Including Mathematica graphics in LaTeX documents: how the TMG (text in graphics) package works

Michael P. Barnett*

November 5, 2010

Introduction

The TMG (text in graphics) package [1] supports the construction of diagrams by MATHEMATICA† Release 7, for inclusion in LaTeX documents. The successive sections that follow discuss:

1. the linear measure, in LaTeX documents, of diagrams that were produced by MATHEMATICA graphics and incorporated by `\includegraphics` commands,
2. how to include only parts of these diagrams, by clipping (cropping),
3. the TMG function `export` that wraps the MATHEMATICA `Export` command to write graphics output,
4. the TMG functions that support the use different fonts, and obtain information about the shapes of the characters,
5. the TMG functions that I use to determine the widths of characters,
6. the detailed structure of the line segments drawn by MATHEMATICA `Line` commands,
7. the TMG functions that determine the character offsets needed to align the strings in multiple `Text` commands,
8. the TMG functions to encode character strings that represent formated text in multiple fonts (I use these to annotate diagrams produced by MATHEMATICA graphics).

1 The linear measure of graphics objects

The PDF file that represents the rhombus in Fig. 1 was produced by the MATHEMATICA statement

```
Export["rhomb100.pdf", Show[Graphics[
   {AbsoluteThickness[.01], Line[{{0,50}, {50,0}, {100,50}, {50,100}, {0,50}}]},
    PlotRange -> {{0,100}, {0,100}}, AspectRatio -> 1, ImageSize -> {100,100}]]]
```

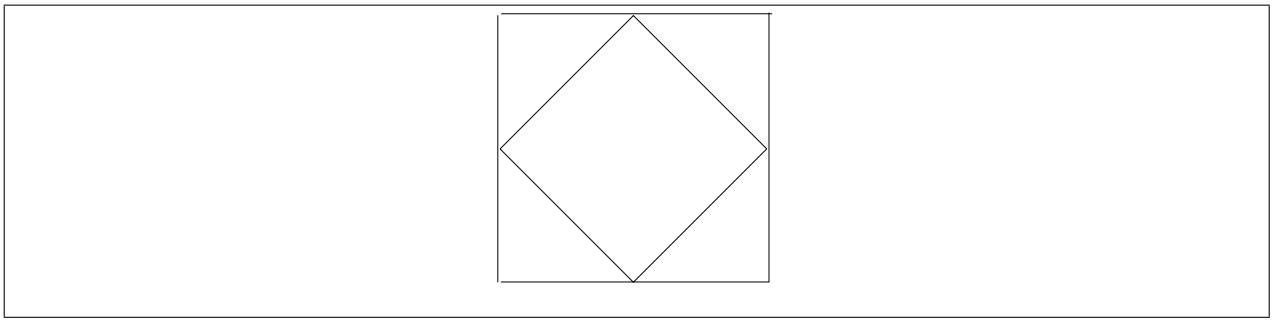

Figure 1: A rhombus.

The sub-expression `Line[{{0,50},{50,0},{100,50},{50,100},{0,50}}]` draws the polygon with apexes (0,50), (50,0), (100,50) and (50,100) in a Cartesian coordinate system that is defined, in relation to the screen and to

*Meadow Lakes, Hightstown, NJ 08520, michaelb@princeton.edu

†MATHEMATICA is a registered trademark of Wolfram Research Inc.

the printout of the exported PDF file by the `PlotRange`, `AspectRatio` and `ImageSize` values that are specified in the `Export` statement. The x coordinates in this system span the range (0, 100) because the 1^{st} item in the `PlotRange` is `{0,100}`. The y coordinates in the system span the range (0, 100) because the 2^{nd} item in the `PlotRange` is `{0,100}`. The aspect ratio of the diagram is 1, because of the `AspectRatio` item. Because of these settings, 1 unit in an x or y coordinate corresponds to 1 printer's point of linear measure in a LaTeX document that incorporates the PDF file, by means of an `includegraphics` command that does not scale or clip. The PDF file for Fig. 2 was produced by the statement containing `"circle100.pdf"` and `Circle[{50,50},50]`, in place of `"rhomb100.pdf"` and the `Line` expression, in the statement that led to Fig. 1.

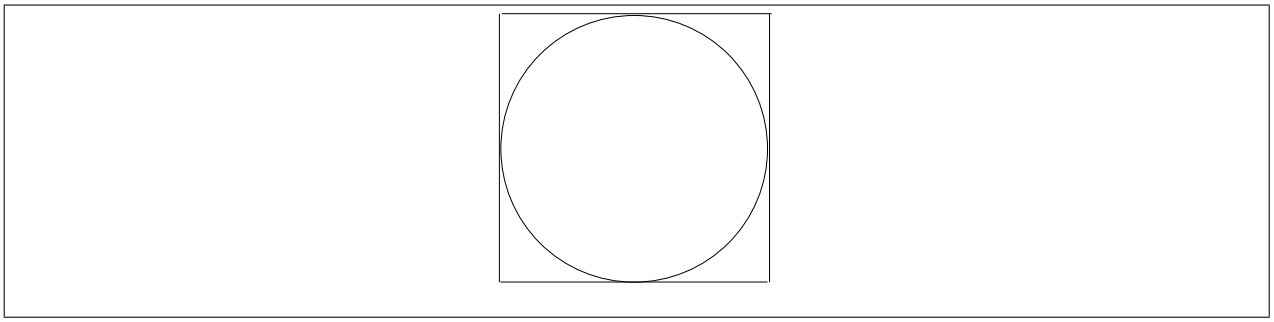

Figure 2: A circle.

The square around the rhombus in Fig. 1 is drawn by LaTeX `\rule` commands.

```
\begin{figure}[!htp]
\begin{center}
\noindent\rule{102pt}{.1pt}
\\[-1.5pt]\noindent\rule{.1pt}{100.5pt}\hspace{0.8pt}\includegraphics{rhomb100.pdf}
\hspace{.8pt}\rule{.1pt}{101.5pt}\hspace{-101pt}\rule{101pt}{.1pt}
\caption{\label{rhomb100}A rhombus.}
\end{center}
\end{figure}
```

This works as follows.

1. The `\rule{102pt}{.1pt}` command draws the top of the square, allowing 2 points for line thickness.
2. The `\\[-1.5pt]` moves the writing position to the left edge of the local print region, and downward by the default interline spacing (9 pt) plus the clearance 1.5 pt found by trial and error.
3. The `\rule{.1pt}{100.5pt}` command draws the left side of the square. The command pushes the writing position down by the amount needed for the top of the rule to fit below the baseline of the previous line.
4. The `\includedgraphics` command brings in the named PDF file, with its lower edge along the line that would constitute the baseline of text that was substituted at this point.
5. The `\rule{.1pt}{101.5pt}` draws the right side of the square upwards from the same baseline.
6. The `\hspace{-101pt}\rule{101pt}{.1pt}` commands move the writing position back to the left edge of the diagram and draw the lower side of the square.

Fig. 3 shows the depression of baselines by LaTeX `\rule` commands. It is produced, without including a PDF file, by

```
\begin{figure}[!htp]
\begin{center}
\noindent first line  \\second line  \\\noindent\rule{126pt}{.1pt}
\\[-1.5pt]\noindent\rule{.1pt}{20.5pt}aligns with base of first vertical rule
\\[-1.5pt]\noindent\rule{.1pt}{30.5pt}aligns with base of second vertical rule
\caption{\label{depressionByRule}Baseline depression by \LaTeX\ rule.}
\end{center}
\end{figure}
```

In general, 1 unit in the `Point`, `Line`, `Circle`, `Text`, `Polygon` and other graphics primitives that contain (x, y) coordinates corresponds to 1 printer's point of linear measure in a LaTeX document, when

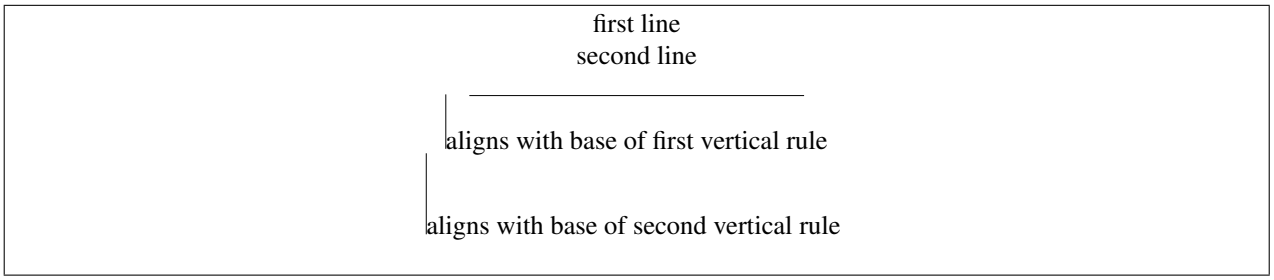

Figure 3: Baseline depression by LaTeX rule.

`PlotRange -> {{`x_ℓ, x_h`},{`y_ℓ, y_h`}}, AspectRatio ->` $\frac{y_h - y_\ell}{x_h - x_\ell}$`,ImageSize -> {`$x_h - x_\ell, y_h - y_\ell$`}`

When a PDF file is included, it pushes the baseline of contextual material downward, if necessary, to align with its lower boundary. This was not necessary in Figs. 1 and 2. But it does occur in the setting of Fig. 4, that is produced by the following LaTeX code.

```
\begin{figure}[!htp]
\begin{center}
{\sc pdf} files that are brought in by {\sc includegraphics}
\\commands push the adjacent text down to a common baseline.
\\circle:\includegraphics{circle100.pdf}rhombus:
\includegraphics{rhomb100.pdf}base aligned.
\\This sentence completes the demonstration of alignment around
\\{\sc pdf} files brought in by {\sc includegraphics} commands.
\caption{\label{alignment}Baseline alignment of included files.}
\end{center}
\end{figure}
```

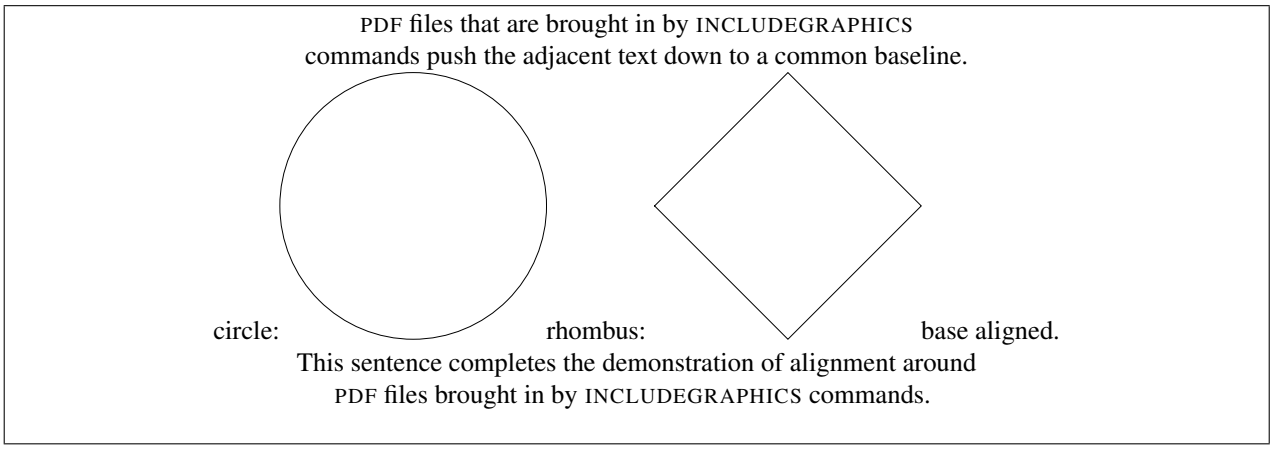

Figure 4: Baseline alignment of included files.

2 Clipping (cropping)

Fig. 5 was produced as follows. I wrote a simple MATHEMATICA function `polygon[`n`,{`x_1, y_1`},{`x_2, y_2`}]` that draws an n-sided regular polygon, with edges $\{x_1, y_1\}$ to $\{x_2, y_2\}$, $\{x_2, y_2\}$to $\{x_3, y_3\}$, ..., that each turns left from its predecessor. The function actually returns a list of two items

1. the MATHEMATICA `Line` expression that depicts the polygon, and
2. the $(x_{lo}, y_{lo}, x_{hi}, y_{hi})$ values for its bounding box in the same coordinate system as $(x_1, y_1), (x_2, y_2)$.

