

METAFONT/METAPOST and a complex Indic script: Malayalam

C. V. Radhakrishnan, K. V. Rajeesh,
K. H. Hussain

Abstract

Malayalam is an Indic script with numerous shape-shifting characters. We explore a reusable component-based design for Malayalam fonts, and develop them using METAFONT/METAPOST [6, 1] to assemble the characters. We discuss the paradigm shift from GUI design tools to ‘code-based’ design of shapes and glyphs, even by non-coders, and the advantages and challenges of using METAFONT/METAPOST to develop an OpenType font for a complex script. Finally, the progress made by our small team is shared.

1 Indic scripts and Malayalam

The Union of India has 23 languages [11], each one being the official language of one or more states. For convenience of governance, the Union was divided into states comprising areas with people speaking the same language. Thus, Kerala is the state of people speaking Malayalam, the adjacent state of Tamil Nadu is that of Tamil-speaking people, Karnataka of Kannada-speaking people, and so on. Any of these 23 languages can be used for official communication, including deliberations in the Indian Parliament. The Indian currency note bears the denomination in all languages (see Figure 1; fewer entries are due to the fact that some of the languages share the same script, apart from Hindi and English, which are already on the face of the note).

The Brahmic scripts [10], the family of languages to which Malayalam belongs, have a few common properties among most of the members. Each consonant has an inherent vowel, which is usually a short ‘a’, and other vowels are written by con-

Figure 1: The picture of the 200 rupees currency note of India; the denomination is printed in 15 scripts. (courtesy: Reserve Bank of India)



Figure 2: A few Malayalam characters to show the generally rounded shape.

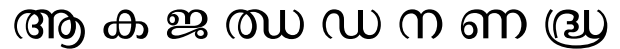
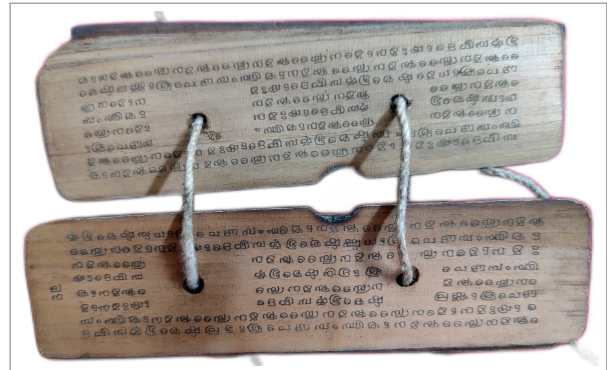


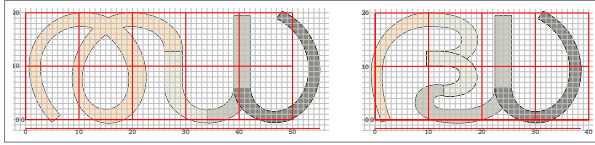
Figure 3: Manuscript leaves of Malayalam text. (courtesy: Wikipedia)



joining with the character. Each vowel has an independent form when not attached to a consonant, and a dependent form, attached to a consonant, at times to both on the left and right sides of the consonant. Up to four consonants can be combined in ligatures. Special marks are added to denote the combination of ‘r’ with another consonant. Nasalization and aspiration of a consonant’s dependent vowel are also denoted by separate signs. The alphabetical order is: vowels, velar consonants, palatal consonants, retroflex consonants, dental consonants, bilabial consonants, approximants, sibilants, and other consonants. Each consonant grouping has four stops (with all four possible values of voicing and aspiration, see Table 3) ending with a nasal consonant [10].

The above properties of the Brahmic family can be found in the Malayalam script. For instance, the first syllable in the word *poppler*, written in Malayalam, needs a vowel representation on both sides of the consonant *p*, which will look like ‘പോ’ where the middle character (പ) represents *p* and those on the sides represent the vowel *o*. Thus, a Malayalam font table can be enormous in size, with over 900 glyphs that constitute basic vowels and consonants for forming 57 characters, while the rest constitute the ligatures, vertical and horizontal conjuncts derived from the basic character set. *RIT Rachana* [3], a popular font in Malayalam, has over 920 glyphs derived from a base font table comprising 117 characters in the Unicode font table [9]. The script occupies the code points between 00D0 and 0D7F in the Unicode table.

Figure 4: Two glyphs showing shape components used with different colors.



2 Rationale

Upon closer examination of the characters in the Malayalam script (refer to Figure 2), one will notice that the majority of these characters are composed of arcs, semicircles, and circles of varying sizes. These elements harmoniously come together to form the distinctive shape of each character. The origins of the rounded and cursive design of these letters can often be attributed to the writing materials that were used in the past.

It is believed that the prevalence of round and cursive shapes in the letters of Indic languages can be traced back to the practicality of the writing instruments employed during their origin. The traditional writing instrument was a long, sharp metallic stylus used for inscribing text on dried and smoked palm leaves. Angular shapes would have been unsuitable as they could potentially tear the delicate leaves as they were being written upon. Hence, it seems plausible that this practical consideration influenced the widespread adoption of round and cursive letter forms (see Figure 3 showcasing a manuscript as an example).

The inherent rounded and cursive nature of the letters naturally led to the conclusion to create a set of predefined components with specific shapes. These components could then be reused effectively to construct complete characters. As one can easily deduce, the concept of reusability not only saves considerable time and effort but also ensures a consistent and uniform quality in terms of curves, cut angles, and similar attributes. Nonetheless, it is essential to acknowledge that this approach does have its limitations, some of which will be discussed in the following sections pertaining to reusable components and glyphs (Sections 3.3 and 3.4).

These limitations have significantly influenced design decisions, resulting in the development of methods to manipulate coordinates, adjust widths, and alter angles in the components as needed to fit the overall shape of the character of which they are part. In Figure 4, which showcases two glyphs, you can observe the utilization of shape components, distinguished by different colors to ease comprehension and analysis.

Figure 5: Four variants of a consonant character (pronounced *dja*) from the same METAFONT source code.



Another significant factor in our decision to utilize METAFONT/METAFONT was the potential for reusing the source code to generate variants of the font family. By employing separate configurations for each variant and requiring only minimal adjustments, we could easily create font variations of remarkable quality, thereby reducing both development time and effort. Four variants of a consonant character (pronounced *dja*), using the same METAFONT source code with different values for a few variables can be seen in Figure 5.

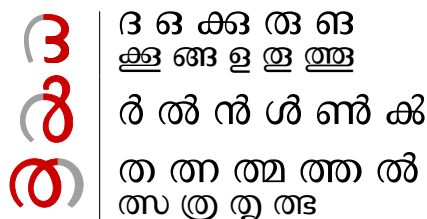
In the realm of character description languages, our choice of the code-based METAFONT/METAFONT can be attributed to our enduring association with the illustrious T_EX [5] and its companions. We derive immense pleasure from employing T_EX's sagacious and programmable markup language to fulfill our multifarious text processing requisites, eschewing the allure of graphical interface-driven applications. However, the merits of METAFONT/METAFONT extend far beyond mere preference.

The selection of METAFONT/METAFONT endows us with an array of supplementary advantages, including seamless cross-platform compatibility and the remarkable capability to produce vector outputs in the form of SVG and PostScript. Furthermore, the maintenance of our codebase becomes a simplified endeavor through the use of text-based source code, fostering clarity and facilitating future modifications. It is a confluence of these factors that harmoniously resonate with our intrinsic predilection for code-driven development, making the adoption of METAFONT/METAFONT an instinctive and judicious decision.

3 Research and design process

Once the design based on reusable components was finalized, the subsequent undertaking entailed identifying the most suitable curves and shapes to fulfill the objective. A comprehensive list of components was meticulously compiled, along with a corresponding catalog of potential characters and glyphs that could harness the potential of these components. To facilitate reader comprehension, a typeset list containing the components and the characters associated with them is provided in PDF format at

Figure 6: Reusable components and characters that can use them. The components are shown in red.



the link in [7]. In Figure 6, you will find three representative examples selected from this extensive list, providing a glimpse into the range of possibilities that await exploration.

Although the choice to delve into the realm of METAFONT/METAPOST came naturally to us, none of us possessed prior familiarity with the language or its intricacies. Consequently, our initial focus revolved around acquiring a firm grasp and cultivating a reasonable proficiency in the art of METAFONT/METAPOST. This endeavor demanded unwavering dedication and a significant investment of time, but it granted us the confidence necessary to embark on our creative journey.

As we delved deeper into the captivating world of METAFONT, we soon encountered a crucial turning point. After careful deliberation, we made the decision to transition to METAPOST. We recognized that METAPOST held a distinct advantage, enabling direct generation of vector outputs in the form of SVG and PostScript. These invaluable features seamlessly aligned with our ultimate objective of crafting fonts using software like FontForge or fontmake. Thus, our pursuit of perfection urged us to embrace the versatility and convenience offered by METAPOST, as it emerged as the perfect companion on our path towards mastering the art of font creation.

