    marg:write("\\setlength{\\bookpagetopmargin}{"..arg.BookTopMargin.."mm}\n")
    marg:write("\\newlength{\\bookpageheight}\n")
    marg:write("\\setlength{\\bookpageheight}{"..arg.BookPageHeight.."mm}\n")
    marg:write("\\newlength{\\bookpagebottommargin}\n")
    marg:write("\\setlength{\\bookpagebottommargin}{"..arg.BookBottomMargin.."mm}\n")
    marg:write("\\newlength{\\bookpagewidth}\n")
    marg:write("\\setlength{\\bookpagewidth}{"..arg.BookPageWidth.."mm}\n")
    marg:write("\\newlength{\\bookpageinnermargin}\n")
    marg:write("\\setlength{\\bookpageinnermargin}{"..arg.BookInnerMargin.."mm}\n")
    marg:write("\\newlength{\\bookpageoutermargin}\n")
    marg:write("\\setlength{\\bookpageoutermargin}{"..arg.BookOuterMargin.."mm}\n")
    marg:write("\\setlength{\\hoffset}{"..hoffset.."mm}\n")
    marg:write("\\setlength{\\marginparsep}{"..marginparsep.."mm}\n")
    marg:write("\\setlength{\\marginparwidth}{"..marginparwidth.."mm}\n")
    marg:write("\\setlength{\\textwidth}{"..textwidth.."mm}\n")
    marg:write("\\setlength{\\oddsidemargin}{"..oddsidemargin.."mm}\n")
    marg:write("\\setlength{\\evensidemargin}{"..evensidemargin.."mm}\n")
    -- Vertical parameters
    marg:write("\\setlength{\\voffset}{"..voffset.."mm}\n")
    marg:write("\\setlength{\\headheight}{"..headheight.."mm}\n")
    marg:write("\\setlength{\\headsep}{"..headsep.."mm}\n")
    marg:write("\\setlength{\\footskip}{"..footskip.."mm}\n")
    marg:write("\\setlength{\\textheight}{"..textheight.."mm}\n")
    marg:write("\\setlength{\\topmargin}{"..topmargin.."mm}\n")
    marg:flush()
    marg:close()
return DeltaX, DeltaY, textwidth,topmargin,oddsidemargin,evensidemargin,textheight
end -- function
```

using a standard Lua method for accessing table values, such as `arg.PaperWidth` and `arg.BookPageWidth`. The code also sets some LaTeX parameters to 0 (based on our input assumptions above) and defines `OneInch` as 25.4. Note that we are working with units in mm, just because it is convenient: hence 1 inch is 25.4 mm. Note the following:

- `local marg = assert(io.open("path_to...;`
- `calcvals( )` returns *multiple* values, a standard feature of Lua.

Of course, the output from `pagecalcs.lua`, `margins.tex`, will subsequently be input into our LaTeX document via `\input margins.tex`, hence you will need to output `margins.tex` into a location where your LuaTeX engine can find it.

6 The LaTeX side of things in LuaTeX

We have seen the Lua code to output `margins.tex` but, of course, we need some way to call the `calcvals( )` function defined in `pagecalcs.lua` and pass in the table `arg` containing our various values. We can do this from LuaTeX using one of the new primitives provided by this amazing new engine: `\directlua{...}`. One simple setup is using the standard LaTeX document class `article.cls`, as follows:

```
\documentclass[11pt,twoside]{article}
\input setpage
\setpage{300}{485}{250}{255}{10}{15}{20}{25}
\begin{document}
....
\end{document}
```

Of course, this should all be wrapped up into a proper LaTeX package but, again, my objective is to demonstrate the basic ideas. `\input setpage` inputs a file (`setpage.tex`) which defines a command `\setpage` that takes a number of arguments to define your custom PDF page, document page and margins and uses `\directlua{...}` to call our Lua function `calcvals( )`, which is itself stored in `pagecalcs.lua`. For example, `\setpage{300}{485}{250}{255}{10}{15}{20}{25}` would assign the following values (in mm):

- `PaperWidth` = 300mm
- `PaperHeight` = 485mm
- `BookPageWidth` = 250mm
- `BookPageHeight` = 255mm
- `BookOuterMargin` = $\mathrm{B_{OM}}$ = 10mm
- `BookInnerMargin` = $\mathrm{B_{IM}}$ =15mm
- `BookTopMargin` = $\mathrm{B_{TM}}$ = 20mm
- `BookBottomMargin` = $\mathrm{B_{BM}}$ =25mm

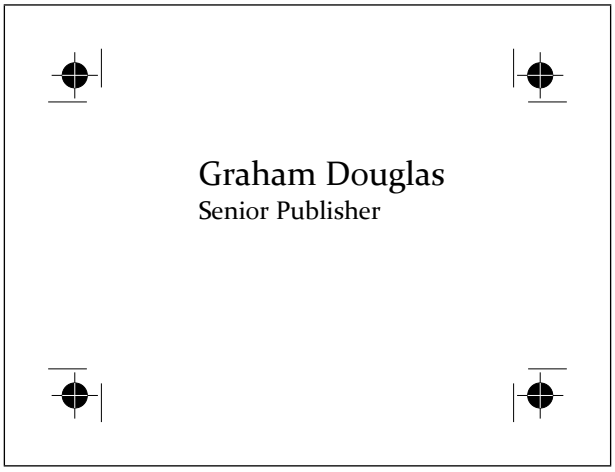

Figure 4. A business card

Here is the code for our file `setpage.tex` which again assumes that `pagecalcs.lua` is saved in a location where your LuaTeX engine can find Lua scripts.

setpage.tex

```
\def\setpage#1#2#3#4#5#6#7#8{%
\directlua {%
  pagecals, loaderror = loadfile("pagecalcs.lua")
  %good code would check the value of loaderror!
  pagecals()
  deltax, deltay, textwidth, topmargin,
  oddsidemargin,evensidemargin,textheight =
    calcvals({PaperWidth=#1, PaperHeight=#2,
              BookPageWidth=#3,
              BookPageHeight=#4, BookOuterMargin=#5,
              BookInnerMargin=#6, BookTopMargin=#7,
              BookBottomMargin=#8})
  pagevars={}
  pagevars["paperwidth"]=#1
  pagevars["paperheight"]=#2
  pagevars["bookpagewidth"]=#3
  pagevars["bookpageheight"]=#4
  pagevars["bookoutermargin"]=#5
  pagevars["bookinnermargin"]=#6
  pagevars["booktopmargin"]=#7
  pagevars["bookbottommargin"]=#8
  pagevars["deltay"]=deltay
  pagevars["deltax"]=deltax
  pagevars["textwidth"]=textwidth
  pagevars["topmargin"]=topmargin
  pagevars["oddsidemargin"]=oddsidemargin
  pagevars["evensidemargin"]=evensidemargin
  pagevars["textheight"]=textheight}
\input margins.tex\relax}
}
```

6.1 A business card

Just by way of an example, Figure 4 shows a business card created by `\setpage{125}{95}{85}{55}{0}{0}{0}{0}`. However, the code to generate the crop marks (see Section 7) at the corners is not shown here.

6.2 Anatomy of setpage.tex

The definition \def\setpage#1#2#3#4#5#6#7#8 is a fairly standard TeX affair to define a macro that takes 8 parameters, but the interesting bit starts with \directlua. Let me just say that, of course, the material in this paper could just as easily be implemented in Perl, or another scripting language, and the TeX engine could make system calls to run the script/interpreter of your choice to output margins.tex. Or, it could be implemented completely outside the TeX engine with the script being run manually. However, the use of LuaTeX's ability to run Lua code with \directlua makes for a very nice integration. For sure, this example does not even begin to indicate the world of possibilities that LuaTeX opens, for that you should visit luatex.org and grab a copy of the latest reference manual. But be warned, LuaTeX is highly addictive, utterly absorbing and quite damaging to other hobbies you may enjoy because it truly opens a whole new universe of typesetting solutions. Now, back to setpage.tex.

In the listing of setpage.tex the following lines are worth some explanation, especially if you are new to Lua or LuaTeX.