The row of polygons was defined in the plot range x = 0 to 468, y = 0 to 648, corresponding to a 6.5 $\times$ 9 inch print region. The bases of successive polygons start at multiples of 30 units and are 10 units long on the screen, and 30 and 10 printers points long in the typeset diagram.

```
polygonList = Table[polygon[n, {30*(n-3), 600}, {30*(n-3)+10, 600}] // N,{n, 3, 8}];
```

The following statements extracted the bounding boxes from the output of the `polygon` expression and wrote the PDF file `polygons.pdf` that represents Fig. 5.

```
polygons = #[[1]]& /@ polygonList;  export[polygons]
```

As explained in more detail in the next section, the command `export[polygons]` wraps

```
Export[
 "polygons.pdf",
 Show[
  Graphics[
   {polygons,
    PlotRange->72*{{0,6.5},{0,9}}, AspectRatio->9/6.5, ImageSize->72*{6.5,9}}]]]
```

The values of `PlotRange`, `AspectRatio` and `ImageSize` are defaults that map the screen onto the default 6.5"$\times$9" print region, with the origin at the lower left corner.

```
PlotRange -> 72 * {{0, 6.5}, {0, 9}}, AspectRatio -> 9/6.5, ImageSize -> 72 * {6.5, 9}
```

This makes each bounding box for the `viewport` in the `\includegraphics` statement the same as the bounding box in the (x, y) system used in the `Point`, `Line` and other primitives used to create the graphics object. The bounding box of the row of polygons was found by trivial MATHEMATICA statements that took the minima and maxima of the x and y coordinates of the individual polygons. These were rounded manually, and a half point allowed at each edge to avoid clipping the actual line. The results are consistent with elementary trigonometry. This gives the values 0, 0, $\ell \cos \frac{2\pi}{5}$, $\frac{\ell}{2}$, $\frac{\ell\sqrt{2}}{2}$ for the projections on each side of the base for the triangle, square, pentagon, hexagon and octagon, with side length ℓ. The corresponding heights are $\frac{\ell\sqrt{3}}{2}$, ℓ, $2\ell \sin \frac{3\pi}{5} \cos \frac{\pi}{5}$, $\ell\sqrt{3}$, $\ell(1 + \sqrt{2})$. The PDF file for Fig. 5 was produced by

```
\begin{figure}[!htp]
\begin{center}
regular polygons with 3--8 edges
\\$\Longrightarrow$\includegraphics[viewport=0 599 168.5 625,clip]
{polygons.pdf}$\Longleftarrow$
\\the text above and below the polygons and the \LaTeX\ arrows
show the extent of the included file
\caption{\label{polygons}A row of regular polygons.}
\end{center}
\end{figure}
```

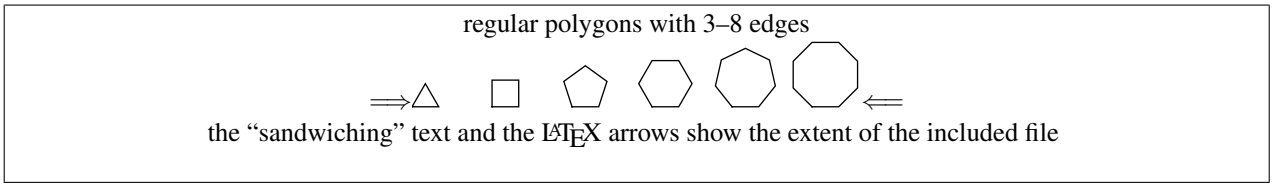

Figure 5: A row of regular polygons.

The individual polygons are clipped and included separately in Fig. 6 by the following LaTeX coding. The tips of the arrows reach the edges of the included material.

```
\begin{figure}[!htp]
\begin{center}
$\Downarrow$\\$\Longrightarrow$\includegraphics
[viewport=0 599.5 10.5 609.5,clip]{polygons.pdf}$\Longleftarrow$
\\$\Updownarrow$\\$\Longrightarrow$\includegraphics
```

```
[viewport=30 599.5 40.5 610.5,clip]{polygons.pdf}$\Longleftarrow$
\\$\Updownarrow$\\$\Longrightarrow$\includegraphics
[viewport=56.5 599.5 73.5 615.9,clip]{polygons.pdf}$\Longleftarrow$
\\$\Updownarrow$\\$\Longrightarrow$\includegraphics
[viewport=84.5 599.5 104.5 617.8,clip]{polygons.pdf}$\Longleftarrow$
\\$\Updownarrow$\\$\Longrightarrow$\includegraphics
[viewport=113.2 599.5 136.8 622.4,clip]{polygons.pdf}$\Longleftarrow$
\\$\Updownarrow$\\$\Longrightarrow$\includegraphics
[viewport=142.4 599.5 168.5 624.7,clip]{polygons.pdf}$\Longleftarrow$
\\$\Uparrow$
\caption{\label{split}The individually clipped polygons.}
\end{center}
\end{figure}
```

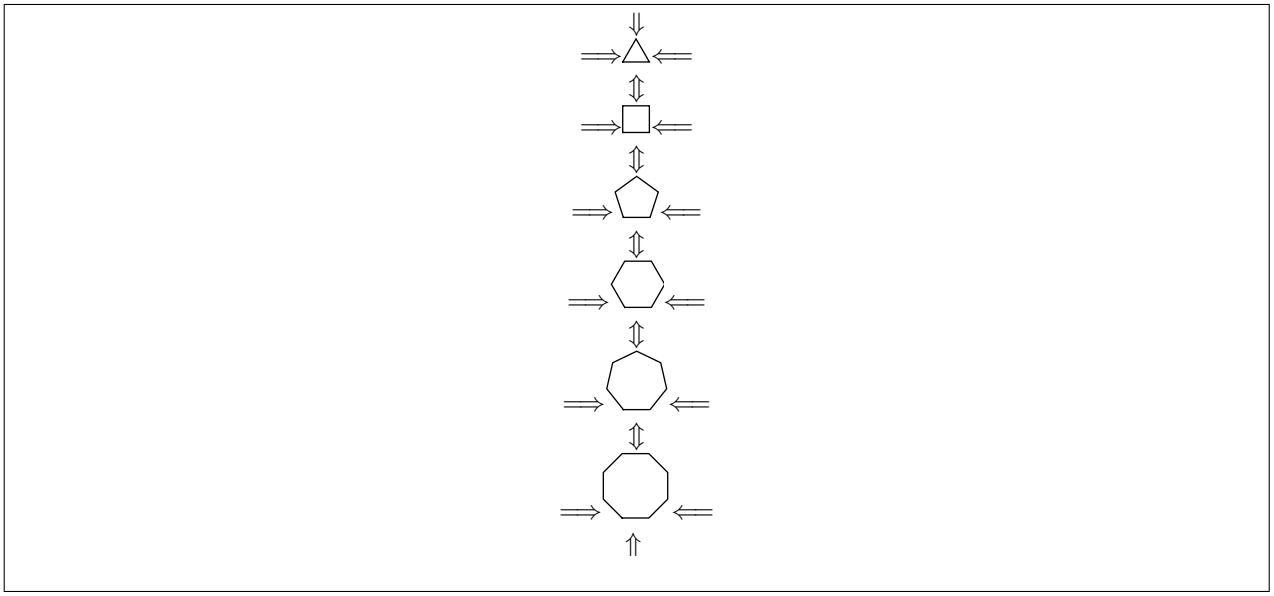

Figure 6: The individually clipped polygons.

3 The export function in tmg

The `export` command has the following forms.

1. `export["`*name*`",`*outputList*`]` : This writes the file *name*`.pdf` that represents the diagram specified by *outputList*, sized by the current setting of `window`, discussed below.
2. `export[`*name*`,`*outputList*`]` : This is executed conditionally when *name* does not have a value. It is evaluated as `export["`*name*`",`*outputList*`]`.
3. `export[`*name*`=`*outputList*`]` : This is executed conditionally when *name* did not have a value immediately before execution. It is evaluated as `export["`*name*`",`*outputList*`]`, and assigns *name* to *outputList*.
4. `export[`*name*`]` : This is executed conditionally when *name* has a value. Denote this by *outputList*. The statement is evaluated as `export["`*name*`",`*outputList*`]`.

In case 1, *name* (and hence *name*`.pdf`) must be valid UNIX file names. In all cases, *outputList* should be the MATHEMATICA expression for a valid graphics object. If it is not, the resulting picture will be flawed or unreadable.

The internal operation of `export` is trivial in cases 1 and 2. In cases 3 and 4, the `HoldAll` attribute preserves the held argument literally. Elementary string operations then

1. convert the held argument to a character string,

2. determine whether the statement conforms to type 3 or 4,
3. dissects it appropriately,
4. constructs the string representation of the corresponding statement in the style of type 1,
5. releases this for execution.

The TMG object `window` defaults to

```
Sequence[PlotRange->72*{{0,6.5},{0,9}}, AspectRatio->9/6.5, ImageSize->72*{8.5,9}]
```

This maps the screen onto the default 6.5"×9" print region, with the origin at the lower left corner. The specification `window` can be reassigned freely.

The statement `exportBounded[`$args$`]` writes the PDF file that produces the same diagram as `export[`$args$`]` without the need to clip in the `\includegraphics` command. It uses the TMG function `boundingBox`. This finds the extremes of `Point`, `Line`, `Circle`, `Polygon`, `Disk` and `Text` objects. The function `offsetAdjust`, that is described in §7, uses `exportBounded` to write the PDF files that produce *e.g.* Figs. 19 and 20, so that these can be collected by `pdfFileConsolidation` (described there, too) without excessive white space.

4 Font shapes

I use the Courier, Times Roman and Symbol fonts to annotate diagrams. These are AFM fonts [2]. In the installation where this work was done, the directory `/usr/licensed/Mathematica-7.0/SystemFiles/Fonts/AFM/` contains the font metric tables for all the type faces that can be accessed in `Text` statements. In this directory, the tables for these fonts are called `Courier.afm`, `Times-Roman.afm`, ... The assignments of `fontMetricPath` and `fontNameList` set the latter to the list of font names that can be used in `Text` statements. The function `showFonts` displays a list of character sets, truncated to fit the line length of the body text. The statement

```
showFonts[fontShapesAll, fontNameList // Reverse, 7, Range[10,630,7]]
```

produced the PDF file for Fig. 7, that displays the entire set in 9 point, spaced vertically by 9 points. Four fonts produce characters with heights that are several times the point size. These overlap vertically in Fig. 7. The fonts are redrawn in Figs. 8 and 9 to avoid this. In `showFonts[`$title, fontList, size, yList$`]`,

1. $title$ makes the name of the output file $title$`.pdf`,
2. $fontList$ is the list of names of the fonts to be displayed, going upward on the page,
3. $size$ is the font size,
4. $yList$ is the list of vertical coordinates of the baselines of the fonts.

The function works through the items in $fontList$. The 1^{st} `Text`statement displays the font name in Courier type. The 2^{nd} `Text` statement displays the characters with ASCII codes 33–255. The normal height fonts and tall fonts are displayed separately in Figs. 8 and 9. These are represented by the PDF files that the next two statements export.

```
showFonts[fontShapesNormal,
 Delete[fontNameList, Partition[Range[20,23], 1]] // Reverse, 9, Range[10,640,10.7] ]

showFonts[fontShapesTall, fontNameList[[Range[20,23]]] // Reverse, 8, Range[60,180,40]]
```

The AFM font tables contain names for the characters. The function `characterNames[`$font$`]` displays these. Thus

```
characterNames[Symbol] ⟹ {Symbol,
{space, exclam, universal, numbersign, existential, percent, ampersand, suchthat, ...
greaterequal, multiply, proportional, partialdiff, bullet, divide, notequal, ...
bracketrightex, bracketrightbt, bracerighttp, bracerightmid, bracerightbt}}
```

The function uses elementary string operations in a direct way. The list `nameLines` consists of strings typified by

```
" C 187 ; WX  449 ; N guillemotright   ; B   42   53  412  427 ;"
```

Fig. 10 shows the characters in the Symbol font that can be produced by the `FromCharacterCode` function of MATHEMATICA, from basic and extended ASCII, and a few more 3-digit numbers. In the figure, each triple consists of (1) the character in the Symbol font that the code represents, or a black rectangle if none is represented, (2) the corresponding key (on the keyboard), when there is one, and (3) the numerical code.