3.1 Initial attempts

Initially, our endeavors centered on creating the foundational characters in adherence to the Unicode table [9], employing fixed coordinates and distinct `penstroke` [6, p. 273] and `draw` [6, p. 271] functions for serif and sans-serif variants respectively. However, it wasn't long before we encountered the inherent limitations of this approach. Firstly, the fixed coordinate system presented a significant drawback, as it required users to manually input all the coordinates, contrasting with the more flexible algebraic expressions that would prompt METAFONT to calculate the coordinates through solving these expressions. The design of METAFONT itself encourages

users to adopt the latter method, as it allows for greater parametrization and versatility.

The sans-serif variants in SVG format, generated through the draw function with uniform line thickness, were initially considered viable candidates. However, they exhibited potential flaws when it came to removing overlaps during the font creation process, whether through FontForge or applications like Inkscape. It became apparent that utilizing stroke commands such as `penstroke`, which enables drawing an envelope of specified thickness and angles around the central line, as dictated by `penpos` [6, p. 273] commands for each coordinate, would ensure seamless overlap removal. This realization prompted us to abandon fixed coordinates in favor of a more suitable approach.

3.2 Parametrized approach

Various factors that impact the shape, angle, and thickness of strokes have been successfully parameterized. These encompass a range of essential parameters, as well as supplementary parameters that rely on the values of the foundational ones. A comprehensive listing of these parameters, including both the fundamental and dependent ones, can be found in Tables 1 and 2.

The parameters `t` and `u` play a crucial role in finely adjusting dimensions, although they differ in their impact. While both parameters contribute to this adjustment process, it's worth noting that `t` has the unique characteristic of having no effect on sans-serif variants. In other words, its influence becomes apparent only when serif variants come into play.

In addition to the aforementioned parametrization, we have also undertaken another set of parameter adjustments concerning the widths and angles of strokes, tailored to accommodate various variants. Initially, our focus encompassed four primary variants: serif, sans-serif, serif thin, and sans-serif thin. However, it is important to note that we have the flexibility to incorporate additional variants in the future, should the need arise.

To facilitate this parametrization process, we defined a numeric variable, `width_angle`, with val-

Table 1: The essential/foundational parameters used.

Parameter	Description	Default
<code>mag</code>	magnification	4
<code>thick</code>	width of thick line	17.2bp
<code>thin</code>	width of thin line	8.3bp
<code>t</code>	unit dimen for adjustments	5.5bp
<code>u</code>	unit width/height	5.5bp

Table 2: Supplementary parameters used.

Parameter	Description	Default
<code>o_cor</code>	overshoot correction	.5u
<code>lbearing</code>	left bearing	2u
<code>rbearing</code>	right bearing	1u
<code>ascent</code>	distance from baseline of character to top edge	10.4u
<code>dscent</code>	distance from baseline of character to bottom edge	0u

ues ranging from 1 to 4, corresponding to the aforementioned variants: serif, sans-serif, serif thin, and sans-serif thin, respectively. Leveraging the GNU tools within our workflow, passing the value of `width_angle` to the build process is an effortless task. This approach also has the advantage of easily observing the successive outputs of different variants as we construct characters.

We strongly believed that utilizing predefined variables for width and angles relating to different directions, based on the cardinal directions of north, north-east, east, south-east, south, south-west, west and north-west, would greatly enhance comprehensibility and ease of use. These variables would also have fixed values assigned to them. To ensure clarity and consistency, width variables will be prefixed with `w_`, followed by one or two characters indicating the respective direction. Similarly, angle variables will be prefixed with `a_`, followed by the same characters indicating the direction. It's worth noting that the east direction deviates slightly from the expected 'e' since using 'e' in the `penpos` command would result in an error. Therefore, the character sequence 'ea' has been employed in its place.

For a comprehensive list of all the width and angle variables, along with their suggested values, please refer to the following code listing. Please bear in mind that the usage of 'ea' instead of 'e' is a unique exception in the provided variables.

For detailed reference, the definitions and corresponding values of variables pertaining to the eight cardinal directions, for each value of the `width_angle` variable, are presented in the subsequent code listings (refer to Listings 1–4). This comprehensive resource offers a valuable point of reference for accessing precise information.

Listing 1: Width and angle variables for `width_angle` value 1 (serif normal).

```

1 thick:=17.2bp*mag; % width of thick line
2 thin:=8.3bp*mag; % width of thin line
3 w_cor:=(thick-thin);
4 w_w=thin; a_w=180; % west
5 w_nw=thin+.25w_cor; a_nw=135; % north-west

```

```

6 w_n=thin+.5w_cor; a_n=90; % north
7 w_ne=thick-.25w_cor; a_ne=45; % north-east
8 w_ea=thick; a_ea=0; % east
9 w_se=thick-.25w_cor; a_se=-45; % south-east
10 w_s=thick-.5w_cor; a_s=-90; % south
11 w_sw=thin+.25w_cor; a_sw=-135; % south-west

```

Listing 2: Width and angle variables for `width_angle` value 2 (sans-serif normal).

```

1 thick:= 1.5u;
2 thin := 1.5u;
3 w_cor:=(thick-thin);
4 t := 0.0u;
5 w_w := thin+.5t; a_w := 180;
6 w_ea := thick+.2t; a_ea := 0;
7 w_n := thin+.2t; a_n := 90;
8 w_sw := thick-.1t; a_sw := -140;
9 w_nw := w_sw; a_nw := 135;
10 w_ne := w_sw; a_ne := 45;
11 w_se := w_ea-1.2t; a_se := -50;
12 w_s := w_n; a_s := -90;

```

Listing 3: Width and angle variables for `width_angle` value 3 (serif thin).

```

1 thick:=8.6bp*mag;
2 thin:=4.3bp*mag;
3 w_cor:=(thick-thin);
4 w_ea:=thick+.2t; a_ea:=0;
5 w_w:=thin+.3t; a_w:=180;
6 w_n:=thin+.5t; a_n:=90;
7 w_sw:=thick-.1t; a_sw:=-140;
8 w_nw:=w_sw; a_nw:=135;
9 w_ne:=w_sw; a_ne:=45;
10 w_se:=w_ea-.6t; a_se:=-50;
11 w_s:=w_n; a_s:=-90;

```

Listing 4: Width and angle variables for `width_angle` value 4 (sans-serif thin).

```

1 thick:= .5u;
2 thin := .5u;
3 w_cor:=(thick-thin);
4 t := 0u;
5 w_w := thin+.5t; a_w := 180;
6 w_sw := thick-.1t; a_sw := -140;
7 w_ea := thick+.2t; a_ea := 0;
8 w_n := thin+.2t; a_n := 90;
9 w_nw := w_sw; a_nw := 135;
10 w_ne := w_sw; a_ne := 45;
11 w_se := w_ea-1.2t; a_se := -50;
12 w_s := w_n; a_s := -90;

```

3.3 Reusable components

Now let us look into a typical instance, the construction of the consonant character ɱ , which bears the phonetic resemblance to the initial syllable of the word 'November'. This character is ingeniously

brought to life through the deployment of two shape components. Keen observers will note the presence of two distinct lobes—an elegant left lobe and an equally poised right lobe. These lobes find their algebraic expressions in the form of two shapes, known as `c_llobe` and `c_rlobe`, respectively. The nomenclature itself displays an inherent clarity, with the prefix ‘c’ symbolizing the component.

The algebraic pursuit of shaping this character involves the harmonious interplay of two fundamental elements. Firstly, we encounter the precise coordinates of each pivotal point that contribute to the formation of the character’s curved contours. Secondly, the width/angle values assigned to each coordinate, accompanied by the stroke command that gracefully connects them, further embellish the visual tapestry. In order to gain a better understanding of this intricate process, the following code unveils the craftsmanship behind its creation.

Listing 5: Listing of the METAPOST source code in the character build file of `m`.

```

1 input mpost-defs; % MetaPost definitions
2 input ml-shape-lib;% lib. of shape comps.
3 input option; % proofing, width/angle opts
4 input out; % PDF/SVG output options
5
6 beginfig(34);
7  coor_c_llobe (1) (0,0);
8  pstroke_c_llobe (1);
9  coor_c_rlobe (n1.2) (x1f.r-.5wd2b,0);
10 pstroke_c_rlobe (2);
11 endfig;
12 end;
```

The files that are evident inputs within the build source, as enumerated in Listing 5, manifest a diverse array of categories. The first file, `mpost-defs.mp`, consists of a select assortment of definitions derived from `plain.mp`, albeit redefined or customized to align with our objectives. The second file, `ml-shape-lib.mp`, is an extensive compendium of shape components with their affiliated macros. Notably, this file internally invokes `ml-glyphs-lib.mp`, which in turn houses the essential definitions of glyphs. The `option.mp` and `out.mp` files help in the build process.

Now, let us examine the macro `coor_c_llobe`, which encompasses the x and y values representing the coordinates of the left lobe. This macro accepts one suffix argument and two expression arguments, namely, `xsh` and `ysh`. These expression arguments serve as containers for the dimensions that dictate the horizontal and vertical shifts of the component when it is positioned within the character construction process.