```
pagecals, loaderror = loadfile("pagecalcs.lua")
%good code would check the value of loaderror!
pagecals()
deltax, deltay, textwidth,topmargin,
oddsidemargin,evensidemargin,textheight =
   calcvals({PaperWidth=#1, PaperHeight=#2,
             BookPageWidth=#3,
             BookPageHeight=#4, BookOuterMargin=#5,
             BookInnerMargin=#6, BookTopMargin=#7,
             BookBottomMargin=#8})
```

The line pagecals, loaderror = loadfile ... uses Lua's method of "running" code stored in a file; it returns a function (here called pagecals) and an error value (here called loaderror), which you should check. Without going into detail, loadfile(...) "loads a Lua chunk from a file but does not run the chunk. Instead, it only compiles the chunk and returns the compiled chunk as a function." (Google, or see page 63 of *Programming in Lua* by Roberto Ierusalimschy, Second Edition, ISBN 85-903798-2-5).

In practical terms, loadfile(...) returns a function (here called pagecals) that you need to run in order to make calcvals(...) accessible. Don't worry about this Lua way of doing things, just accept it for present purposes. Once we have run pagecals() we can then call calcvals({...}) with the input values (as a Lua table).

The next few lines of setpage.tex create another Lua table, pagevars, in which we store the values returned by calcvals({...}) and the values of the \setpage parameters #1...#8. Finally, it inputs margins.tex into our document. The reason for creating pagevars will be outlined in Section 7.2.

6.3 A more complete example

By way of a more complete demonstration, the following example shows \setpage being used to define a custom page of 234mm tall × 156mm wide being output on a PDF page size of 180mm wide × 260mm tall (just enough to contain the crop marks, see Section 7). This example uses a number of additional packages including atbegshi.sty and fancyhdr.sty. In particular, fancyhdr.sty has been used to output the headers and footers to show that \setpage cooperates with fancyhdr.sty. The package atbegshi.sty is used to output the crop marks on the pages. See Figure 5. There is also some homegrown code to setup the crop marks (lines 9 to 12 of example.tex)

example.tex

```
\documentclass[11pt,twoside]{article}
\input setpages
\setpage{180}{260}{156}{234}{15}{25}{10}{10}
\usepackage{fontspec}
\usepackage{atbegshi}
\usepackage{fancyhdr}
\begin{document}
% start of crop mark code
\input pix
\startpix
\imbox{c:/crops.pdf}{crop}
\endpix
% end of crop mark code
\setmainfont[Ligatures=TeX,Numbers=OldStyle]
  {Constantia}
\fontsize{10}{11}\selectfont
\pagestyle{fancy}
\fancyhead{} % clear all header fields
\fancyhead[RO,LE]
  {\bfseries The performance of new graduates}
\fancyfoot{} % clear all footer fields
\fancyfoot[LE,RO]{\thepage}
\fancyfoot[LO,CE]{From: K. Grant}
\fancyfoot[CO,RE]{To: Dean A. Smith}
\renewcommand{\headrulewidth}{0.4pt}
\renewcommand{\footrulewidth}{0.4pt}

\textsc{but i must explain} to you how all this
  mistaken...