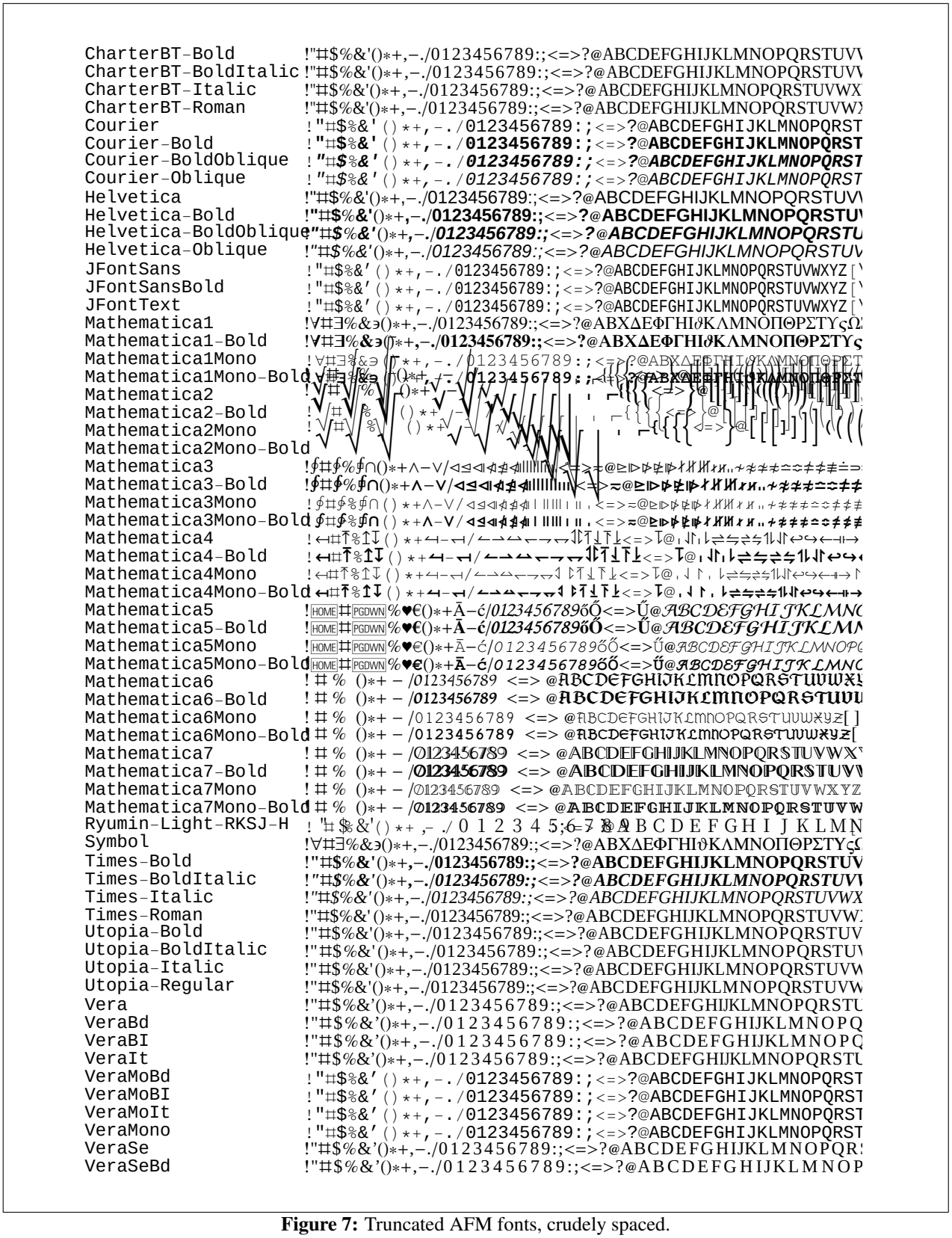

Figure 7: Truncated AFM fonts, crudely spaced.

Figure 8: Normal height AFM fonts.

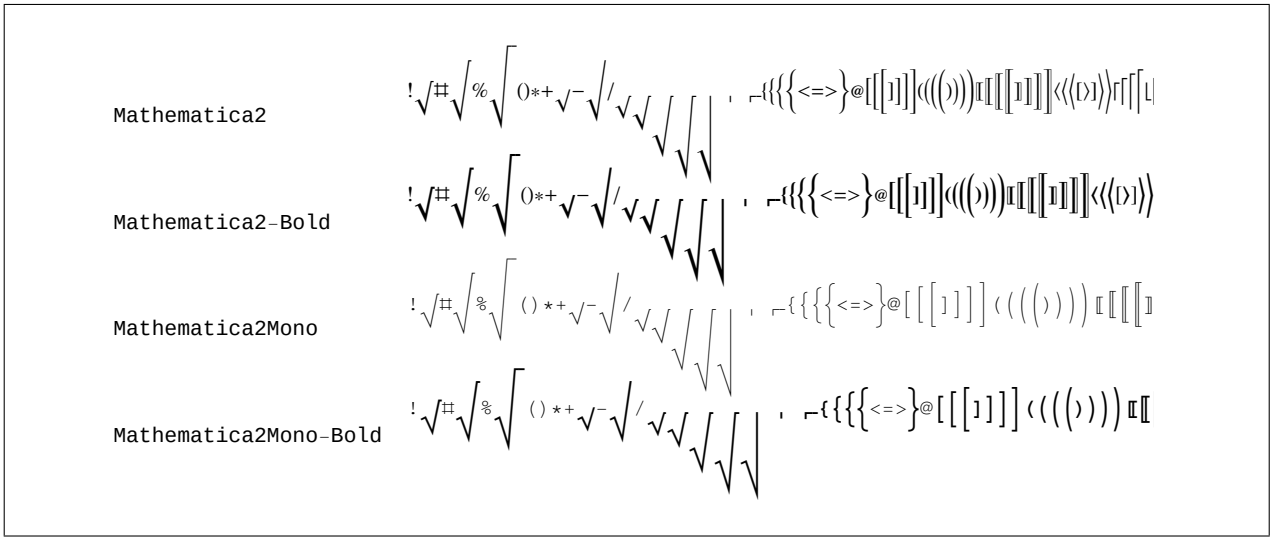

Figure 9: Tall AFM fonts.

! ! 033	∀ " 034	# # 035	∃ $ 036	% % 037	& & 038	∋ ' 039	((040
)) 041	∗ * 042	+ + 043	, , 044	− - 045	. . 046	/ / 047	0 0 048
1 1 049	2 2 050	3 3 051	4 4 052	5 5 053	6 6 054	7 7 055	8 8 056
9 9 057	: : 058	; ; 059	< < 060	= = 061	> > 062	? ? 063	≅ @ 064
Α A 065	Β B 066	Χ C 067	Δ D 068	Ε E 069	Φ F 070	Γ G 071	Η H 072
Ι I 073	ϑ J 074	Κ K 075	Λ L 076	Μ M 077	Ν N 078	Ο O 079	Π P 080
Θ Q 081	Ρ R 082	Σ S 083	Τ T 084	Υ U 085	ς V 086	Ω W 087	Ξ X 088
Ψ Y 089	Ζ Z 090	[[091	∴ \ 092	]] 093	⊥ ^ 094	_ _ 095	‾ ` 096
α a 097	β b 098	χ c 099	δ d 100	ε e 101	φ f 102	γ g 103	η h 104
ι i 105	φ j 106	κ k 107	λ l 108	μ m 109	ν n 110	ο o 111	π p 112
θ q 113	ρ r 114	σ s 115	τ t 116	υ u 117	ϖ v 118	ω w 119	ξ x 120
ψ y 121	ζ z 122	{ { 123	| | 124	} } 125	∼ ~ 126	ϒ 161	′ 162
≤ 163	⁄ 164	∞ 165	ƒ 166	♣ 167	♦ 168	♥ 169	♠ 170
↔ 171	← 172	↑ 173	→ 174	↓ 175	° 176	± 177	″ 178
≥ 179	× 180	∝ 181	∂ 182	• 183	÷ 191	≠ 208	≡ 215
≈ 221	… 222	∣ 240	⏐ 253	⎯ 254	← 256	↵ 257	ℵ 258
ℑ 259	℘ 268	ℜ 269	™ 274	∏ 275	√ 276	⋅ 277	¬ 300
∧ 301	∨ 305	⇔ 322	⇐ 336	⇑ 337	⇒ 352	⇓ 353	◊ 402

Figure 10: Courier equivalents and ASCII codes for the Symbol type face.

To produce Fig. 10, I first ran `charactersCodesAndKeys[Symbol, Range[33,512], all]`. This showed the result of trying to convert every integer in the range 33–512 to a character in `Symbol` font. Then I typed the list of codes that gave valid characters (leaving a few that did not, to show their effect), and used this as the 2^{nd} argument in the statement `charactersCodesAndKeys[Symbol, symbolCodeList]`. This wrote the file `codesAndKeys.pdf`, which produced the figure.

Three-digit octal numbers are also used to represent characters in MATHEMATICA. These numbers have the head `Symbol`. If v stands for an octal number, the system converts `"`v`"` to a character string. This is either (1) a valid character, (2) a null string or (3) a string (*e.g.* `"$Failed"`) whose length exceeds 1. Cases 2 and 3 are not valid characters, and forming a case 3 string usually triggers a diagnostic. Figs. 12 and 13 show the correspondence of octal numbers and characters in Courier, Times-Roman and Symbol fonts. The PDF files were produced by

```
octalCharacterMap[octalCourierAndTimes, {"Courier", "Times-Roman"}]

octalCharacterMap[octalSymbol, Symbol]
```

The function `octalCharacterMap` deals with some MATHEMATICA pathology. It assigns

1. `octalNumberList` to the list of strings
   `{"000", "001", ..., "007", "010",..., "017", "020", ..., "777",`
2. `octalListRaw` to these with `"\\"` prepended, *i.e.* with the `FullForm` `"\\001",...,`
   (a) acting on these with `ToExpression`, to give symbols with the appearance `\101`, ...,
   (b) acting on these with `ToString`, to give `"A"`, ...,
3. `octalListRawLengths` to the string lengths of the elements of `octalListRaw`,
4. `positionsOfDudItems` to the indexes of elements of `octalListRaw` that are longer than 1,
5. `octalListRaw` found by substituting `""` in place of each invalid element.

The rest of the procedure displays, for each octal value 000_8 to 777_8:

1. the corresponding character in the font under consideration, when one exists,
2. a space or black rectangle, otherwise.

In `octalCharacterMap` and several other functions, I abbreviate the details of a font by names such as `T10` for 10 point Times-Roman. These are generated by `makeAbbreviationsForFontSpecifications`. Each name consists of `font`, then `C`, `T` or `S` for Courier, Times-Roman and Symbol, respectively, and the point size as a 2-digit number. Thus `fontS09` has the value `Sequence[FontFamily->ToString[Symbol], FontSize->9]`.

Special characters in the UNICODE set supported by MATHEMATICA can also be included between quote marks in `Text` statements. The usage is shown by the following statement. It produced the PDF file that represents Fig. 11.

```
export[unicodeExample, Text[Style[
 "\[Aleph]\[GreaterGreater]\[Integral]\[ScriptCapitalA]",  FontSize -> 20],  {50,600}]]
```

The UNICODE set supported by MATHEMATICA is displayed on

`http://reference.wolfram.com/mathematica/guide/ListingOfNamedCharacters.html`.

The usage is shown by the following Fig. 11 and the statements that produced it. Figs. 12 and 13 show the OCTAL

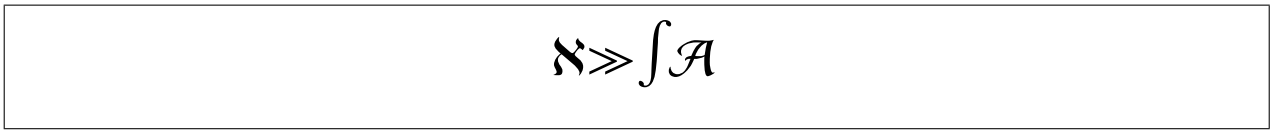

Figure 11: A sequence of UNICODE characters.

codes for characters in the fonts that I use most often.