Listing 6: Listing of the definition of `c_llobe`.

```

1 def coor_c_llobe (suffix $)(expr xsh,ysh) =
2   z$a=(xsh+.55b, ysh+0h);
3   z$b=(xsh+.05b, ysh+.5h);
4   z$c=(xsh+.8b, ysh+1h); oc$c(-.5);
5   z$d=(xsh+1.5b, ysh+.63h);
6   z$f=(x$d, ysh+0h);
7 enddef;
8
9 def pstroke_c_llobe (suffix $) =
10  penpos$a(w_w-.5t,a_sw);
11  penpos$b(w_w+.2t, a_w);
12  penpos$c(w_n, a_n);
13  penpos$d(w_ea-.1t,a_ea);
14  penpos$f(w_ea-.1t,a_ea);
15  penstroke subpath (start,stop) of
16    (z$a.e .. z$b.e{up} .. {right}z$c.e
17     .. z$d.e{down} .. z$f.e);
18  penlabels($a,$b,$c,$d,$f);
19 enddef;
```

In terms of providing the x -coordinate values, we employ the symbol ‘b’ to represent breadth, while the y -values are denoted by ‘h’, which stands for height. By default, the default values assigned to `b` and `h` are 10u and 20u respectively. This flexibility empowers us to modify the values of `b` and `h`, thereby generating condensed or extended variants with ease.

The macro responsible for stroking the paths is named `pstroke_c_llobe`. It encompasses all the `penpos` commands, specifying the width and angle of each coordinate as defined within the corresponding `coor_⟨component⟩` macro. It is worth noting that the width and angle values are expressed in terms of the width/angle variables, as outlined in Listings 1–4, which correspond to the specific font variant being built.

In addition to the `penpos` commands, the path stroking macro also includes one or more (in other cases) `penstroke` commands. These commands connect each coordinate in a sequential manner, employing curves or straight lines as required by the character design. Further, the macro incorporates one or more `penlabels` commands, which facilitate the printing of labels and the left/right edges of the coordinates when generating proofs. It is important to mention that if the value of the `proofing` variable exceeds 2, the `penlabels` command will also display the angle and width values of each coordinate, providing valuable insights during the debugging phase.

The coordinate and path stroking macros for `c_rlobe` are provided in Listing 7, to allow the reader to examine the implementation.

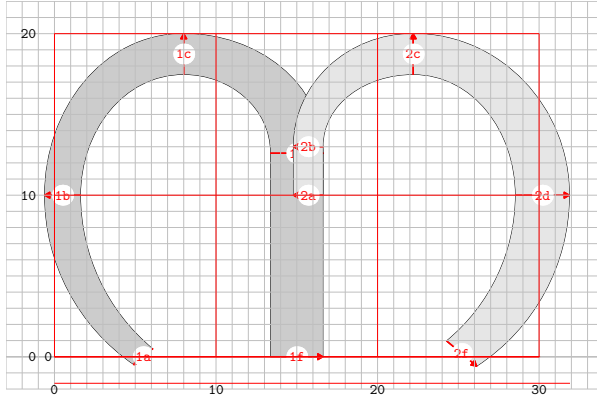


Figure 7: The proof image of the character *m*.

Listing 7: Listing of the definition of `c_riobe`.

```

1 def coor_c_riobe (suffix $)(expr xsh,ysh) =
2   z$a=(xsh+0b, ysh+.5h);
3   z$b=(x$a, .65h);
4   z$c=(xsh+.65b, ysh+1h);
5   z$d=(xsh+1.45b, ysh+.5h);
6   z$f=(xsh+.95b, ysh+0h+.2t);
7   y$c:=y$c-.5wd$c;
8 enddef;
9
10 def pstroke_c_riobe (suffix $) =
11   penpos$a(w_sw-.5w_cor,a_w);
12   penpos$b(wd$a,a_w);
13   penpos$c(w_n,a_n);
14   penpos$d(w_ea,a_ea);
15   penpos$f(w_s-.1t,a_se+10);
16   penstroke subpath (start,stop) of
17     (z$a.e{up}
18     .. z$b.e{up} .. z$c.e{right}
19     .. z$d.e{down} .. z$f.e);
20   penlabels($a,$b,$c,$d,$f);
21 enddef;

```

Figure 7 illustrates the character constructed for the serif normal variant of the font (`wa=1`), utilizing the code presented in Listings 5–7. The components have been visually distinguished using varying shades of gray to aid comprehension.

In a prior section (Section 2, Rationale), we highlighted the fact that the predefined component approach is not exempt from limitations. It is worth noting that there were instances where we encountered the need to adjust the position of coordinates to align with a particular character shape or design. This task proved to be quite challenging, given that the x and y values of the coordinates had been predetermined. Consequently, in order to overcome this obstacle, we undertook the task of redefining the `z` macro [6, p. 277] that assigns values to x and y coordinates. This revised version of the macro now

includes (refer to Listing 8) a check for any delta values associated with x or y , to add prior to assigning the respective original values.

Listing 8: The modified definition of `z` macro.

```

1 vardef z@#=(x@# - if known dx@#: dx@#
2   else:0 fi,
3   y@# - if known dy@#: dy@# else:0 fi)
4 enddef;

```

Thus, if there exists a definition for `dx` or `dy` associated with a coordinate `z`, signifying the intended horizontal and vertical shifts respectively, then these shifts will duly be applied to their respective coordinate values prior to final assignment within the pair definition. To illustrate this concept, let us consider the example `dx1b = -2u` (provided in code Listing 9). It is worth noting that the delta values need to be provided just before the occurrence of `coor_c_riobe`.

Listing 9: Example to show the shifting of coordinates.

```

1 beginfig(32);
2   % dx1b=-2u;
3   coor_g_da (1) (0,0);
4   pstroke_g_da (1);
5 endfig;

```

This code generates the consonant character `β` (using different subroutines than the previous examples). Should we desire a slightly more rounded contour for the left curve, it becomes necessary to adjust the position of the coordinate labelled `1b` towards the left, aligning it with the desired dimensions. This adjustment can be accomplished by employing the delta variable, written as `dx1b = -2u`. It specifies the intended shift of `2u` to the left. It's commented out in line 2 of Listing 9. To gain a visual understanding of the original design alongside the modified version, kindly refer to the images presented in Figure 8.

Similarly, we can utilize the `dy` approach to shift the vertical position. Nevertheless, the need for precise adjustments extends beyond mere shifts in the x and y directions. At times, it becomes necessary to modify the predetermined angles and widths using the `penpos` commands for individual coordinates. To enable this functionality, we had to redefine the original `penpos` command [6, p. 273], as shown in Listing 10.

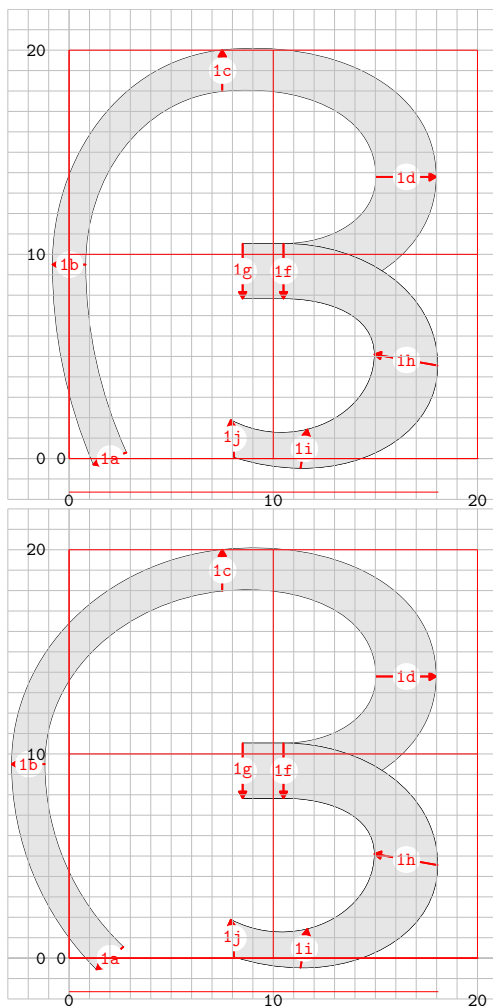
Listing 10: Redefined `penpos`.

```

1 vardef xangle@#(expr xd) =
2   (ang@#)=(xd); enddef;
3 vardef xwidth@#(expr xb) =
4   (wd@#)=(xb); enddef;
5 vardef penpos@#(expr b,d) =

```

Figure 8: Example showing horizontal shift of a coordinate. The figure at the top is the original character, while the one below shows the midpoint of the left curve shifted by $2u$ towards the left.