\end{document}
```

7 Crop marks

Many of the figures in this paper contain a small graphic at the corners of the area defined by the document pages. They are called *crop marks*, also referred to as "printers marks", "cut marks" or "trim marks", and are used to indicate the physical size of the final printed document pages. They are used during commercial printing activities, such as page imposition, colour separation, folding

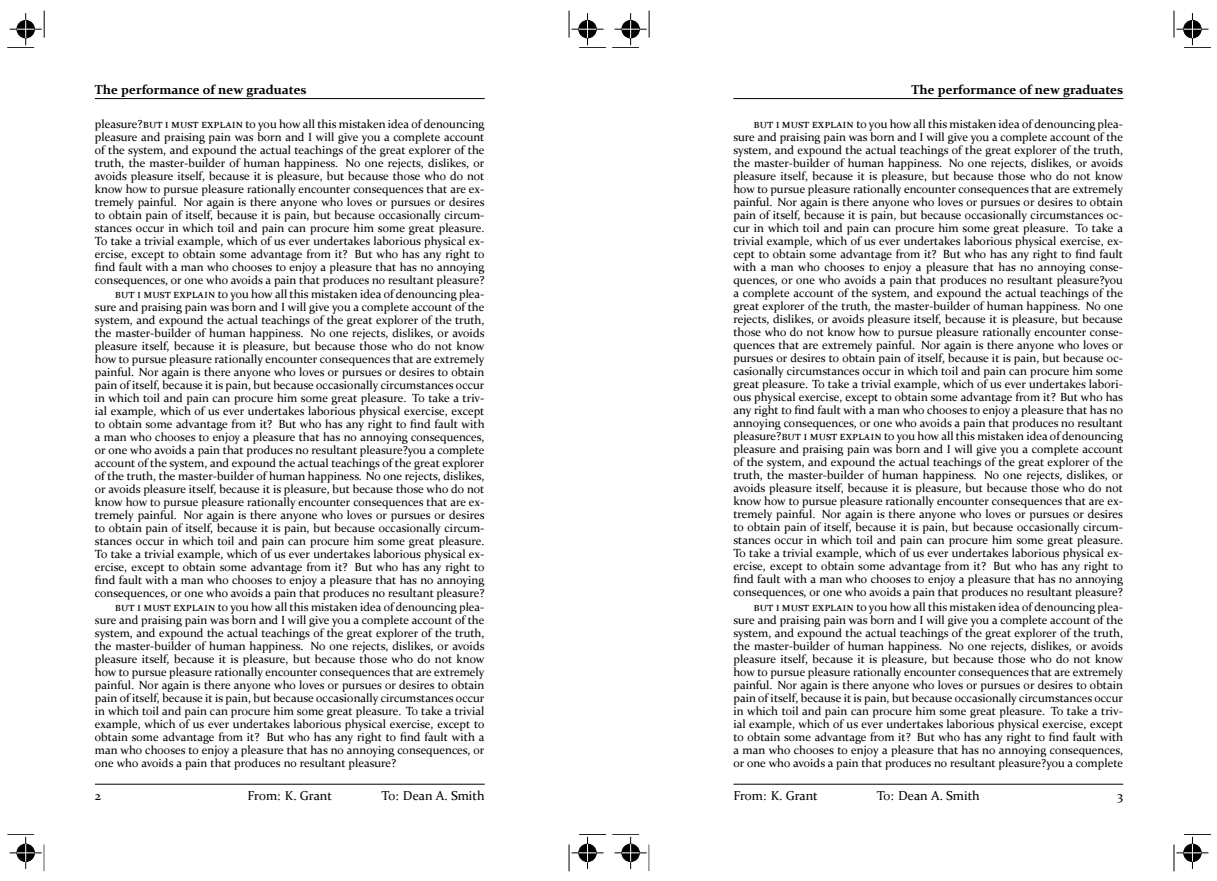

Figure 5. Using `setpage.tex` with `fancyhdr.sty` and `atbegshi.sty`.

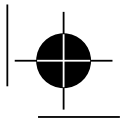

Figure 6. The crop mark produced by `crop.ps`.

and trimming. The physical appearance of crop marks will vary depending on the application used to generate the pages but, of course, with LuaTEX and other TEX engines you are free to create your own. With LuaTEX you can take advantage of the built-in METAPOST library, MPlib, to create crop marks with great precision. Figure 6 shows one design of crop mark, used by the author in various projects, which was created through some simple hand-rolled PostScript code (shown in listing `crops.ps` at the end of this article). To use this with LuaTEX or pdfTEX run the PostScript code through GhostScript or Adobe's Distiller to create a PDF file.

7.1 Crop marks and PDF XObjects

There are, of course, many possible techiques for getting crop marks placed on your pages, including PGF and TikZ, METAPOST, LaTEX packages and so forth. Here, I'll give an overview of the technique which has been used to generate crop marks on some of the figures in this paper. The crop mark shown in Figure 6 is stored in an external PDF file and embedded *just once* into the PDF generated by LuaTEX as a PDF object type called a *form XObject*, which is a "self-contained description of any sequence of graphics objects". The use of form XObjects helps to minimise the size of the PDF file: the data (PDF graphics operators) describing the XObject (our crop mark) are included into the PDF file just once. Instead of embedding the data multiple times, each page in your PDF file can "reference" the XObject by applying PDF operators to re-use it as part of the content of your pages. For example, by performing various coordinate transformations, such as translation or rotation, and then drawing the XObject into your transformed coordinate system.

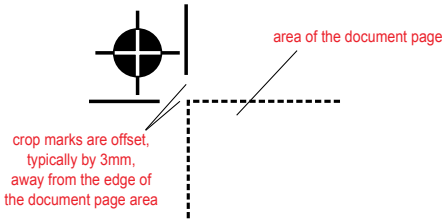

Figure 7. The location of crop marks relative to the document page.

7.2 Placing crop marks

Although crop marks are placed at the corners of the printed page, there are a couple of points to consider, especially when using the technique of an external graphic.

- crop marks are offset from the actual page area, typically by 3mm, so that they do not risk "contaminating" the actual printed area (see Figure 7);