5 Character widths

When contiguous text in a diagram has to be displayed by several `Text` statements, their horizontal coordinates depend on the widths of the successive pieces. Each of these widths is the sum of widths of its constituent characters. Figs. 14–17 show how I start to find character widths. These figures show the top and bottom of the displays that `fontWidthDisplay["Times-Roman",`n`]` produces, with $n = 1$ and 2. In the statements for Fig. 14,

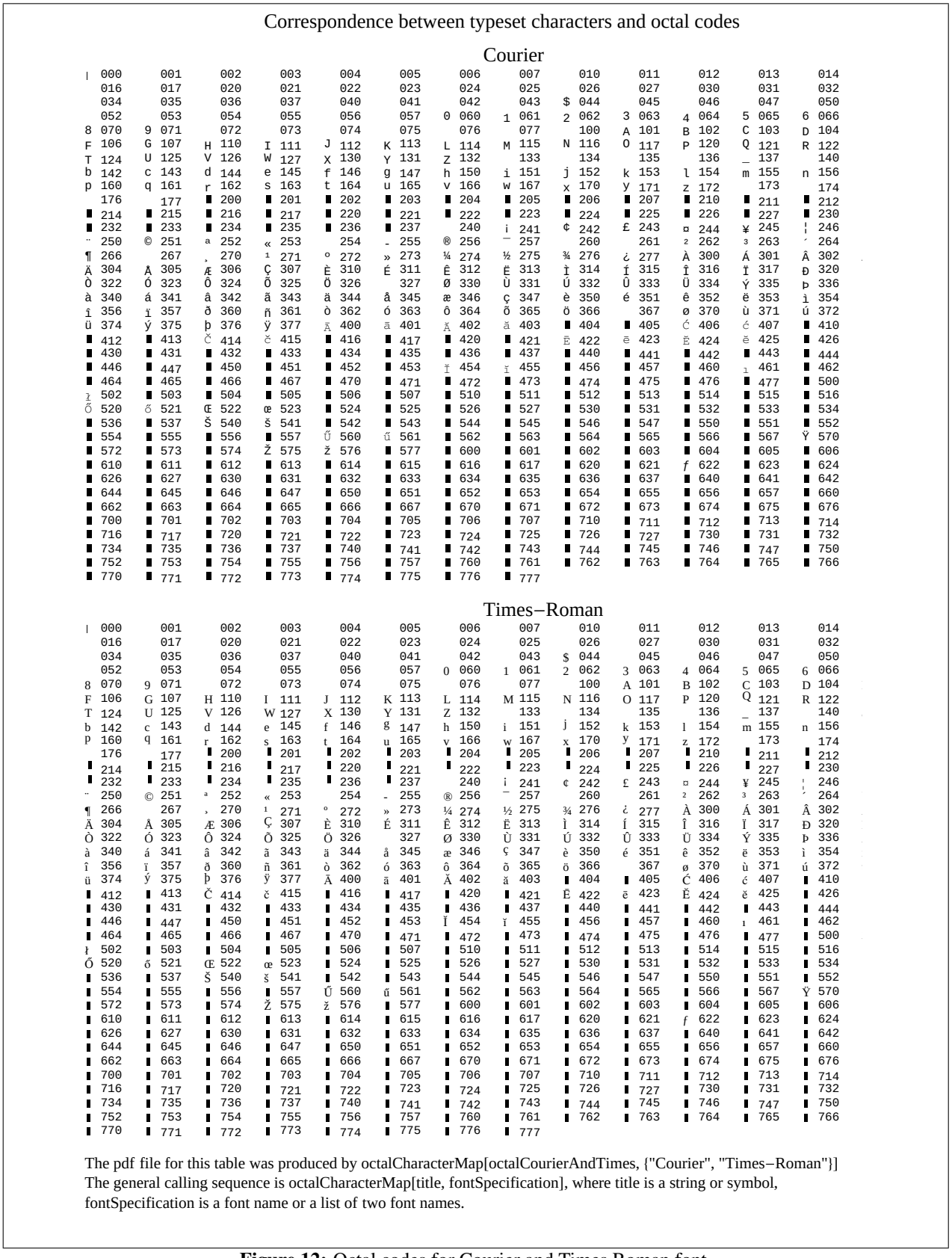

Figure 12: Octal codes for Courier and Times Roman font.

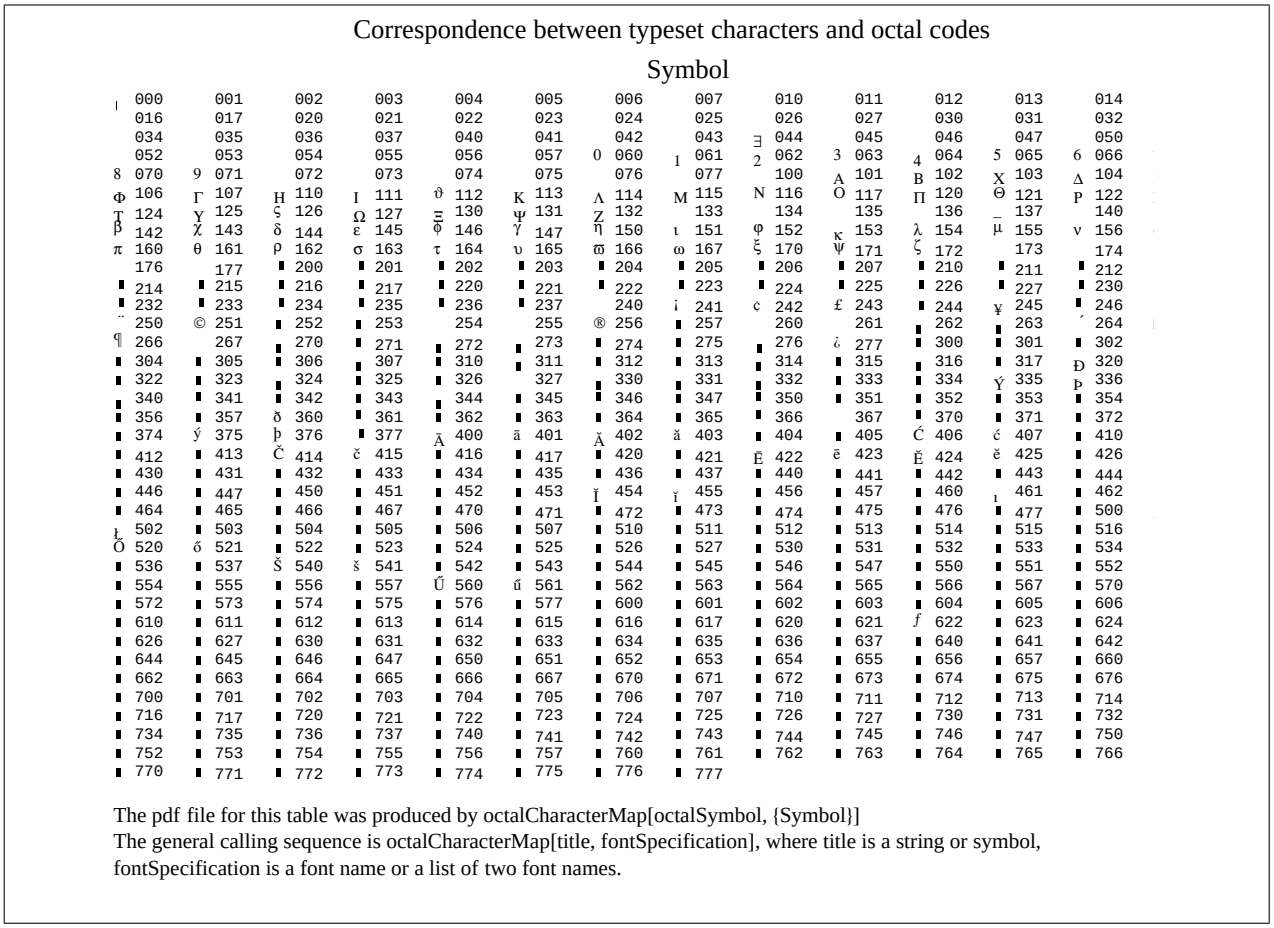

Figure 13: Octal codes for Symbol font.

```
Text[Style["aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"],
      FontFamily->"Times-Roman", FontSize->10, {0,585},{-1,-1}]
```

writes the 1^{st} line in the body of the display. The rest of the body is written by the corresponding statements containing `b`, `c`, ... The vertical grid lines are spaced at intervals corresponding to 0.5 points in the widths of individual 10 point characters. Thus, b, d, g, h, k, n, o, p, q are 5 points wide, and m is approximately 7.75 points wide. The several parts of the display are produced by about 10 simple TMG functions that `fontWidthDisplay` invokes.

Many characters have integer or half odd integer widths. Fig. 18 shows how other widths are fine tuned. The statement `refineWidth["Times-Roman","!",3.25]` wrote the PDF file for this figure. In general, `refineWidth[`$font, character, w$`]` writes the file `widthRefine_`id`.pdf`, where id is the actual character when it is a letter, and its ASCII code otherwise. Fig. 18 displays the string "!!" in 10 points Times-Roman 30 times. A short vertical line is drawn through each of these strings, to mark the end of a string of 40 characters with widths w, $w + .01$, ..., $w + .29$. The values of w are displayed at the left. The width is read off the display, accordingly, as 3.32. I noted the character widths by inspection, and tabulated these manually in `widthsTable`, that begins

```
characterWidth["Courier", v_String] /; StringLength[v] == 1 := 6

characterWidth["TimesRoman", v_String] /; StringLength[v] == 1 :=
 Switch[v,
  "i" | "j" | "l" | "t", 2.78,
  "f" | "r", 3.32,
  "s", 3.89,
  "a" | "c" | "e" | "z", 4.44,
  Alternatives @@ Characters["bdghknopquvxy"], 5,
  "w", 7.23,
  "m", 7.78,
  "W", 9.44,
  "M", 8.89,
  "A" | "N", 7.21,
...
```

The file `tmg.extra` loads this file unless the assignment `readWidths=False` is executed prior to `<<tmg.extra`. Fig. 19 shows the start of the visual check of the measured widths for Times-Roman. The 1^{st} row of letters was set from the string `"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"` in a single `Text` statement. The 2^{nd} row was set from a list of 40 `Text` statements, that each contains the 1-character string `"a"`, with x coordinates increasing by the width that I had recorded in `widthsTable`. Figs. 20– 22 show the corresponding foot of the lower case panel and the top and the foot of the upper case panel. The PDF file for the 1^{st} panel was produced by `letterWidthCheck["Times-Roman", lower]`. In general, the two arguments of this function are the font and the case. There are too many keyboard characters that are not letters to fit on a single page in the style of Figs. 14 and 17. In each of the pairs "()", "[]", "<>", ",." and ":;", the two characters have the same width. The function `nonLetterWidthCheck[`$font$`]` writes the PDF file that checks 40 copies of all the other characters and 20 copies of these pairs. Hence, for example, Fig. 23. The coding of the function is elementary.

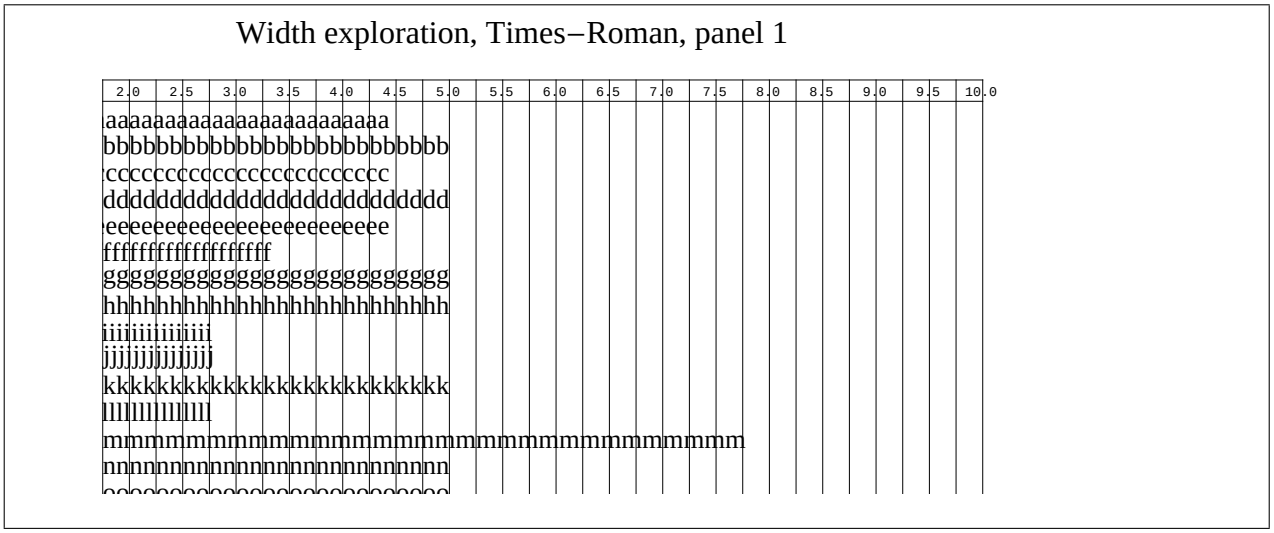

Figure 14: Determining the widths of Times-Roman characters, top of panel 1.