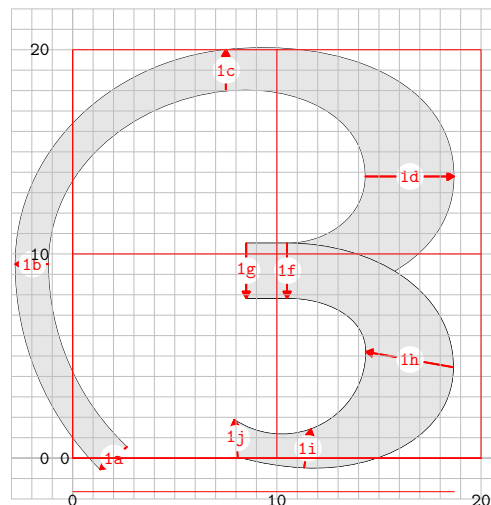
```

6 if unknown ang@#: xangle@#(d); fi
7 if unknown wd@#: xwidth@#(b); fi
8 (x@#r-x@#l,y@#r-y@#l) =
9   (wd@#,0) rotated ang@#;
10 x@#=.5(x@#l+x@#r);
11 y@#=.5(y@#l+y@#r);
12 enddef;

```

Let's examine the impact of utilizing these features through a practical example. In the image on the right side of Figure 8, it appears that the angle of coordinate 1a (the bottom ending of the left-hand stroke) isn't correct as initially set by the command `penpos$a(w_w+.2t, a_sw-20)`. However, it was deemed suitable for the image on the left side of the same figure. To address this, we can make adjustments by inserting the code `ang1a = a_sw`; in

Figure 9: Revised image of the character after changing the widths and angles.



Listing 11: New source with changed angle and widths.

```

1 beginfig(32);
2 dx1b=-2u; % change x-pos
3 ang1a=a_sw; % change angle
4 coor_g_da (1) (0,0);
5 wd1d=wd1h=w_ea+1t; % change width
6 pstroke_g_da (1);
7 endfig;

```

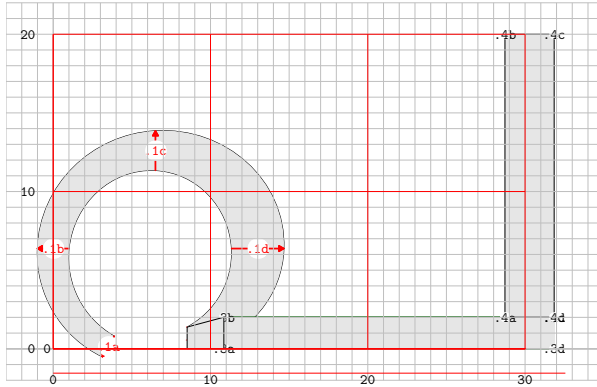
the character code; `ang⟨coordinate⟩=` followed by the value is the syntax of the command.

In a similar fashion, we can also modify the width of any coordinate by using the command `wd⟨coordinate⟩ = ⟨value⟩`. Suppose we wish to alter points 1d and 1h. This can be achieved by incorporating the code `wd1d = wd1h = w_ea+1t`. Essentially, this means that one unit of width will be added to the current width (which was initially defined as `w_ea`).

Please refer to the updated image of the character in Figure 9 and take a moment to compare it with the right image in Figure 8.

There remain a few additional features worthy of explanation, which we shall defer to a subsequent section (refer to Section 3.5). These features give the user additional tools to undertake the task with utmost ease. Among these capabilities are the ability to incise a path at any given point, ascertain the coordinates, angle, and width of said incision point, the redefined `penlabels` command, as well as the utilization of the `find_outline`, `pstroke_stem` and `overshoot` correction commands.

Figure 10: The consonant character, pa .



3.4 Reusable glyphs

The necessity for reusable definitions of glyphs, much like that of reusable components, arose when encountering certain horizontal conjuncts such as pa , pa , pa , and so forth, where characters are repeated horizontally. Similarly, the need for reusable definitions arose in the case of vertical conjuncts like pa , pa , etc., that involve the repetition of the same characters vertically. Such requirements also emerged in cases such as pa , pa , pa , and others, where different characters are stacked vertically.

In such circumstances, it is only natural to harness the programmability of METAPOST as a logical progression. This enables the definition of all the glyphs that might undergo repetition, whether in the formation of horizontal or vertical conjuncts, or when glyphs are combined with vowel signs to form ligatures, such as pa , pa , pa (phonetically equivalent to pra , pru , pru), derived from the consonant ‘ pa ’ (pa). Needless to say, the utilization of these reusable definitions greatly expedites the creation of conjunct build files.

Let us now see the typical composition of a glyph definition by carefully examining the source code for the consonant character pa (refer to Figure 10), given in Listing 12. This exploration will shed light on the intricate details that contribute to the formation of this particular character.

Listing 12: The glyph definition of pa from ml-glyph-lib.mp .

```

1 def gl_pa (suffix prx) =
2   coor_c_ra_sm (prx.1)(0,0);
3   % Lift up end point of ra_sm (1f)
4   % and set width relative to its
5   % start point (1a)
6   y.prx.1f := y.prx.1a.1;
7   wd.prx.1f := wd.prx.1a;
8   pstroke_c_ra_sm (prx.1);

```

```

9   stroke_stem (prx.3)
10    (x.prx.1f.r+30,0,2.1b,w_w);
11  pstroke_stem (prx.4)
12    (x.prx.3d-thick,y.prx.3c,thick,1h-w_w);
13  pstroke_edge ((x.prx.1f.l,y.prx.3a),
14    z.prx.1f.l,z.prx.3b,z.prx.3a);
15  reset_xst;
16 enddef;

```

The definitions of the routines coor_c_ra_sm and pstroke_c_ra_sm called by gl_pa are part of ml-shape-lib.mp and are given in Listing 13.

Listing 13: The definition of coordinates, width, angles and path stroke of pa from ml-shape-lib.mp .

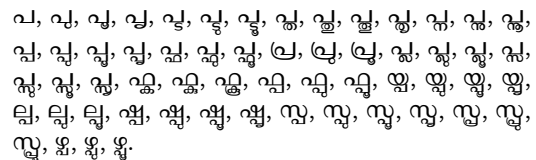
```

1 def coor_c_ra_sm (suffix $(expr xsh,ysh) =
2   z$a=(xsh+.35b, ysh+0); oc$a(.1);
3   z$b=(xsh+0b, .5[y$a,y$c]);
4   z$c=(.5[x$b,x$d], ysh+.63h);
5   z$d=(xsh+1.3b, .5[y$a,y$c]);
6   z$f=(xsh+.9b, y$a+.0h);
7 enddef;
8
9 def pstroke_c_ra_sm (suffix $) =
10  penpos$a(w_w-.5t, a_sw+20);
11  penpos$b(w_w, a_w);
12  penpos$c(w_n, a_n);
13  penpos$d(w_ea, a_ea);
14  penpos$f(w_s-.5t, a_se);
15  penstroke subpath (start,stop) of
16    (z$a.e .. z$b.e .. z$c.e ..
17    z$d.e .. z$f.e);
18  penlabels($a,$b,$c,$d,$f);
19 enddef;

```

The commands pstroke_stem , stroke_stem and pstroke_edge and their usage are described in detail in Section 3.5.

The above definitions allow building, fairly easily, a plethora of glyphs (58 in number) listed below, those where pa is an integral part:



The suffix argument, denoted as prx , within the glyph definition of gl_pa may pose a puzzling query in the minds of readers, warranting a thorough explanation. It is crucial to comprehend that the suffixes of macros must possess unique identities within a $\text{beginfig} \dots \text{endfig}$ environment. Failure to adhere to this requirement will result in an error, halting the processing by METAPOST.

In instances where we need to invoke the same shape functions multiple times, as exemplified in the source code in Listing 14 (its output can be seen

in Figure 11), it becomes imperative to ensure the uniqueness of suffixes. To achieve this, we resort to the practice of prefixing the suffixes with additional characters while calling the glyph definitions. (The usage of `vconj` shall be explained in Section 3.5.7 on vertical conjuncts.)

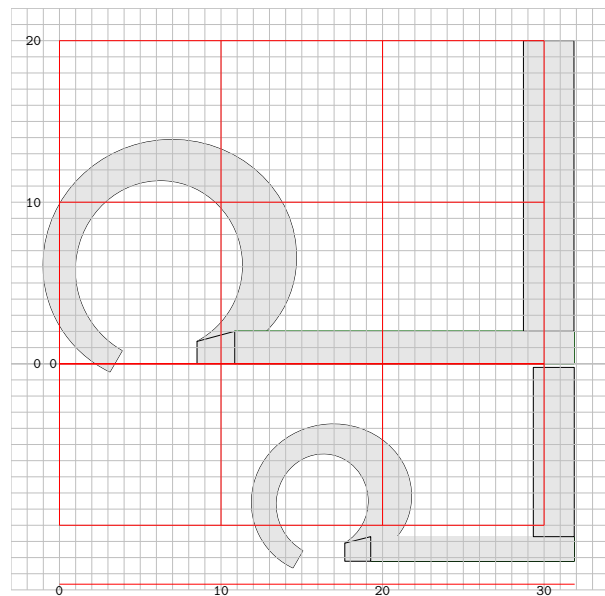
Listing 14: The source listing of `p1p1`, a vertical conjunct ഘ

```

1 beginfig(00);
2 g1:=image(gl_pa(p)); % first prefix
3 vconj:=true; width_angle(wa_n);
4 g2:=image(gl_pa(pp)); % different prefix
5 g3:=g2 xscaled .6 yscaled .6;
6 currentpicture:=g1;
7 addto currentpicture also g3
8   shifted (xpart (lrcorner g1)
9     -xpart(urcorner g3),-(12u+5));
10 endfig;

```

Figure 11: The vertical conjunct, ഘ , created by the code in Listing 14.