- a more subtle point when placing crop marks contained in an external file is to take into account the actual width of the lines of the crop marks themselves and shift the positioning of the crop mark, relative to page corners, by half the width of the lines (see Figure 8);

Advanced readers will also observe that if you are involved in colour printing work, which requires colour separations, then of course, you may need to ensure that your crop marks appear on each colour separation/plate. However, this is beyond the scope of this paper and *definitely* something you should be discussing with the prepress department of your printing company. Additionally, the PDF specification makes specific provision for something called *Printer's mark annotations* which "...provide a mechanism for incorporating printer's marks into the PDF representation of a page, while keeping them separate from the actual page content." Again, this is beyond the scope of this paper and the interested reader is referred to the PDF specification (1.4 and later).

7.3 Getting crop marks onto the PDF page

Firstly, let me note that at the time of writing this article (April 2011), not only does LuaTeX support pdfTeX's facilities for embedding external PDF files but, in addition, it offers the built-in `epdf` library, thanks to the ongoing development work of Hartmut Henkel. The `epdf` library looks to provide a wonderful API for working with PDFs so by the time you read this article you'll likely have even more options at your disposal.

To place the crop marks you will, of course, need the (x, y) coordinates of the corners of your page, relative

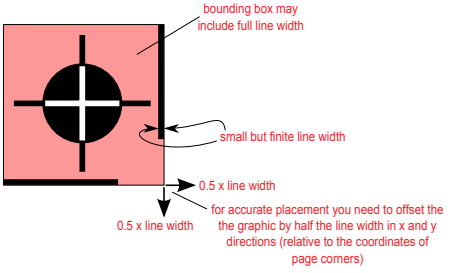

Figure 8. When placing an external graphic as a crop mark you may need to make micro-adjustments to account for the widths of lines.

to some origin which is *usually* the bottom left-hand corner of the PDF document area. Although this origin is common in PDF documents the reader should be aware that it is not *required* by the PDF standard. Refer to the PDF specification for details on "user space", coordinate systems and the CropBox for detailed guidance.

In outline you need to:

1. embed your crop mark graphic as a form XObject;
2. determine the (x, y) coordinates of the corners of your page;
3. for each corner, apply some PDF operators to place and rotate the form XObject;
4. get the crop marks shipped out on every page.

Embedding a form XObject. This is quite straightforward using pdfTeX's primitive `\pdfximage`.

Calculating page-corner coordinates. Section 6.2 discussed `setpage.tex` and the creation of a Lua table called `pagevars` and here is where we can make use of it. Looking at `setpage.tex` you will see that `pagevars` stores the following values (in mm):