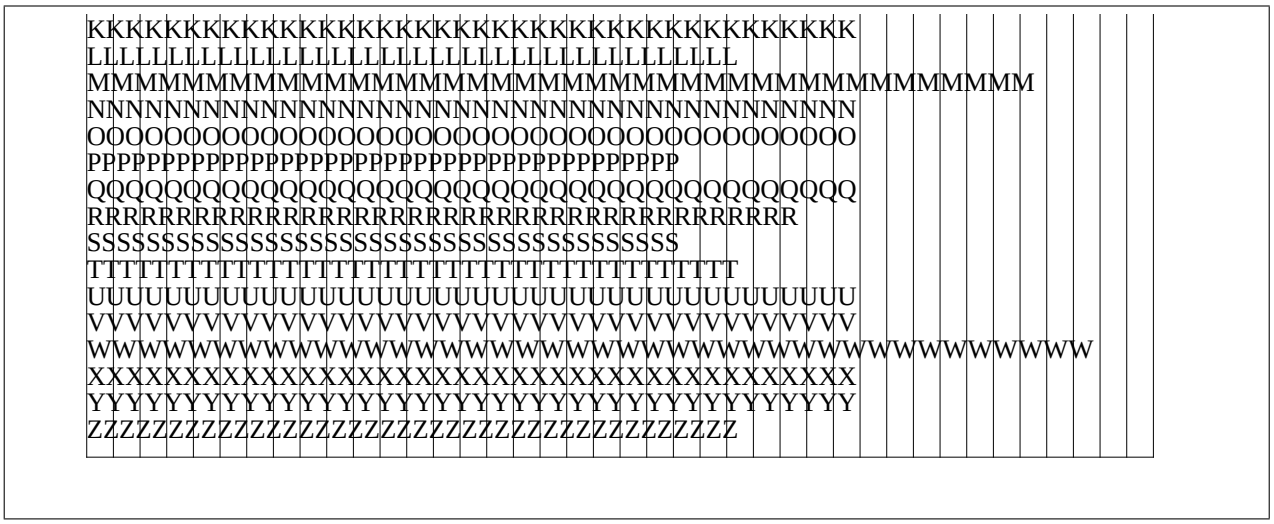

Figure 15: Determining the widths of Times-Roman characters, foot of panel 1.

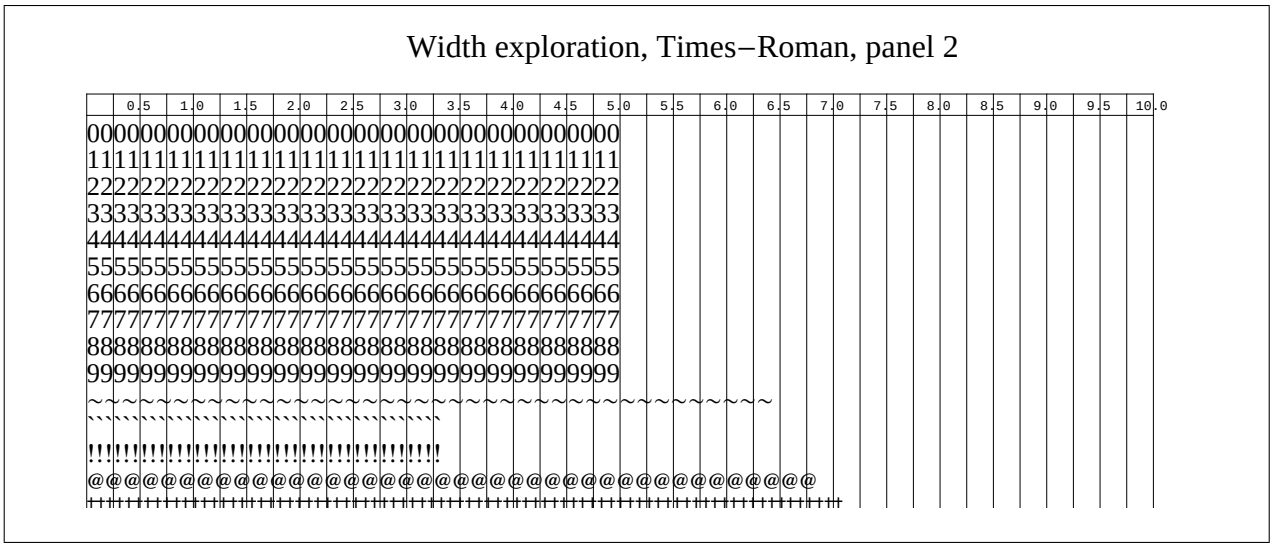

Figure 16: Determining the widths of Times-Roman characters, top of panel 2.

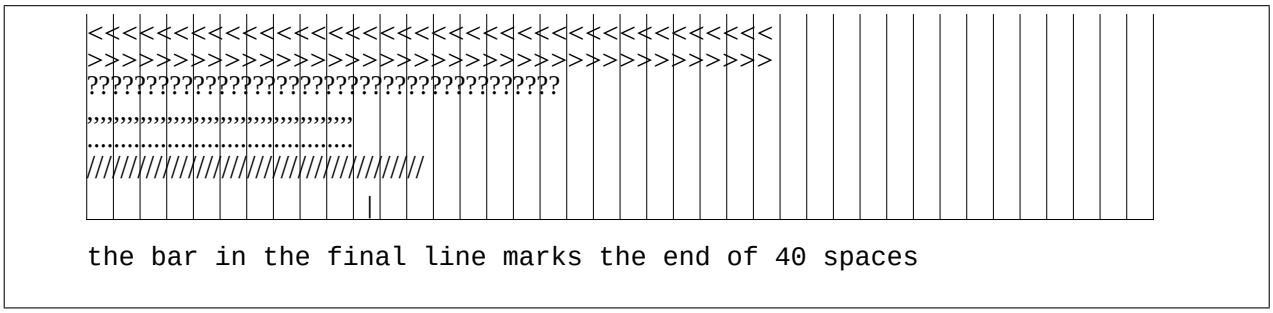

Figure 17: Determining the widths of Times-Roman characters, foot of panel 2.

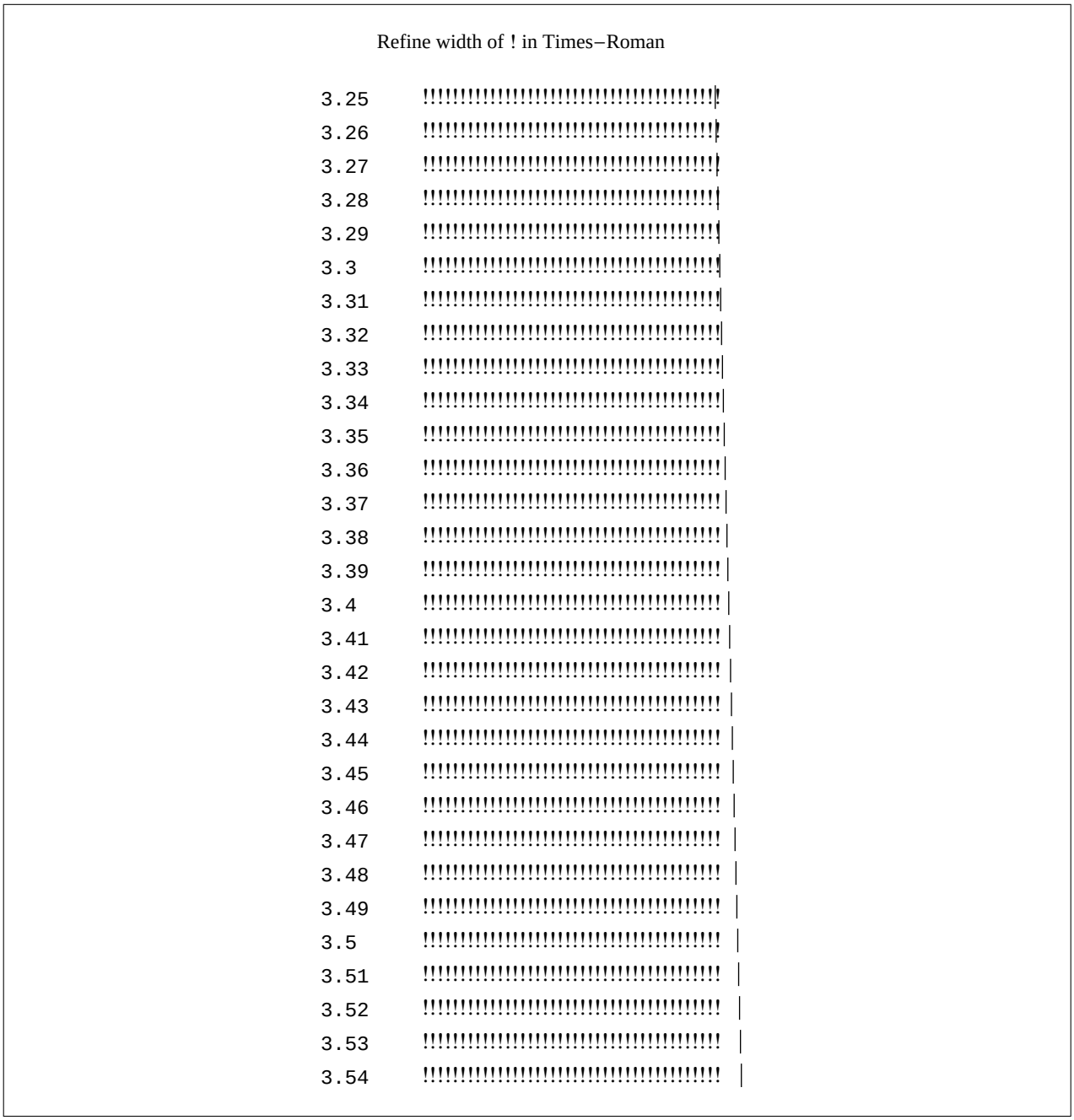

Figure 18: Refining the measured width of a character.

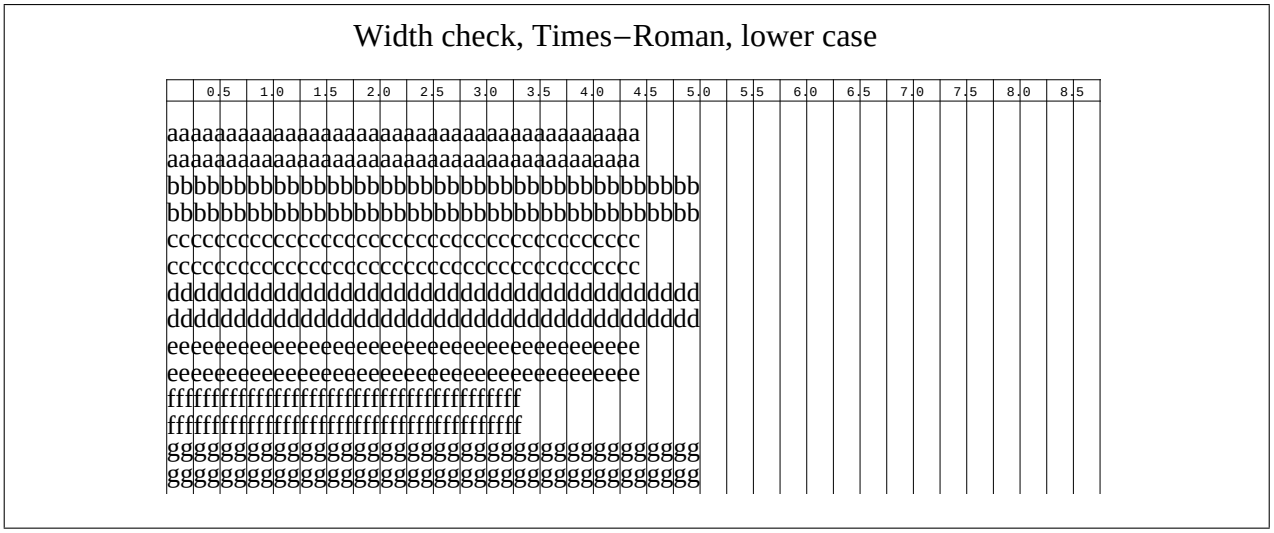

Figure 19: Checking the measured widths of Times-Roman lower case — top of panel.