3.4.1 A word about glyph naming

It’s time to provide a brief explanation of the glyph naming convention utilized in our fonts. Instead of using Unicode code points as identifiers for characters, or Adobe glyph list conventions, we have chosen a specialized abbreviated form for both vowels and consonants. This methodology was developed by one of the authors, Hussain, over two decades ago for simplified glyph naming in the fonts he had created, including the most popular, “Rachana”.

Vowels are indicated by the prefix `m1_` followed by the corresponding vowel sound in the Latin

script. For example, `m1_a` represents the vowel, അ (OD05).

Each of the consonant groups, such as velar, palatal, retroflex, dental, bilabial, approximants, sibilants, and others, consist of four stops encompassing all possible values of voicing and aspiration. They are named using one or two representative characters in Latin script (`k` for velar, `ch` for palatal, `t` for retroflex, `th` for dental, `p` biblabial; but each member of the approximants, sibilants and others has been assigned a unique character depending on the sound), followed by a numerical index ranging from 1 to 4. For instance, the first velar consonant ക (OD15) is designated as `k1`. As you can infer, the other three (ഖ , ഗ , ഘ) are named `k2`, `k3`, and `k4` respectively. Certain vowel signs and consonants conjoin with many base characters; these are also named appropriately. Table 3 provides a detailed picture of the naming of consonants. It is interesting to note that this naming convention suits most Indic scripts.

Table 3: Table showing all the consonants, vowel signs and their glyph names.

	Voiceless		Voiced		
	Unasp.	Asp.	Unasp.	Asp.	Nasal
velar	k1 ക	k2 ഖ	k3 ഗ	k4 ഘ	ng ങ
palatal	ch1 ച	ch2 ഛ	ch3 ജ	ch4 ഝ	nj ഞ
retroflex	t1 ട	t2 ഢ	t3 ഡ	t4 ഢ്യ	nh ണ
dental	th1 ത	th2 ഥ	th3 ദ	th4 ധ	n1 ന
bilabial	p1 പ	p2 ഫ	p3 ബ	p4 ഃ	m1 മ
approximants	y1 യ	r3 ര	l3 ല	v1 വ	
sibilants	z1 ശ	sh ഷ	s1 സ	h1 ഹ	o ഔ
others	lh ള	zh ഴ	rh റ		
vowel/ consonant signs	u1 ു	u2 ൂ	y2 ്യ	r4 ൃ	l3 ല്
	v2 ്				

Unasp. = unaspirated; Asp. = aspirated


This convention allows for convenient usage of these intuitive consonant names when constructing conjuncts and ligatures, eliminating the need for lengthy and less user-friendly combinations of

Listing 15: Source listing of the redefined `penlabels`.

```

1 vardef penlabels@#(text t) =
2   if proofing > 1:
3     forsuffices $$=l,r: forsuffices $=t:
4       if known z$:
5         interim linecap:=rounded;
6         interim ahlength:=8bp;
7         interim ahangle:=60;
8         drawarrow z$.l -- z$.r
9           withcolor red;
10        drawdot (x$,y$) withpen pencircle
11          scaled 5mm withcolor white;
12        defaultscale:=.75;
13        s_len:=length(str$); st_idx:=s_len-3;
14        makelabel@#(substring(st_idx,s_len)of
15          (str$),(x$,y$)) withcolor red;
16        if proofing > 2:
17          label(decimal(wd$),z$-(0,1u))
18            withcolor .5white;
19          label(decimal(ang$),z$-(0,1.5u))
20            withcolor .5white;
21        fi
22      fi
23    endfor
24  endfor
25 fi
26 enddef;

```

code points. Undoubtedly, the glyph name of `k1th1r3` for  (pronounced like *kthra*) is much easier to remember than the cryptic hex sequence `OD05 OD4D OD24 OD4D OD31`.

3.5 Additional features

The supplementary features elucidated in this section are not absolutely imperative for carrying out the font creation process. Nonetheless, they enhance the workflow by equipping users with additional tools that contribute to making their lives a tad more convenient.

3.5.1 Redefined `penlabels` command

The command `penlabels` [6, pp. 36, 274] has been redefined (see Listing 15):

1. to display no labels if `proofing < 2`.
2. if `proofing = 2`, to display the labels in a white circle (since the path is filled with gray color in proof mode) and display a red arrow, the head of which points to the right edge which provides an indication of the angle visually. This is in lieu of the default method of displaying `l` and `r` labels in black.

3. to display the widths and angle of the coordinates if `proofing > 2`, which is handy in certain debugging situations.

3.5.2 Slicing the path — subpath

The `subpath` command [6, p. 133] has been extensively utilized in the `penstroke` macros, as it provides a convenient means to slice the path at arbitrary locations without affecting the path's flow or curvature, even when the cut point happens to intersect a curve. Most `penstroke` commands, if required, include the `subpath` command with (`start`, `stop`) variables as its arguments, defaulting to 0 and `infinity` respectively. This allows users to modify these variables in the build file as per the dictates of the shape of the glyph. The code, as shown in Listing 16, illustrates how the start and stop points of two consonants, `t1` (`s`) and `th3` (`ʒ`), have been applied to the cut just after the start of `t1` and towards the end of `th3` to create derivative glyphs of `t1r1` (`ʂ`) and `th3r1` (`ʒ̣`) respectively.

Listing 16: Application of `subpath`.

```

1 %%% glyph def of t1 %%%
2 def gl-Ta =
3   coor_g_ta (t1.1) (0,0);
4   reset_cut; start:=xstt; % set start of subpath
5   pstroke_g_ta (t1.1); % strokes from 'start'
6   reset_xst;
7 enddef;
8 %%% glyph def of t1r1 %%%
9 def gl-TR =
10  xstt:=3; gl-Ta; % draw curve from 4th point
11  ang_cor:=-5;
12  coor_c_prkar (t1.5) (x.t1.1f,3u);
13  start:=0;
14  wd.t1.5a=wd.t1.1f;
15  x.t1.5f=x.t1.1g;
16  x.t1.5a=x.t1.1d;
17  y.t1.5a=y.t1.1d;
18  wd.t1.5f=wd.t1.1g;
19  pstroke_c_prkar (t1.5);
20  reset_cut;
21 enddef;
22 %%% glyph def of th3 %%%
23 def gl-da =
24  coor_g_da (th3.1) (0,0);
25  stop:=xstp; % stop last part of curve at 'xstp'
26  pstroke_g_da (th3.1);
27  reset_xst;
28 enddef;
29 %%% glyph def of th3r1 %%%
30 def gl-dR =
31  dy.th3.1f=-2u;
32  coor_g_da (th3.1) (0,0);
33  stop:=1; % stop curve at 2nd point
34  % of last part

```

Figure 12: Illustration of subpath operation. The image at left is the consonant **s**, the right-hand image is after slicing at fourth coordinate is applied, and the one below is after appending the vowel sign to the sliced character.

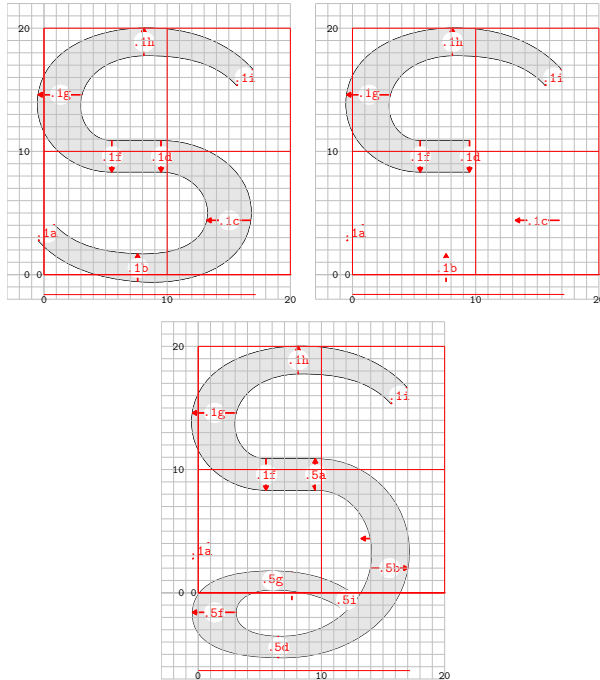
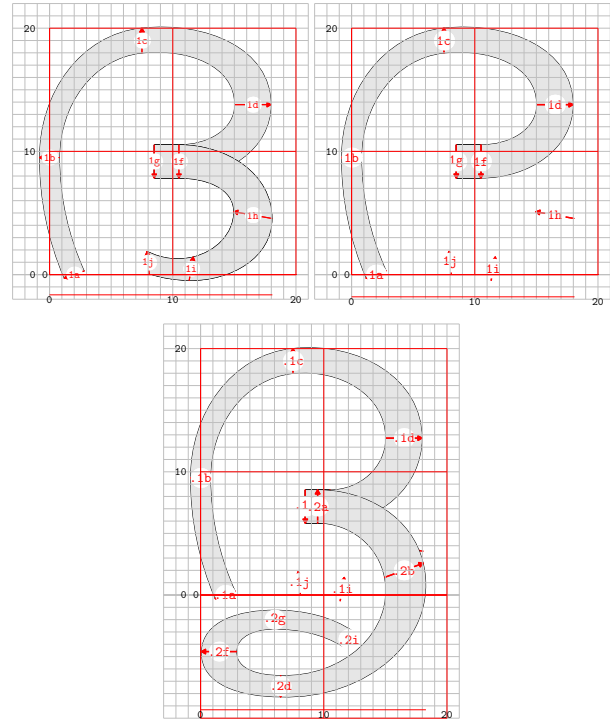