```
pagevars["paperwidth"]=#1
pagevars["paperheight"]=#2
pagevars["bookpagewidth"]=#3
pagevars["bookpageheight"]=#4
pagevars["bookoutermargin"]=#5
pagevars["bookinnermargin"]=#6
pagevars["booktopmargin"]=#7
pagevars["bookbottommargin"]=#8
pagevars["deltay"]=deltay
pagevars["deltax"]=deltax
```

We can re-use the values stored in `pagevars`, via `\directlua{ }`, to calculate our page corners, relative to the lower-left corner of the main PDF document. Remembering that `pagevars` stores our values in mm, the following Lua fragment calculates the values we need using the default PDF user space units of 72 points = 1 inch.

```
\directlua{
   dx = 72.0*(pagevars["deltax"]/25.4)
```

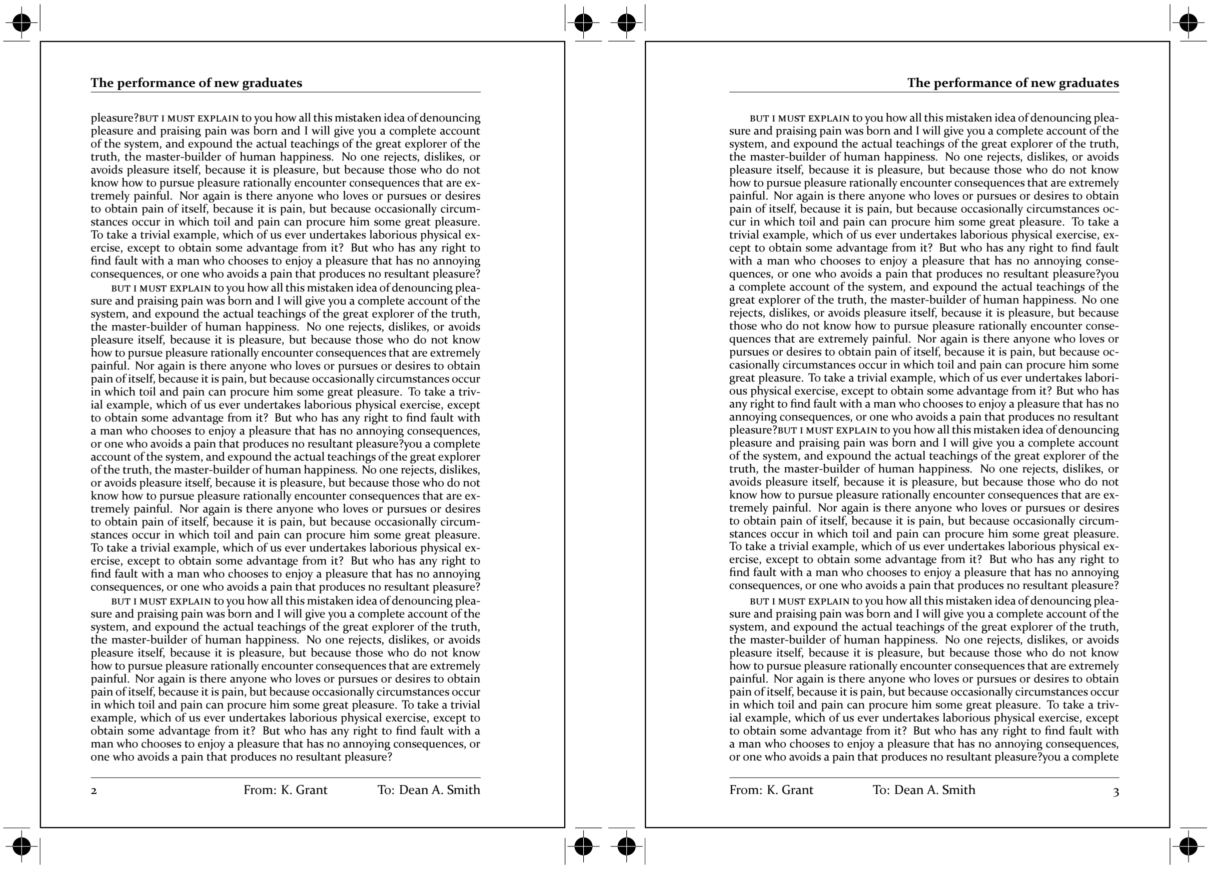

Figure 9. Using the `AtBeginShipout` command from `atbegshi` to draw a rectangle around the page area.