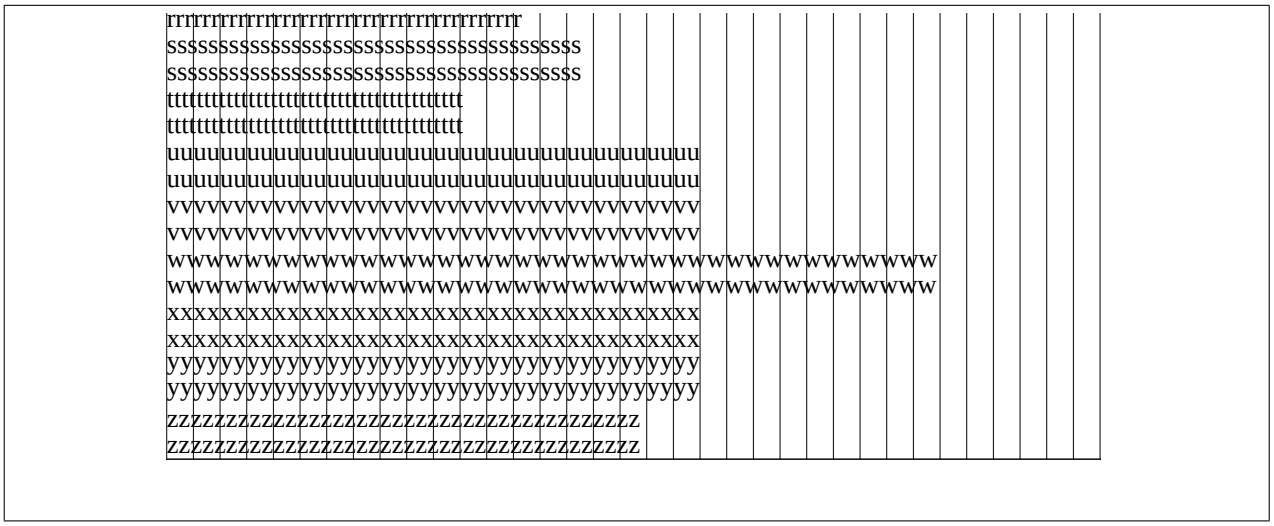

Figure 20: Checking the measured widths of Times-Roman lower case — foot of panel.

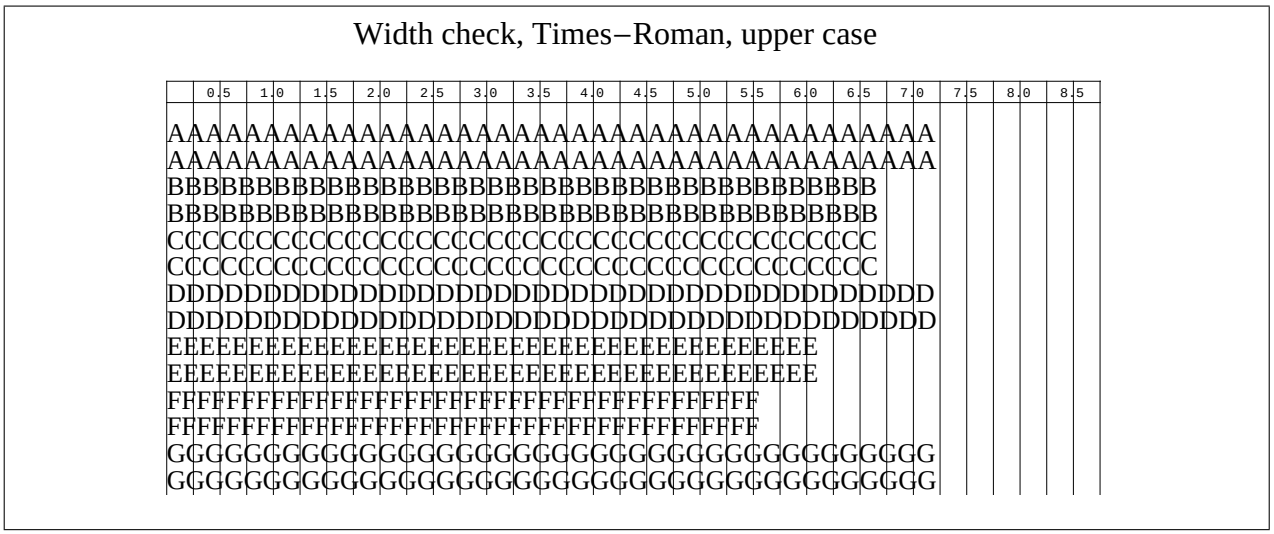

Figure 21: Checking the measured widths of Times-Roman upper case — top of panel.

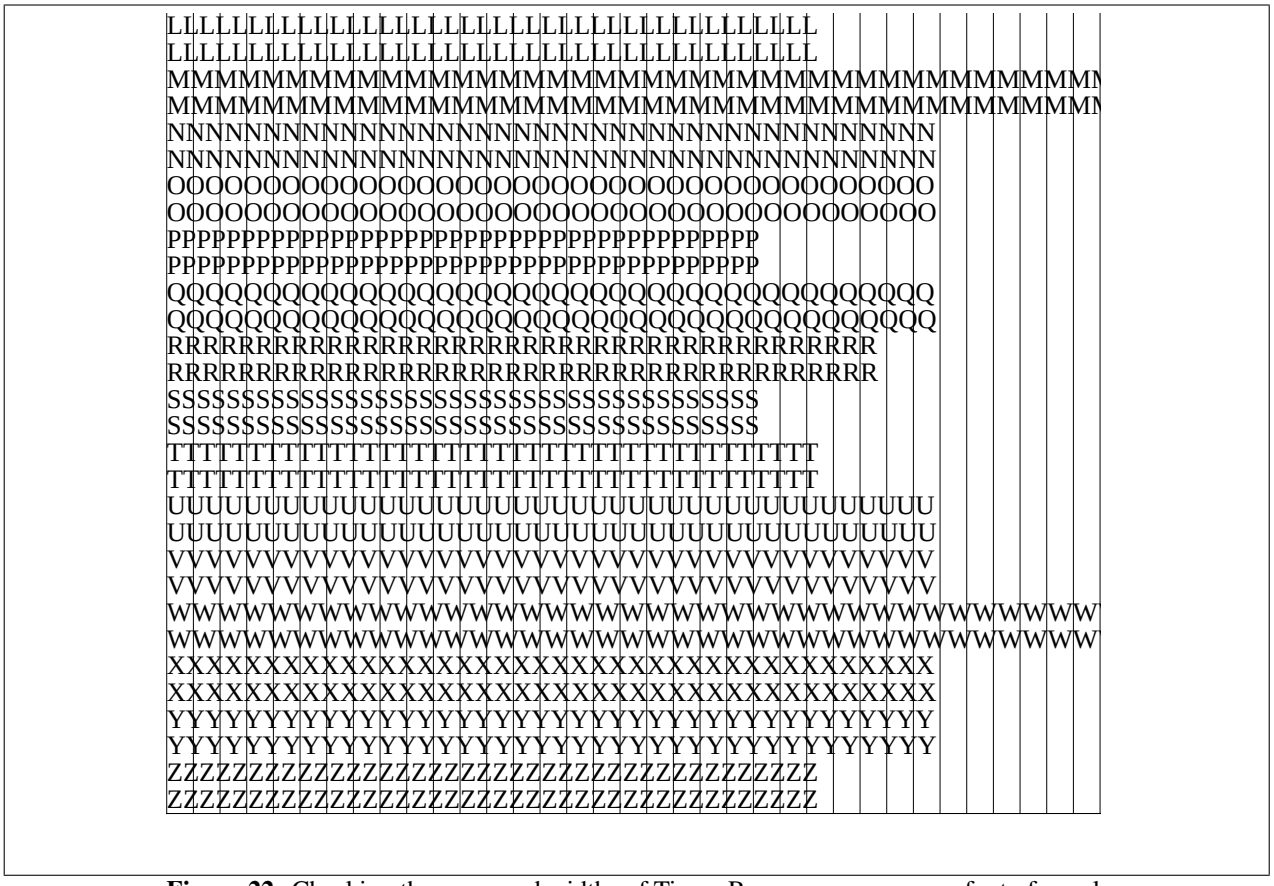

Figure 22: Checking the measured widths of Times-Roman upper case — foot of panel.

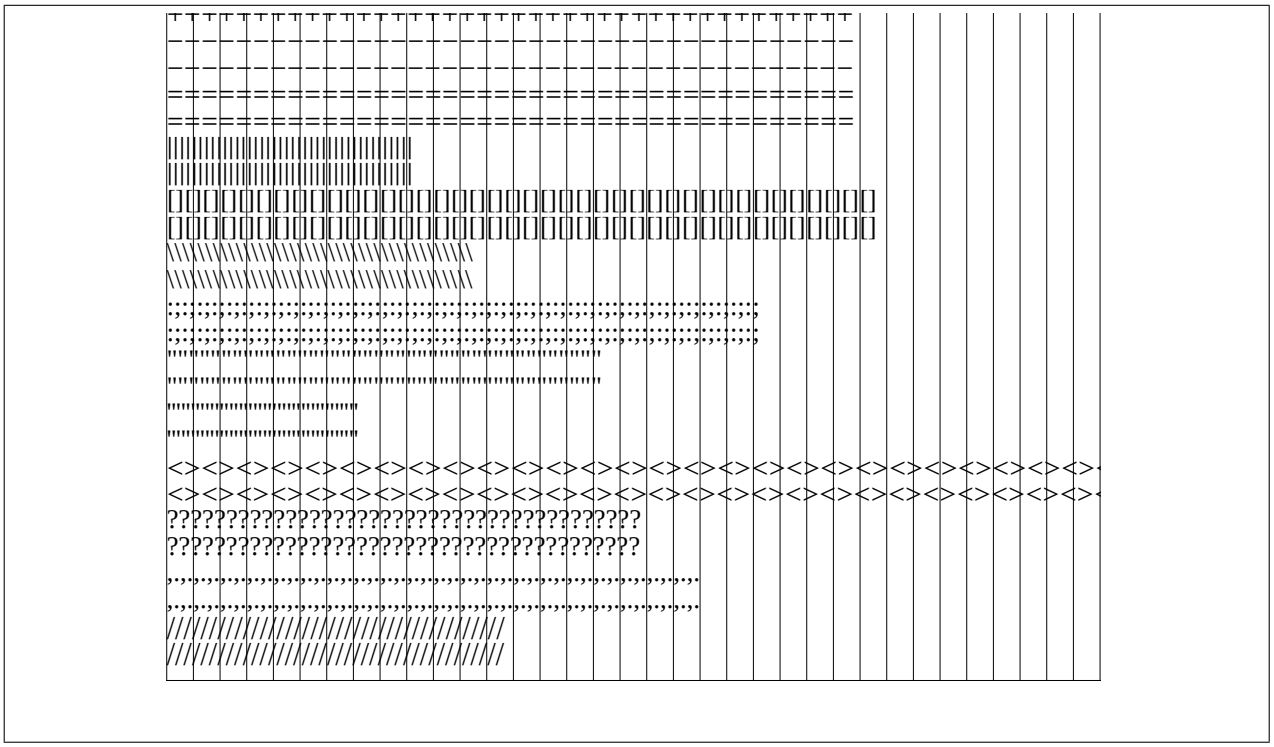

Figure 23: Checking the measured widths of Times-Roman characters that are not letters — foot of panel.

6 Mathematica "Line"s and vertical alignment

I define the y coordinate of a piece of text as the y coordinate of the base of the serif of an "A" that is in the text, or which would look natural if inserted in the text. I determine the precise vertical position of a piece of text, in experimental analyses of the effects of the offset parameter, by drawing a `Line` object in its immediate neighbourhood. The expression `Line[{{`x_1, y`},{`x_2, y`}}]` actually draws a horizontal rectangle that is centered vertically about its nominal y coordinate. The thickness of the line is set to w points by `AbsoluteThickness[`w`]`. Fig. 24 shows that the x coordinates of the left and right ends of the rectangle are $x_1 - w/2$ and $x_2 + w/2$.