Figure 13: Illustration of subpath operation (continued). The image at the left is the consonant **ʌ**, the middle image is sliced at the beginning of last lobe and the one below is after appending the vowel sign to the sliced character.



```

35 pstroke_g_da (th3.1);
36 coor_c_krkar (th3.2)
37   (x.th3.1f-1u,0);
38 y.th3.2a:=y.th3.1f;
39 wd.th3.2a:=wd.th3.1g;
40 stop:=infinity;
41 pstroke_c_krkar (th3.2);
42 enddef;

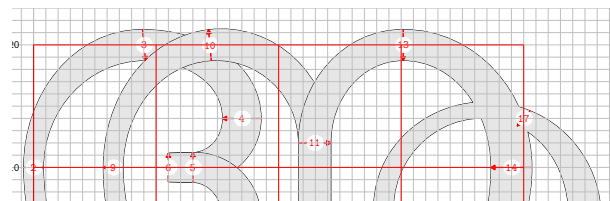
```

The three images in Figure 12 illustrate the subpath operation with respect to the consonant **s** and the second set in Figure 13 is that of **ʌ**.

3.5.3 Overshoot correction

The overshoot correction was previously done using a dimension variable `oc` which has the default value of `0.5u`. Since in serif versions, the characters have different stroke widths at top or bottom owing to the different angles of curves, and the `oc` variable is insufficient to manage overshoot corrections in this situation. However, it was felt that addition of a half stroke width at the bottom and subtraction of a half stroke width at the top will be the ideal solution for it. The images in Figures 14 and 15 illustrate the state of overshoot in pre- and post-application scenario respectively.

Figure 14: Overshoot curves at the top of serif version of the vowel **ᱠᱟ**.



The syntax of the command is:

```
oc⟨coordinate suffix⟩ (⟨corr value⟩)
```

A few usage examples are provided below, for situations of both directly coded coordinates like `z1`, `z2`, `z3`, ... and `$`-suffixed situations like `z$a`, `z$b`, `z$c`, ... The correction line is provided just after the coordinate definition of the point needing correction.

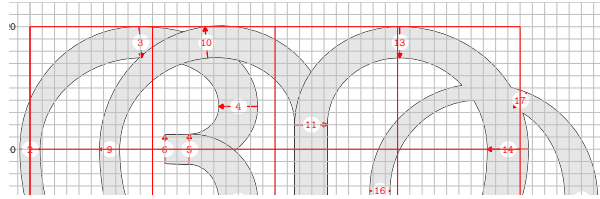
```

1 oc1 (-0.5);
2 oc5 (0.5);
3 oc$a(-0.5);
4 oc$d(0.5);

```

Any arbitrary value can be given as the argument.

Figure 15: Corrected overshoot curves at the top of serif version of the vowel æ .



Listing 17: Definition and usage of `pstroke_stem`.

```

1 %% definition:
2 def pstroke_stem (suffix $)
3   (expr xsh,ysh,width,height) =
4   x$a=x$b=xsh+0b; x$c=x$d=xsh+width;
5   y$a=y$d=ysh+0h; y$b=y$c=ysh+height;
6   filldraw z$a -- z$b --
7     z$c -- z$d --cycle withcolor gcolor;
8   if proofing>0:
9     draw z$a -- z$b -- z$c --
10      z$d --cycle withcolor black; fi
11   labels($a,$b,$c,$d);
12 enddef;
13 %% usage:
14 % pstroke_stem (<suffix>)
15 %   (<h-shift>, <v-shift>,
16 %   <width>, <height>);

```

3.5.4 The `pstroke_stem` macro

The horizontal and vertical stems that form part of some characters (e.g., œ , ø , æ , æ , æ , æ , ...) are drawn using the function `pstroke_stem`; its usage is shown in Listing 17.

After one argument for the suffix, the function requires four expression arguments: horizontal and vertical shifts, width, and height of the stem. If the height is greater than the width, the stroke becomes a vertical stem.

Readers are encouraged to examine lines 9–12 of Listing 12, where the usage of `pstroke_stem` is evident. Instead of `pstroke_stem`, you can see `stroke_stem` in line 10. Both have the same functionality, except that the latter invokes `outline` mode which is explained later (Section 3.5.10).

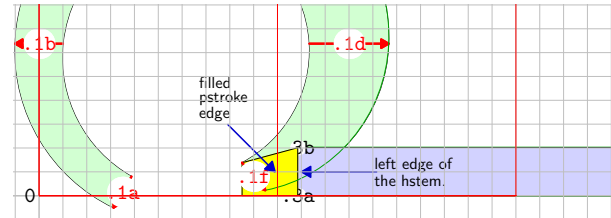
The image in Figure 10 demonstrates the effect of the aforementioned code.

3.5.5 The `pstroke_edge` macro

There exists a typographical nuance of reducing the height of an edge of a horizontal stem when it joins with a curved stroke as seen in Figure 16.

The `pstroke_edge` command draws a cyclic path connecting the left end coordinates of the hor-

Figure 16: Example of `pstroke_edge` when the left edge of a horizontal stem joins with the rounded part of the character. The path in light green color is the character; the hstem is colored light blue while the segment filled with yellow color is the `pstroke_edge`.



izontal stem (points 3a and 3b in the figure) with the left and right edges of end point of the curve (1f) and fills it. The macro definition and usage are provided in Listing 18.

Listing 18: Definition and usage of `pstroke_edge`.

```

1 %% definition:
2 def pstroke_edge (expr ll,ul,ur,lr) =
3   filldraw ll -- ul -- ur -- lr --
4     cycle withcolor gcolor;
5   if proofing>0:
6     draw ll -- ul -- ur -- lr --
7     cycle withcolor black; fi
8 enddef;
9 %% usage example:
10 %% lines 13, 14 of Listing 12
11 pstroke_edge (
12   (x.prx.1f.1, y.prx.3a),
13   z.prx.1f.1,
14   z.prx.3b,
15   z.prx.3a
16 );

```

The four coordinates needed for `pstroke_edge` can be provided starting from any point, as long as the coordinates are sequentially in cyclic order, no matter clockwise or anticlockwise.

3.5.6 DocGrid and PrintGrid

The `DocGrid` macro overlays a grid on top of the glyph image in `proofmode` (`proofing > 0`). It is an extended form of Knuth’s `makegrid` macro, explained in [6, p. 275]. Listing 19 provides the source code of the macro.

Listing 19: The source code of `DocGrid`.

```

1 def DocGrid (expr w,h) =
2   if proofing > 0:
3     begingroup
4       defaultscale := 1.1;
5       pickup pencircle scaled minor_rulewidth;
6       rulecolor:=minor_rulecolor;
7       bm=(ypart(11corner pp)-1u);

```

```

8     makegrid(0,for i=u-4u step u
9         until w+3u: , i endfor)
10    (0,for i=0 step u
11        until h+2u+1: , i endfor)
12    makegrid(0,for i=u-4u step u
13        until w+3u:, i endfor)
14    (0,for i=0 step -u
15        until bm-1u: , i endfor)
16    pickup pencircle scaled major_rulewidth;
17    rulecolor:= major_rulecolor;
18    makegrid(0,for i=0 step 10u until w+2u+1u: ,
19        i endfor)
20    (0,for i=0 step 10u until h+2u+1: ,
21        i endfor);
22    makegrid(0,for i=0 step 10u
23        until w+2u+1u: , i endfor)
24    (0,for i=0 step -10u
25        until bm-1u: , i endfor);
26    draw (0,bm)--(w,bm) withcolor rulecolor;
27    makelabel.lft("0", (0u,0u));
28    makelabel.bot("0", (0u,bm));
29    makelabel.bot("10", (10u,bm));
30    makelabel.bot("20", (20u,bm));
31    if (w+2u) >= 30u :
32        makelabel.bot("30", (30u,bm)); fi
33    if (w+2u) >= 40u :
34        makelabel.bot("40", (40u,bm)); fi
35    if (w+2u) >= 50u :
36        makelabel.bot("50", (50u,bm)); fi
37    if (w+2u) >= 60u :
38        makelabel.bot("60", (60u,bm)); fi
39    if (w+2u) >= 70u :
40        makelabel.bot("70", (70u,bm)); fi
41    makelabel.lft("0", (-1u,0u));
42    makelabel.lft("10", (-1u,10u));
43    makelabel.lft("20", (-1u,20u));
44    endgroup;
45    fi;
46    enddef;

```

The DocGrid macro requires the width and height of the character as its arguments and adds 2u space around the bounding box of the character before overlaying with the grid. It is invoked by the PrintGrid function through the `endfig` hook, `extra_endfig`, as shown in Listing 20.