```
   dy = 72.0*(pagevars["deltay"]/25.4)
   bph=72.0*(pagevars["bookpageheight"]/25.4)
   bpw=72.0*(pagevars["bookpagewidth"]/25.4)
}
```

Working from Figures 2 and 3 and starting at the lower-left corner working clockwise, we can see that the page corner coordinates are $(\mathtt{dx}, \mathtt{dy})$, $(\mathtt{dx}, \mathtt{dy} + \mathtt{bph})$, $(\mathtt{dx} + \mathtt{bpw}, \mathtt{dy} + \mathtt{bph})$, $(\mathtt{dx} + \mathtt{bpw}, \mathtt{dy})$

Placing the form XObject. In addition to the commands inherited from pdfTEX, LuaTEX provides an extensive and rapidly developing set of APIs for working with PDF files, so you have a lot of choice in tackling this aspect of the problem. Here, we'll default to summarizing pdfTEX's `\pdfliteral page {<PDF code>}` which injects `PDF code` into the main PDF, establishing the origin as the lower-left corner of the PDF document page. This suits our needs because we already have our coordinates, from Section 7.3, relative to the lower-left corner of the main PDF document.

The graphic representing our crop mark, stored as a form XObject, clearly needs to be rotated as we move around the four corners of our page. Translation and rotation to establish a transformed coordinate system is a very standard operation within graphics and PDF work, typically achieved via matrix multiplications and won't be covered here. The PDF specification contains a very nice description of the relevant matrix maths and transformation of its coordinate systems and spaces.

PDF coordinate transformations. In PDF terms, for each page corner we need to:

1. modify the current transformation matrix using the PDF `cm` operator;
2. make a "call" to "paint" the embedded XObject using the PDF `Do` operator.

making sure, of course, that any modification to the current transformation matrix is local and enclosed within a PDF `q`...`Q` pair to save and restore the current PDF graphics state. To "execute" the `\pdfliteral ...` command which injects the PDF code to place and draw our crop mark, we use LuaTEX's `tex.print()` API call within a `\directlua { }` command. In the author's working code, this is achieved by calls such as this

```
tex.print("\\pdfliteral page{q "..s..
  " cm  /Im\csname pdf:crop\endcsname \space Do Q}")
```

By way of a small example, the following Lua fragment draws a rectangle around the area occupied by our pages within the actual PDF document area (see Figure 9 and compare to Figure 5).

```
% switch off current meaning of \\
\let\temp\\
\let\\\relax
\directlua{
  dx = 72.0*(pagevars["deltax"]/25.4)
  dy = 72.0*(pagevars["deltay"]/25.4)
  bph=72.0*(pagevars["bookpageheight"]/25.4)
  bpw=72.0*(pagevars["bookpagewidth"]/25.4)
  tex.print("\\pdfliteral page{q "..dx.."
    "..dy.." "..bpw.." "..bph.." re  S Q}")
}
```

7.4 Shipping crop marks on every page

And finally, you will need to ensure that your crop marks are shipped out on every page. One way to achieve this is to use the excellent `atbegshi` package by Heiko Oberdiek. In outline, you use `\AtBeginShipout{...}` with a `directlua{ }` command containing the Lua code required to place your crop marks. Here is a short fragment drawing a rectangle around our page. Note the `\let\temp\\` and `\let\\\relax` which temporarily disable LaTeX's definition of `\\` so that we can use this construct within `tex.print()`. See Figure 9.