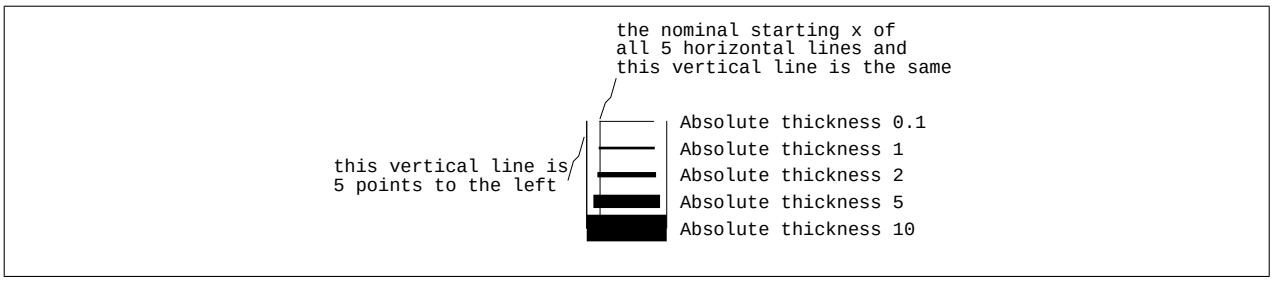

Figure 24: Thickness and ends of MATHEMATICA lines.

The following statements produced the PDF file for Fig. 24.

```
x0 = 200; y0 = 600; length = 20;

lineStart =
{
(* 1st line, thickness 0.1 *)
AbsoluteThickness[0.1], Line[{{x0,y0}, {x0+length,y0}}],
(*      caption            *)
  Text[Style["Absolute thickness 0.1", FontFamily->"Courier", FontSize ->7],
  {x0+length+10, y0}, {-1, 0}],
(* 2nd line, thickness 1 *)
AbsoluteThickness[1], Line[{{x0,y0-10}, {x0+length,y0-10}}],
(*      caption            *)
   Text[Style["Absolute thickness 1", FontFamily->"Courier", FontSize ->7],
  {x0+length+10, y0-10}, {-1, 0}],
(* 3rd line, thickness 2 *)
AbsoluteThickness[2], Line[{{x0,y0-20}, {x0+length,y0-20}}],
(*      caption            *)
   Text[Style["Absolute thickness 2", FontFamily->"Courier", FontSize ->7],
  {x0+length+10, y0-20}, {-1, 0}],
(* 4th line, thickness 5 *)
AbsoluteThickness[5], Line[{{x0,y0-30}, {x0+length,y0-30}}],
(*      caption            *)
   Text[Style["Absolute thickness 5", FontFamily->"Courier", FontSize ->7],
  {x0+length+10, y0-30}, {-1, 0}],
(* 5th line, thickness 10 *)
AbsoluteThickness[10], Line[{{x0,y0-40}, {x0+length,y0-40}}],
(*      caption            *)
   Text[Style["Absolute thickness 10", FontFamily->"Courier", FontSize ->7],
  {x0+length+10, y0-40}, {-1, 0}],
AbsoluteThickness[0.1],
(*      upper "squiggle"   *)
Line[{{x0,y0}, {x0,y0-40}}],
Line[{{x0-5,y0}, {x0-5, y0-40}}],
Line[{{x0,y0+1}, {x0+2,y0+7}, {x0+4,y0+9}, {x0+6, y0+16}}],
(*   upper annotation      *)
Text[Style["the nominal starting x of",
```

```
  FontFamily -> "Courier", FontSize -> 7], {x0+6, y0+29}, {-1, -1}],
Text[Style["all 5 horizontal lines and",
  FontFamily -> "Courier", FontSize -> 7], {x0+6, y0+24}, {-1, -1}],
Text[Style["this vertical line is the same",
  FontFamily -> "Courier", FontSize -> 7], {x0+6, y0+17}, {-1, -1}],
(*    lower left annotation  *)
Text[Style["this vertical line is",
  FontFamily -> "Courier", FontSize -> 7], {x0-100, y0-20},{-1, -1}],
Text[Style["5 points to the left",
  FontFamily -> "Courier", FontSize -> 7], {x0-100, y0-29},{-1, -1}],
(*   lower "squiggle"        *)
Line[{{x0-12,y0-22}, {x0-10,y0-15}, {x0-08, y0-13}, {x0-06, y0-6}}],
(* vertical lines at ends of horizontal line with thickness 10 *)
Line[{{x0-5, y0-40}, {x0-5, y0}}],
Line[{{x0+length+5, y0-40}, {x0+length+5, y0}}]}

export[lineStart]
```

Fig. 25 shows the vertical centering of the `Line` objects. To the unaided eye, and under magnification, the horizontal medians of the lines of decreasing thickness, with the same y coordinates, look collinear with each other and with the centers of the three `Point` objects with the same y coordinate. The diagonals of a square defined symmetrically about this y value intersect on a collinear line segment. The PDF file for Fig. 25 was produced by the script that follows. The figure and script show that the`AbsoluteThickness` and `AbsolutePointSize` arguments are treated as 0.1 when smaller values are specified. The following statements drew the body of the diagram. The further statements

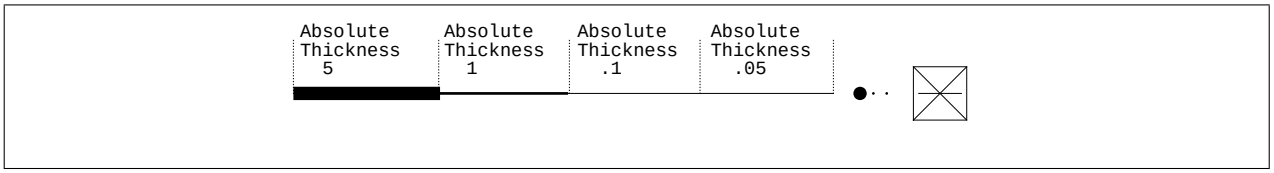

Figure 25: Vertical centering of MATHEMATICA lines.

that provided the annotations are explained in `tmgExamples.auto`.

```
x0 = 10; y0 = 500; length = 50; x1 := x0 + 4 length + 30;

lines :=
{(* Line object nominally 5 points thick: *)
 AbsoluteThickness[5], Line[{{x0, y0}, {x0 + length, y0}}],
 (* Line object with same y value nominally 1 point thick *)
 AbsoluteThickness[1],
  Line[{{x0 + length, y0}, {x0 + 2 length, y0}}],
 (* Line object with same y value nominally .1 points thick *)
 AbsoluteThickness[.1],
  Line[{{x0 + 2 length, y0}, {x0 + 3 length, y0}}],
 (* Line object with same y value that nominally .05 points thick *)
 AbsoluteThickness[.05],
  Line[{{x0 + 3 length, y0}, {x0 + 4 length, y0}}]}

points :=
{(* Point object, with 5 point diameter and center at same y *)
 AbsolutePointSize[5], Point[{x0 + 4 length + 10, y0}],
 (* Point object, with 1 point diameter and center at same y *)
 AbsolutePointSize[1], Point[{x0 + 4 length + 15, y0}],
 (* Point object, with .1 point diameter and center at same y *)
 AbsolutePointSize[.1], Point[{x0 + 4 length+20, y0}] }

intersection :=
{(* rectangle with median at same y value *)
```

```
 Line[{{x1, y0-10}, {x1+20, y0-10},
       {x1+20, y0+10}, {x1, y0+10}, {x1, y0-10}}],
 (* diagonals of this rectangle *)
 Line[{{x1, y0+10}, {x1+20, y0-10}}],
 Line[{{x1, y0-10}, {x1+20, y0+10 }}],
 (* Line object with same y as earlier Line objects *)
 Line[{{x1+2, y0}, {x1+18, y0}}]}
```

It is easy to relate the base of the serif of an "A" to a `Line` object when a large font is used. Fig. 26 shows an "A" in 500 point Times-Roman font, with "lines" drawn through the base and top of the serifs. Even though these "lines" are rectangles with finite thickness, this is small enough in relation to the serifs to preserve their individuality when the graphics object is halved in size.

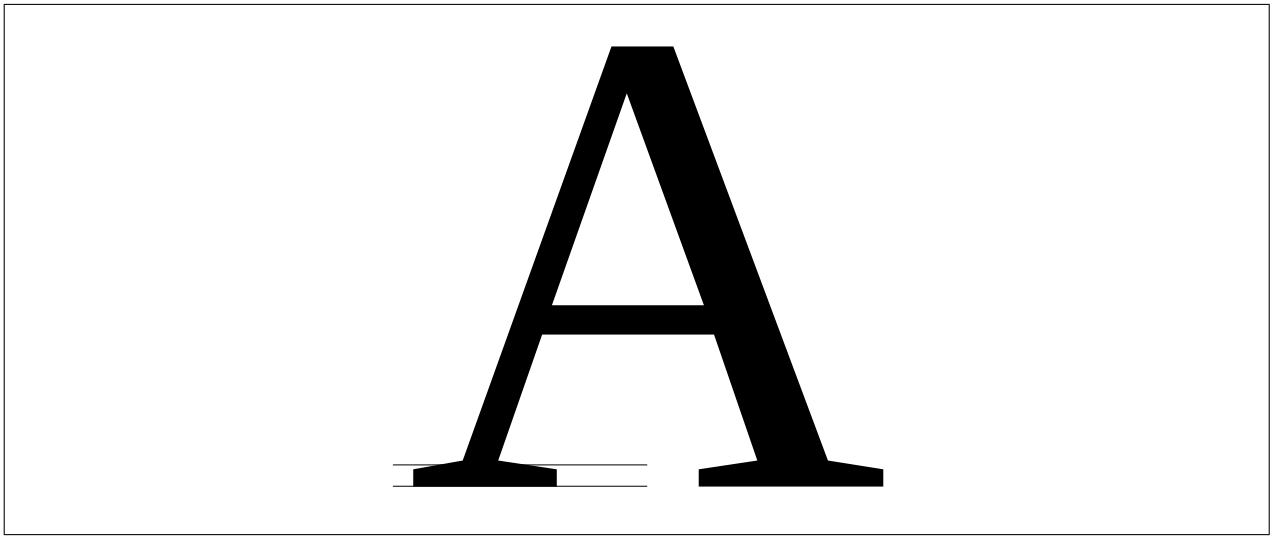

Figure 26: Boundaries of a serif.

7 Measuring offsets

7.1 Base offsets

Fig. 27 shows how I determine the offset for a string that includes the full variety of ascenders and descenders, that puts their common baseline at a specified y coordinate (see *e.g.* [3]). The alphabet and digits in the 1^{st} line of the body, and the horizontal rule beneath these, were output by the two statements

```
Text[ Style[
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789",
   FontFamily -> "Courier", FontSize -> 8], {40, 587}, {-1, -0.8}]

Line[{{40, 587}, {480, 587}}]
```

The further lines show the vertical displacement for the successive offsets in the specified range. The file for Fig. 27 was written by the statement

```
varyBaseOffsets[{{Courier, 8, letters <> digits, -0.8, -0.0, .05}}, "Courier_8"]
```

Here, the file name is `varyBaseOffsets_Courier_8_page_01.pdf`. The 1^{st} argument of `varyBaseOffsets` is a list of just one 7-item specification. In general, it can consist of any number of these. This usually leads to separate PDF files for several pages of output, named `varyBaseOffsets_`tag`_page_`n`.pdf`, where `"`tag`"` is the 2^{nd} argument of `varyBaseOffset`. The structure of this function allows for the possible continuation of the panel for a particular font style and size across a page break. I ran this function using the list of combinations of Courier, Times-Roman and Symbol fonts in sizes 4–20, and offsets -.6 to -.3 by intervals of 0.2. This gave 34 pages of output, as separate files. The statement `pdfFileConsolidation[]` combined all the pages produced by `varyBaseOffsets`,

that were in the working directory, into a single LaTeX document. Inspection narrowed the ranges that spanned the values which aligned the baselines with the reference lines. Using these in `varyBaseOffsets` produced 22 pages. Consolidating these produced the file `varyBaseOffsetsShortened.pdf` that accompanies the present explanation of TMG in the distribution material. The following statement produced it, by overriding the defaults for the successive arguments. These are the root of the output PDF file name, a collective name covering the files to be collected, and exclusions from what it covers.

```
pdfFileConsolidation[varyBaseOffsetsShortened, varyBaseOffsets,
 {"varyBaseOffsets_Courier_8_page_01"}]
```

Figure 27: Effect of varying the base offset.

Inspection gives the following base offsets, that can be varied by $\pm.02$ in many cases without destroying the required horizontal alignment. I cannot find any pattern in the mean values or the tolerance. Even worse, deciding which offsets are best is subjective, and I am concerned that, to some extent, the processes described in this section and the next may measure artefacts. Some of the effects that I have seen on the screen are artificial. These include the relative placement of horizontal lines and text, particularly when scaling is changed.

point size:	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Courier	-.46	-.51	-.57	-.43	-.42	-.49	-.50	-.57	-.56	-.51	-.51	-.53	-.53	-.51	-.59	-.59	-.55
Times	-.47	-.32	-.41	-.42	-.49	-.53	-.45	-.45	-.49	-.48	-.43	-.47	-.48	-.48	-.50	-.43	-.46
Symbol	-.57	-.34	-.43	-.50	-.53	-.40	-.46	-.50	-.53	-.49	-.49	-.49	-.51	-.56	-.48	-.46	-.51

Table 1. Base offsets.

