How to Render Mathematical Symbols in Java

Yuemei Sun

Master of Science Degree Project

Department of Computer Science

Oregon State University

Corvallis, Oregon

U.S.A

Committee Member:

Dr. Jonathan Herlocker

Dr. Bella Bose

Dr. Eric Mortensen

March 2003

Acknowledgements

I would like to thank my major advisor Dr. Jonathan Herlocker for his guidance and encouragement. I greatly appreciate my committee members, Dr. Bella Bose and Dr. Eric Mortensen, for their time and advice on my project report. I would also like to acknowledge Anton N. Dragunov, a PhD student in the Department of Computer Science at Oregon State University, who improved and integrated my code into our bigger project. I am very grateful to my husband Kaichang Li for his love and wholehearted support. I thank my family and friends in China for their love and encouragement. I would like to thank all the people that gave me support during my study at OSU.

Table of Contents

Abstract

Current math software provides limited interactivity and dynamics when conveying math ideas. We are developing prototype Java toolkits to display math-embedded documents in a more dynamic and interactive way. The core task of current project is to create a Java implementation of a MathML (in which a math expression is encoded) rendering engine, which can render a math expression in such a way that not only the whole expression but also every term in it is accessible from user as well as from the parts of the document that include it. The focus of this paper is on how to render each single math symbol, be it “normal” or “stretchy”.

Keywords

Java, Unicode, Encoding, Font, Glyph, Typography, Stretchy symbol, MathML

Introduction

How many times do you have to turn the pages of a journal or your textbook back and forth, trying to follow the derivations of a complex math formula, which always span several pages, and lose track of where you are? How many times have you been bored when an article or a textbook goes on and on explaining a formula or a math symbol you've already known? Or how many times have you got lost when you try to understand a formula that contains some symbols or formulas you don't know but the author assumes that you do?

Presenting a math formula or expression in such a static and non-interactive way makes the process of understanding it frustrating and time-consuming. The wide use of digital documents as a way of exchanging information makes it even urgent to address these problems. Many efforts have been made to communicate math ideas dynamically and interactively in digital format.

The W3C has defined MathML (Mathematical Markup Language), a subset of XML, as a recommended way to encode mathematical expressions so that they can be embedded in web pages[1]. MathML provides some mechanisms for dynamic and interactive display of math expressions. For example, MathML allows you to add links to its sub-expressions, and to have certain interactivity for them by passing them as arguments to an element called <maction>, which will then grant interactivities according to the given attributes. The attributes include "toggle", "statusline", tooltip", "highlight", "menu", the names of which speak for themselves what they mean[2]. Many implementations of MathML have emerged. Netscape, Mozilla, and Amaya have incorporated functionalities that can display MathML directly; some add-on software

packages have been built to enable MathML display in popular browsers, for example, MathPlayer for Microsoft's IE. Math equation editors and authoring tools also came out, among which were WebEQ and MathType[3].

Although some of these software do provide a certain level of interactivity and certain amount of dynamics—for example, WebEQ can expand or shrink an expression when mouse is over certain part of it (e.g., eclipse and the expression it represents)[4] — they don't provide convenience for navigating through derivations, nor do they provide any possibility to animate each term in an arbitrary way[2].

More specifically, current mathematical software didn't solve the following problems as summarized by Anton Dragunov, a Ph.D student and the major contributor of this project: 1) "inability to view, compare, and contrast related mathematical artifacts" and their textual explanation "within a single field of vision due to limitations of print/display/ medium", 2) inappropriate amount of detail in mathematical explanations for all potential readers, 3) inconsistency in the use of notations among authors and documents,[2] and 4) as far as scholarly scientific communications is concerned, the inability to add internal and external links to math expressions embedded in PDF files, or to resize the math expressions included as an image in html files as the font size of the surrounding text changes.[5]

In an effort to solve these problems, we are developing a prototype Java toolkit to display math expression embedded documents in a more interactive way. Our software will enable a math expression, as well as every term in it, to interact with the user and the parts of the document that include it. In this way, users can not only see everything, including the derivations of a formula and the explanation of symbols, in one page, but

also arbitrarily animate each term as needed, such as repositioning each term, enlarging a symbol, and so forth. Our software can be used to better communicate math in journal articles (PDF file), power point representation, elementary education, and web courses. Our software can also serve as an authoring tool, with which scientists can more easily explain their complex formulas.