```
\AtBeginShipout{
  \let\temp\\%
  \let\\\relax
  \directlua{
  dx = 72.0*(pagevars["deltax"]/25.4)
  dy = 72.0*(pagevars["deltay"]/25.4)
  bph=72.0*(pagevars["bookpageheight"]/25.4)
  bpw=72.0*(pagevars["bookpagewidth"]/25.4)
  tex.print("\\pdfliteral page{q "..dx.."
    "..dy.." "..bpw.." "..bph.." re S  Q}")
  }}
```

8 Conclusions and summary

The discussions and use of LuaTeX in this paper do not even begin to hint at the amazing versatility and potential of this incredible next-generation TeX engine. For that you should download and browse the latest LuaTeX Reference Manual from `luatex.org` and take a look at the available APIs. However, it is the author's hope that this paper has offered an interesting and practical starting point for anyone who wishes to explore LuaTeX.

8.1 Why I absolutely love LuaTeX

Here, I am grateful to the MAPS editors for allowing me a section of space for some personal reflection and indulgence. I first encountered LuaTeX when researching tools for typesetting my Arabic study notes and came across videos of conference presentations, by Hans Hagen and Idris Hamid, on River Valley's absolutely fabulous river-valley.tv website. I was transfixed and stunned by the truly beautiful and unbelievably sophisticated Arabic typography being presented by Hans and Idris. What was this amazing application? I recall being up to the early hours of the morning, on a work day..., Googling until I understood that the underlying TeX engine powering this incredible work was LuaTeX via, of course, the highly sophisticated ConTeXt package.

I had encountered the Lua language some years before and experimented with it for a number of projects, so I was aware of Lua's power, flexibility and ease of use as an embeddable scripting language with a wonderfully clean and straightforward C API, and certainly far easier to use than the APIs of more "heavyweight" scripting languages. Personally, I think the choice of Lua as the scripting language is perfect. For sure, we've all seen the mailing list or newsgroup arguments and threads spiralling into flame wars debating the merits of scripting language X or scripting language Y, with domain experts proposing complex arguments for and against a particular language. But LuaTeX is *here*, it is being *actively developed*, it *works* and it opens a truly *staggering* world of new solutions and opportunities to use the sophisticated algorithms of TeX exposed and accessible through a wonderfully simple but powerful scripting language.

The combination of a TeX-based typesetting engine coupled with OpenType fonts, UTF-8 input, exposure of TeX's internals, plug-ins through DLLs (on Windows) and all enabled through Lua as a scripting language was something I just had to explore. Well, nearly 18 months after my "discovery" of LuaTeX I have still not resumed my Arabic studies, continuing to follow the very latest updates of LuaTeX, compiling it from the latest source code as soon as any updates are available: sometimes daily. I am hooked, pure and simple! Be warned, LuaTeX is highly addictive, utterly absorbing and quite damaging to other hobbies you may enjoy. Happy TeXing.

8.2 Publication note

The material presented in this paper is based on independent work undertaken by the author in his spare time and does not necessarily reflect, represent or express any views or focus of interest or opinions of his employer the Institute of Physics or any of its group companies, including IOP Publishing Limited.

Graham Douglas, Senior Publisher
IOP Publishing, Dirac House, Temple Back,
Bristol BS1 6BE, UK
graham.douglas@readytext.co.uk

crops.ps

```
/ss   {setgray} def
/ssgs {/col exch def gsave col setgray} def
/gres {grestore} def
/MM {25.4 div 72 mul} def
%============================
% Define lengths
%============================
%offset from page
/TrimOffset 3 MM def
%length of trim mark
/TrimLength 8 MM def
/TotalLength TrimOffset TrimLength add def
%set the PostScript page size
<</PageSize
[TotalLength 0.25 add TotalLength 0.25 add]
>> setpagedevice
%Define Circle's parameters
%================================
%Centre of the circle
/CircleCentre TotalLength 2 div def
/Sqrt2 2 sqrt def
% Radius of circle
/CircleRadius  TotalLength Sqrt2 3 mul div def
% Define hair lengths
%============================================
/LengthOfBlackHair Sqrt2 1 sub Sqrt2 div neg
TotalLength mul TrimLength add 2 mul def
/LengthOfWhiteHair CircleRadius 2 mul def
%Positioning of black hairs
%============================================
/BlackHairGap TotalLength LengthOfBlackHair sub 2 div def
/WhiteHairGap TotalLength LengthOfWhiteHair sub 2 div def
%============================================
%Drawing procedures
%============================================
/DrawTrimLines        {
 0 ssgs
 0 TrimOffset moveto
 0 TrimLength rlineto
 TrimOffset neg 0 moveto
 TrimLength neg 0 rlineto
 stroke
 gres
} def

/DrawBlackHairs {
 0 ssgs
 CircleCentre neg BlackHairGap moveto
 0 LengthOfBlackHair rlineto stroke
 BlackHairGap neg CircleCentre moveto
 LengthOfBlackHair neg 0 rlineto stroke
 gres
} def

/DrawCircle {
 newpath
 0 ssgs
 CircleCentre neg CircleCentre CircleRadius
 0 360 arc
 gsave fill grestore stroke
 gres
} def

/DrawWhiteHairs {
 1 ssgs
 CircleCentre neg WhiteHairGap moveto
 0 LengthOfWhiteHair rlineto stroke
 WhiteHairGap neg CircleCentre moveto
 LengthOfWhiteHair neg 0 rlineto stroke
 gres
} def

/DrawAllMarks {
 0.5 setlinewidth
 DrawTrimLines
 DrawBlackHairs
 DrawCircle
 DrawWhiteHairs
} def

TotalLength 0.25 add 0 translate
-1 1 scale

TotalLength 0.25 translate
DrawAllMarks
showpage
```