7.2 Individual character offsets

Fig. 28 shows how I find offsets for individual characters to give baseline alignment. At the top of the figure, the string "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" is displayed by a single `Text` statement. This contains the offset that makes the common baseline of the characters coincide with the horizontal line drawn by a `Line` expression with the same y coordinate as item 1. The next row superposes

1. the string drawn by a `Text` statement that differs from that just mentioned in just the y coordinate, and
2. the output of 62 separate `Text` expressions, that each draws a single character, with the same y coordinate, and the y offset -0.76 shown at the left of the row.

The further rows of Fig. 28 provide the corresponding superpositions with offsets increasing from -0.745 to -0.26 by 0.015. The PDF file `varyIndividualOffsets_Courier08_page_01.pdf` was written by

```
varyIndividualOffsets[
 {Courier, 8, letters <> digits, -0.42, -.76, -0.26, 0.015}, "Courier08"]
```

I wanted to find the offset for each character that gave the best superposition. The general form of the statement that wrote the PDF file for Fig. 28 is `varyIndividualOffsets[`$font, size, string, baseoffset, \sigma_<, \sigma_>, \sigma_\Delta$`]` where $string$ specifies the characters to be explored, and the test offsets range from $\sigma_<$ to $\sigma_>$ by σ_Δ. I used this function extensively, and drafted tables of the requisite individual offsets that I found by inspection. Unfortunately, besides being very tedious and frustrating, this work is subjective. The required offsets can be specified within a range of 0.02 for some characters. But the effects are indistinguishable over a range of 0.5 for other characters. A concern that artefacts, or errors in programming, play a role in the displays overshadows these observations,

I consider the lighter the character the closer the superposition. For most lower case letters, this is met by offsets -0.655 to -0.565. Often, it is also met by alternating offsets -0.555 to -0.535. This is disconcerting. I took the median values that looked acceptable, and then refined those by the steps described below.

The first runs of `varyIndividualOffsets` spanned ranges that were excessive. The output showed the narrower ranges that needed spanning. The statements that gave the next round of output are in the accompanying file `tmgExamples.auto`. This output is consolidated in `exploreOffsets.pdf`. I typed a separate statement for each combination of font face and size that I found optimal by inspection. For Courier 8, this was

```
makeAllPairs[Courier, 8,
{-.64, abcdefhiklmnorstuvwxzABCDEFGHIJKLMNOPRSTUVWXYZ0123456789, -.45, j, -.30, gpqyQ}]
```

This assigned `allPairs["Courier", 8]` to the list

```
{{"0",-0.64}, {"1",-0.64}, ..., {"a",-0.64}, {"A", -0.64}, ...,
{"F", -0.64}, {"g", -0.3}, {"G", -0.64}, ..., {"I",-0.64},
{"j",-0.45}, {"J",-0.64}, ..., {"z",-0.64}, {"Z",-0.64}}
```

Here, the offset -0.64 is paired with each individual character in the string in the `makeAllPairs` statement with which it is associated. The offset -0.45 is paired with j, and -.30 with g, p, q, y and Q. In general,

`makeAllPairs[`$face, size$`,{`σ_1`,{"`$c_{1,1}c_{1,2}\ldots c_{1,k_1}$`"},..., {`$\sigma_\ell$`,{"`$c_{\ell,1}c_{\ell,2}\ldots c_{\ell,k_\ell}$`"}}]`

constructs the string representation of the following `Set` statement, and then uses `ToExpression` to convert this to the actual assignment.

`allPairs[`$face, size$`] = {{"`$c_{1,1}$`",`σ_1`},{"`$c_{1,2}$`",`σ_1`},...,{"`c_{1,k_1}`",`σ_1`}, {"`$c_{2,1}$`",`σ_2`},...,{"`c_{ℓ,k_ℓ}`",`σ_{k_ℓ}`}}`

Figs. 29 and 30 show how the offsets were tested and adjusted. The 1^{st} statement displays the test string, character by character, using the tentative offsets, above the same string set by a single `Text` statement. The 2^{nd} statement alters the specified offsets by the given amounts. Repeating the 1^{st} statement (with `C9c` in place of `C9b`) produced Fig. 30. In general `offsetCompare[`f, s, t, r`]` writes the file `offsetCompare_`r`.pdf` that displays the string t set in the two ways in font f, size s. The statement `offsetAdjust[`f, s`,{{`c_1, δ_1`},...{`c_k, δ_k`}}]` alters the offsets of $c_1, \ldots, c_k$ in font f, size s, by $\delta_1, \ldots, \delta_k$, respectively.

```
offsetCompare[Courier, 9,
     "the quick brown fox jumped over the lazy dog", "C9b"]

offsetAdjust[Courier, 9, {{g, -.10},
  {j, -.12}, {l, -.05}, {p, -.10}, {q, -.05}, {y, -.1}}]
```

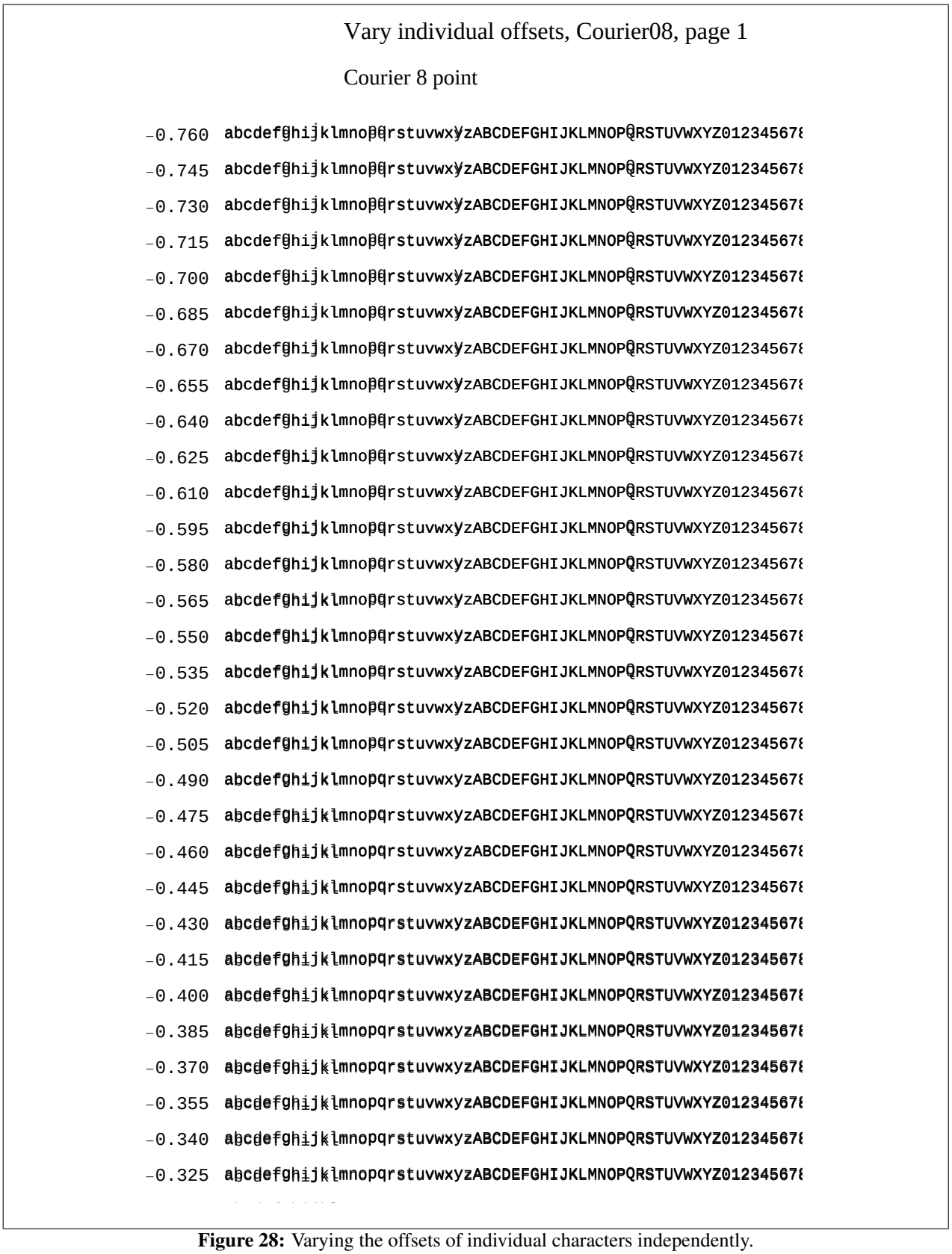

Figure 28: Varying the offsets of individual characters independently.

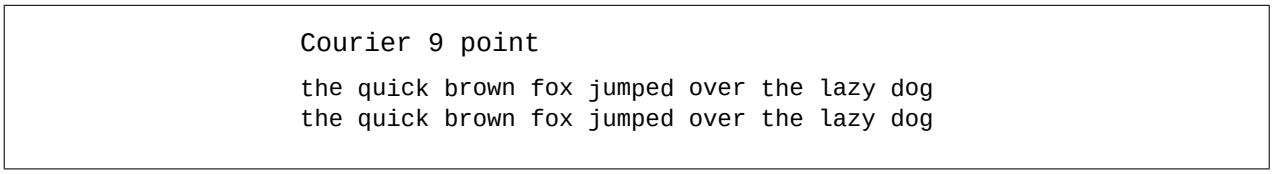

Figure 29: Before adjustment of offsets.

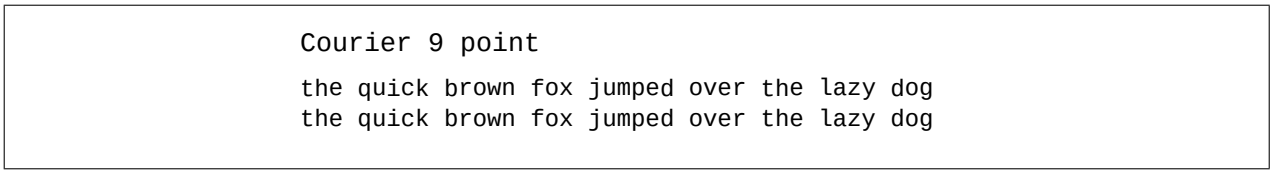

Figure 30: After adjustment of offsets.

The output of `offsetAdjust` was written by the function `exportBounded`, described in §3 to avoid excessive white space when a set of PDF files that it produces are collected by `pdfFileConsolidation`. The statement `saveAllPairs` writes the set of `allPairs` assignments, that are currently in force, as the file `allPairsTable`. As a preliminary, the previous version of the file is preserved as `allPairsTableX`n, where n is an automatically created backup number. The most recent version is read during the loading of `tmg.extra` unless `loadAllPairs` was pre-assigned the value `False`.

I wrote several more functions that produced displays which helped adjust offsets and explore their effects, but I do not think that these helped materially. They are available on request, but including descriptions here would be excessive. And, in any case, my object in writing [1] and this supplement is to elicit a solution that removes the need for the work on offset measurement that these describe.

8 The encode function

The usage of `encodeString[`$font, size, x_0, y_0, string$`]` and `encodeSequence[`$font, size, x_0, y_0, \{items\}$`]` is described fully in the TUGboat paper that this document supplements. These functions return expressions that occur 1^{st} in the pairs returned, respectively, by `encodeAndMeasureString[`$font, size, x_0, y_0, string$`]` and `encodeAndMeasureSequence[`$font, size, x_0, y_0, \{items\}$`]` . The 2^{nd} item returned by these functions is, respectively, the width of the string, and the final values of the output coordinates (x, y). The internal operation of the functions is elementary. The function `encode[`$string$`]` abbreviates `encode[Courier,10,0,600,`$string$`]`.

References

[1] M.P. Barnett, Aligning text in diagrams exported by MATHEMATICA: a question about the POSTSCRIPT infrastructure. TUGboat, in press for 31 (3) 2010; `http://tug.org/TUGboat/production/31-3/tb99barnett.pdf`.

[2] PostScript Language Reference, Adobe Systems Incorporated, `http://www.adobe.com/products/postscript/pdfs/PLRM.pdf`.

[3] Baseline (typography), `http://en.wikipedia.org/wiki/Baseline_(typography)`.