Listing 20: The source code of PrintGrid.

```

1 def PrintGrid =
2     picture pp; pp:=currentpicture;
3     pw = xpart(urcorner pp);
4     ph = ypart(urcorner pp);
5     DocGrid(pw,ph);
6 enddef;
7
8 extra_endfig:="if proofing = 0:
9     add_space_around; else: PrintGrid; fi";

```

3.5.7 The vertical conjuncts

The process of generating vertical conjuncts, where two characters are stacked on top of each other, was mentioned in Section 3.4, Reusable glyphs. The corresponding source code in Listing 14 demonstrated how this is achieved. You may notice that the below-base character is always scaled down to ensure typographic appeal and to limit the overall depth of the glyph to a reasonable level. However, this scaling action has the unintended consequence of reducing the stroke width in the bottom character.

To address the limitation of width reduction, a boolean variable called `vconj` is introduced. When set to true, it automatically increases the width of the `thick` and `thin` lines by a factor of 1.35. Since all other stroke widths are derived from these two fundamental dimensions, the overall width dimensions are adjusted proportionately and accurately.

Listing 21: Change of the widths `thick` and `thin` depending on the state of the boolean, `vconj`.

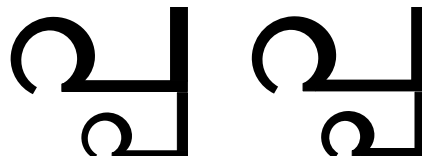
```

1 wa:=1;
2 if vconj:
3     thick:=(1.35*17.2bp)*mag; % width of thick line
4     thin:=(1.35*8.3bp)*mag;   % width of thin line
5 else:
6     thick:=17.2bp*mag;        % width of thick line
7     thin:=8.3bp*mag;         % width of thin line
8 fi

```

The images in Figure 17 show the difference in stroke widths of the below-base character with different states of the `vconj` boolean.

Figure 17: The effect of the boolean `vconj` on the stroke width of the below-base character. The image on the left is when `vconj` is false; observe that the widths of strokes of the below-base character are thinner than its counterpart on the right, with `vconj` true.



3.5.8 The consonant doubling macro

Some consonants, such as ച, ഞ, യ, ള (phonetically similar to *chcha*, *bba*, *yya*, *vva*), behave in a particular fashion when conjuncts with the same consonants are created. This is different from others like ഴ, ഴ്, ഴ് (ppa, nna as in *running*, *gga*), etc., when forming vertical conjuncts with its own copy of the below-base character and മ്, ത്, ള് (mma, ththa, lla as in *culling*), etc., where conjuncts are

formed with their own copy of the post-base character packed horizontally.

Since the shape of the bottom construct is an ideal candidate for a component, we devised one, called `c_cons_dbl`, the source of which is provided in Listing 22.

Listing 22: Consonant doubling macro, `c_cons_dbl`.

```
1 def c_cons_dbl (suffix $)(expr xsh,ysh,wid, hgt) =
2   pstroke_stem ($) (xsh+0b,ysh+0h,-thick,-hgt);
3   dx$a.c=-.15b;
4   stroke_stem ($a)(x$b,y$b-thin,-wid,thin);
5   if not outln_i: ypenstroke stem; fi
6   z$aa=(x$a-.65wid,y$a); penpos$aa(w_n,a_w);
7   z$ab=(x$a.d-.2b,y$a.d+.5wd$ab);
8   penpos$ab(.6thin,a_n);
9   penstroke z$aa.e {dir 263} .. z$ab.e;
10  penlabels($aa,$ab);
11  pstroke_edge((x$ab.r,y$a.d),z$ab.r,z$a.c,z$a.d);
12 enddef;
```

Figure 18: The consonant doubling macro in action.



The macro requires four expression arguments of horizontal shift, vertical shift, width and height, after the suffix argument. A usage example showing the source of the conjunct `᳚` is provided in Listing 23.

Listing 23: Usage of the consonant doubling macro.

```
1 beginfig(20);
2   coor_c_ch_lt (1) (0u,0u);
3   pstroke_c_ch_lt (1);
4   pstroke_stem (2)(0,0,3b,thin);
5   pstroke_stem (3)(x2d,0,thick,1h);
6   c_cons_dbl (4)(x3d,y3a,1.8b,.4h);
7 endfig;
```

3.5.9 The vowel signs

Among all the vowel signs, four of them — `ᳵ`, `ᳶ`, `᳷`, `᳸` — exhibit the tendency to join with consonants to form conjuncts. Another speciality is that each of the first two — `ᳵ` and `ᳶ` — has four different forms depending on the shape and other characteristics of the conjoining consonant. A few examples below illustrate the diverse conjunct formation with the sign `ᳵ`:

- (i) `ᳵ`, `ᳶ` (*ku, ru*);
- (ii) `᳷`, `᳸`, `᳹`, ... (*gu, ju, thu*);
- (iii) `ᳺ`, `᳻`, `᳼`, ... (*nu, ṅu, nnu*) and
- (iv) `᳽`, `᳾`, `᳿`, `ᳺ`, ... (*du, pu, bu, mu*).

Similarly, the longer form `ᳶ` creates four different conjuncts that correspond to the shorter forms cited in the previous list:

- (i) `ᳶ`, `᳷` (*jū, rū*);
- (ii) `᳸`, `᳹`, `ᳺ`, ... (*kū, gū, thū*);
- (iii) `᳻`, `᳼`, `᳽`, ... (*nū, ṅū, nnū*) and
- (iv) `᳾`, `᳿`, `ᳺ`, `᳻`, ... (*dū, pū, bū, mū*).

However, the other two vowel signs — `᳷`, `᳸` — do not tend to create different kinds of conjuncts. The first assumes the uniform shape of `᳷`, `᳸`, `᳹`, `ᳺ`, ... while the conjuncts of the latter have the shape of `᳸`, `᳹`, `ᳺ`, `᳻`, ... There is yet another consonant, `ᳺ`, that conjoins with base characters to make the ‘reph’ form as `ᳺ`, `᳻`, `᳾`, `᳿`, ... that is defined using `make_reph` macro (see Table 4).

We observed that the majority of the conjuncts with the first two vowel signs tend to form rounded variants of `ᳵ` and `ᳶ` (see the examples in item iv of the above two lists); hence, two macros have been designed so that the corresponding conjuncts are created via simple calling of one of these macros augmented with appropriate *x, y* coordinate values at which to attach. The two examples of the conjuncts `᳾` and `᳿` provided in Figure 19 illustrate the process. The corresponding METAPOST sources are in Listing 24.

Figure 19: Formation of conjuncts with rounded vowel signs aligned with the bottom of a horizontal stem.



Listing 24: The conjuncts with rounded vowel forms.

```
1 %%% ᳾ %%%
2 def gl_pu (suffix prx) =
3   outln:=true;
4   gl_pa(prx);
5   make_stem_u (21) (x.prx.3d-.5wd21c,
6     y.prx.3c-.75wd21b);
7 enddef;
8 %%% ᳿ %%%
9 def gl_puu (suffix prx) =
10  outln:=true;
11  gl_pa(prx);
12  make_stem_uu (21) (x.prx.3d-(x21c.r-x21b),
13    y.prx.3c-.5wd21b);
```

Two macros, `make_stem_u` and `make_stem_uu` are used to align the `ᳵ` and `ᳶ` respectively with the

horizontal stem of a . One may also notice a boolean `outln` has been set true to improve the alignment process (this will be explained in detail in Section 3.5.10 on `outline` mode). The macro requires two arguments, the horizontal and vertical coordinates, where the rounded object will align with the stem. The sources in Listing 25 will amplify this further.

Listing 25: The conjuncts with rounded vowel forms conjoining with horizontal stems of consonants.

```

1 %%% ̣-sign %%%
2 def make_stem_u (suffix $) (expr xsh,ysh) =
3   coor_vl_round_u_alt ($) (xsh,ysh);
4   pstroke_vl_round_u_alt (21);
5   if not noreverse: stem:= reverse stem; fi
6   find_outlines(rmpath,stem)(P);
7   for i=1 upto P.num: ypenstroke P[i]; endfor
8 enddef;
9 %%% ̣-sign %%%
10 def make_stem_uu (suffix $) (expr xsh,ysh) =
11   coor_vl_round_uu_alt ($) (xsh,ysh);
12   pstroke_vl_round_uu_alt (21);
13   if not noreverse: stem:= reverse stem; fi
14   find_outlines(rmpath,stem)(P);
15   for i=1 upto P.num: ypenstroke P[i]; endfor
16 enddef;

```

As you might surmise, there are also variant forms of the macros to accommodate rounded bottom curves where the vowel sign is expected to align. The source code is provided in Listing 26, and the corresponding output in Figure 20.

Listing 26: The conjuncts with rounded vowel forms conjoin with bottom curves of consonants.

```

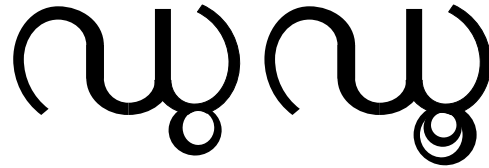
1 %%% ̣ %%%
2 def gl_Du (suffix prx) =
3   outln:=true;
4   gl_Da (prx);
5   make_round_u (21) (x.prx.4a+20+(x21c-x21b),
6                     y.prx.4a.r-.75wd21b);
7 enddef;
8 %%% ̣ %%%
9 def gl_Duu (suffix prx) =
10  outln:=true;
11  gl_Da (prx);
12  make_round_uu (21) (x.prx.4a,y.prx.4a+.1wd21b);
13 enddef;

```

The other forms of these two vowels are limited to a very few consonants and therefore created individually by slicing the paths at appropriate locations and adding necessary components from the `ml-shape-lib.mp` library.

A variety of macros, listed in Table 4, defined in `ml-vlsigns-lib.mp`, can be used to create conjuncts with different vowels depending on the shape of the consonant and the final form of the conjunct.

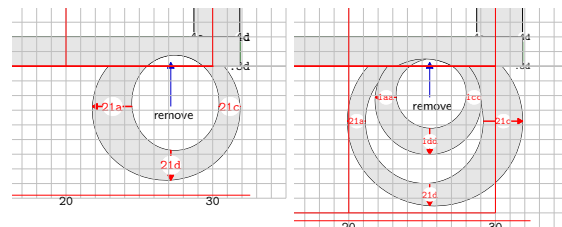
Figure 20: Formation of conjuncts with rounded vowel signs aligned with the bottom of a curve.



3.5.10 outline mode

The rounded forms of the vowel signs ̣ and ̣̣ present the unusual challenge of excising the portions not encompassed by their curved structures, when juxtaposed on a horizontal stem or rounded path. This typographical intricacy is meticulously adhered to by all the fonts crafted and published by the Rachana Institute of Typography. Figure 21 shows two illustrative images, demonstrating this requisite nuance, ensuring harmonious integration of the rounded vowel signs within the textual fabric.

Figure 21: Example images showcasing the removal of uncovered parts from ̣ and ̣̣ .



Implementing this particular requirement in METAFONT presents itself as a formidable undertaking. Happily, however, the bundled `plain_ex.mp` library accompanying the METATYPE1 package [4] emerged as a veritable fairy godmother to navigate this very challenge. Thus we will share a snippet from the self-documented code base of `plain_ex.mp`, perfectly suited for the readers seeking enlightenment in this matter.

The problem can be stated as follows: two paths are given (precisely: expressions of type `path`); assume that the positively directed (anti-clockwise) path accomplishes filling, and negatively directed (clockwise)—erasing; the task is to find the outline of the resulting (visible) figure. Such a task is known as “removing overlaps” which seems too narrow for such a complex operation. Actually, the basic macro of that part, i.e., `find_outlines`, accomplishes set-theory operations: sum, difference and product, depending on the turning number of the input paths. The illustration below demonstrates the results

We thus developed a font build tool in Python, heavily utilizing FontForge Python libraries, driven by a configuration file to (1) import the SVG outlines into glyph slots; (2) assign Unicode codepoints to them; (3) set left/right bearing, width and other properties; (4) set font metadata and (5) generate the final OpenType font formats such as TTF, OTF, WOFF. Thus, the overall development workflow is:

METAPOST \rightarrow SVG \rightarrow FontForge + scripts
 \rightarrow OTF/TTF/WOFF

Each character is defined in its own METAPOST file following the glyph naming convention, which is then compiled to generate an SVG file in a directory for the variant being generated (`serif-regular`, `serif-thin`, etc.), bearing the same glyph name, for e.g., `ml_a.mp` \rightarrow `ml_a.svg`. In general, all the glyph names in the METAPOST files, generated SVG files, config files and OpenType layout rules follow the convention explained in Section 3.4.1.

Once the SVG files are generated, the next round of scripts, driven by a configuration file that associates glyph names with its Unicode codepoint and other properties, are run. The Malayalam script has many conjuncts that do not have individual codepoints themselves but are formed by a combination of basic Unicode codepoints. These glyphs do not need a mapping entry for codepoint but may need other properties.

The configuration file also holds the font metadata such as family name, PostScript name, variant name, version, license, etc. Crucially, it should also be possible to adjust the left/right side bearings of all the glyphs. A default bearing value suffices for most; but quite a few need either a smaller or negative value, for e.g. ി (U+0D3F, glyph name `i1`) and ൿ (U+0D4D, glyph name `xx`) which extends outside the left margin (negative side bearing). In addition, setting the width for non-printable characters like `space` is crucial, as this affects word spacing.

The common `ini` configuration file format is used; and a self-explanatory sample file is provided in Listing 27.

Listing 27: Configuration file for font building.

```

1 # Metadata
2 [font]
3 family=Sayahna
4 name=Sayahna-Regular
5 version=0.9.1
6 ascent=820
7 descent=180
8 copyright=Copyright 2021-2023 Rachana
9     Institute of Typography
10     <info@rachana.org.in>
11
```

```

12 # SVG, OpenType feature file, Unicode mapping
13 [source]
14 glyphdir=svgs-regular/
15 featurefile=features/sayahna-feature.fea
16 ucglyphmapfile=tools/rit-ml-uc-glyph.map
17
18 # Width of specific glyphs
19 [width]
20 space=300
21
22 # Default and overridden left, right bearings
23 [bearing]
24 default=30,40
25 i1=-74,30 # negative left bearings
26 i2=-80,30
27 r1=-112,30
28 xx=-57,30
29 y2=-70,30
30 y2u1=-70,30
31 y2u2=-70,30
32 v2=-40,30
```

The font build tool performs a number of steps:

- Assemble all the SVG format glyphs found in the target directory `glyphdir`.
- Remove overlap of outlines.
- Add additional glyphs/codepoints for space, zero-width joiner/non-joiner (U+0020, U+200D, U+200C), etc. These are used in certain character combinations.
- Adjust side bearings/widths of glyphs as specified in the config file.
- Set metadata.
- Assign the Unicode code points for all base characters, given in a simple mapping file `rit-ml-uc-glyph.map`.
- Apply the OpenType shaping rules for Malayalam [8], given in a file `sayahna-feature.fea`.
- Set kerning values as specified.
- Finally, produce the font in any desired format(s), such as TTF, OTF or WOFF2.

All these steps are fully automated by a few Makefile targets. Thus, if a font developer wishes to make amendments to a glyph, she can make the changes in the METAPOST file and then run a `make font` (or equivalent) command that goes through all the above steps and generates the revised font a few moments later.

5 Data availability

All of the source code of the project, including METAPOST libraries, glyph definitions, font build tools, OpenType shaping rules, Unicode mapping files and other configuration files, is available at gitlab.com/rit-fonts/Sayahna-font under free software licenses, such as the LPPL and the OFL.

References

- [1] J. Hobby, et al. *MetaPost: A Users Manual*.
tug.org/metapost
 - [2] K. Hosny. The Punk Nova font. 2010.
github.com/aliftype/punk-otf
 - [3] K.H. Hussain, R. Chitrajakumar, et al. The Rachana font. 2023.
gitlab.com/rit-fonts/RIT-Rachana
 - [4] B. Jackowski, J.M. Nowacki, P. Strzelczyk. Programming PostScript Type 1 fonts using METATYPE1: Auditing, enhancing, creating. *TUGboat* 24(3), 2003. Proceedings of EuroT_EX 2003.
tug.org/TUGboat/tb24-3/jackowski.pdf
 - [5] D.E. Knuth. *The T_EXbook*, vol. A of *Computers & Typesetting*. American Mathematical Society and Addison Wesley, Reading, Massachusetts, 1986.
 - [6] D.E. Knuth. *The METAFONTbook*, vol. C of *Computers & Typesetting*. American Mathematical Society and Addison-Wesley, Reading, Massachusetts, 1986.
 - [7] Rachana Institute of Typography. Malayalam glyphs, components and sectors. 2022.
rachana.org.in/docs/rit-comp-list.pdf
 - [8] K.V. Rajeesh. Malayalam OpenType shaping rules. 2020. gitlab.com/rit-fonts/malayalam-shaping/
 - [9] Unicode. Malayalam: 0D00–0D7F.
unicode.org/versions/latest/ch12.pdf
unicode.org/charts/PDF/U0D00.pdf
 - [10] Wikipedia. Brahmic scripts. 2019.
wikipedia.org/wiki/Brahmic_scripts
 - [11] Wikipedia. Languages of India. 2023.
wikipedia.org/wiki/Languages_of_India
- ◇ C. V. Radhakrishnan
River Valley Technologies, River Valley
Campus, Malayinkeezh
Trivandrum 695571, India
`cvr (at) river-valley (dot) org`
<http://river-valley.com>
ORCID 0000-0001-7511-2910
 - ◇ K. V. Rajeesh
Rachana Institute of Typography
`rajeesh (at) rachana (dot) org (dot) in`
<https://rachana.org.in/>
 - ◇ K. H. Hussain
Rachana Institute of Typography, Jagathy,
Trivandrum 695014, Kerala, India
`hussain (at) rachana (dot) org (dot) in`
<https://rachana.org.in>