A journey of thousand miles starts with a step. Before we can implement more complex functionalities, we must first be able to display a math expression and make each term in it separately accessible, which is the core part of current project.

This paper focuses on how to render a mathematical symbol in Java, both normal symbol such as " ∑ " and stretchy symbol like " ∫ ". The topics tackled include:

1. Adding math fonts to Java runtime
2. Additional challenges of rendering math
3. My implementation and contributions
4. Future work

Before discussing these topics, let's first get familiar with the terms used in this paper.

Basic Terminology

Character — Also abstract character. "The smallest component of written language that has semantic value, refers to the abstract meaning or abstract shape" [6], rather than a specific shape, and has no numeric value by itself unless encoded.

Font — A set of characters in any specified design and size[7]. It defines the shape of each character.

Font family	"A group of font series that are reflected in basic design but differ in either weight of the design, from ultra-light to 'black' or ultra-bold, or width, from ultra-condensed to ultra-expanded."[8]
Glyph	Visual representation (shape) of a character. "An image for a character in a particular font and style."[9]
Glyph Index	Also called glyph code. "A numeric code that refers to a glyph in a font. May be local to a particular font; that is, a different font containing the same glyphs may use different codes."[10]
Unicode	A universal standard for representing a broader character set using two-byte encoding for each of them. It's a one-to-one mapping from an integer to an abstract character. It doesn't offer any methods for rendering text on screen or paper. In other words, it's a character encoding, not glyph encoding. Therefore, an abstract character such as "ARABIC LETTER BEH" which has the U+0628 code value can have different visual representations (shapes or glyphs), where "U+" stands for Unicode and "0628" is in hexadecimal format.[11]
Plane	"A range of 65,536 (10000_{16}) contiguous Unicode code points, where the first code point is an integer multiple of 65,636 (10000_{16}). Planes are numbered from 0 to 16, with the number being the first code point of the plane divided by 65,536. Thus Plane 0 is U+0000..U+FFFF, Plane 1 is U+10000..U+1FFFF, ..., and Plane 16 (10_{16}) is U+100000..10FFFF."[12]
BMP	Short for Basic Multilingual Plane, the first 64k characters, an area of the code space of Unicode Version 3.0 that contains codes for 49,194

	characters from the world's alphabets, ideograph sets, and symbol collections (including mathematical symbols); plane-0 of Unicode, representing Unicode characters from U+0000 to U+FFFF eg: U+05D1.
PUA	The Unicode's Private Use Area, representing Unicode characters from U+E000 to U+F8FF, which is exempted from receiving any official assignments and is left to applications for their internal use.
Font Encoding table	Mapping from characters to indices of desired glyphs in a font. Tells what character each glyph represents. Also called "encoding vector"[13].
Normal symbol	A mathematical character whose visual representation is a simple glyph.
Stretchy symbol	A mathematical character whose visual representation can be built by assembling several glyphs from left to right or from top to bottom.

Part I: Choosing a Java Font Approach

As explained above, Unicode is a character encoding, not a glyph encoding. Therefore, although every math symbol has a unique code point in Unicode, that code point itself does not offer any visual representation. We have to have a font or fonts that can draw these math symbols. Unfortunately, the standard fonts that come with all major operating systems can only render Latin 1 (basic Latin), Extended Latin-A, Extended Latin-B, Greek symbols, Cyrillic, general punctuation, currency symbol, and some

mathematical operators, where Latin 1 covers Unicode characters ranging from U+0000 to U+007F, Extended Latin-A, U+0100 to U+017F, Extended Latin-B, U+0180 to U+024F, Greek, U+0370 to U+03FF, and Cyrillic, U+0400 to U+04FF. [14] Thanks to Wolfram Research, Inc., we now have a set of math fonts (**Mathematica 4.1 Fonts**) that can render most of the math symbols we need now.[1]

Now that we have the fonts, how do we let Java runtime know about them? Do we install them locally, or do we call them dynamically at runtime? This depends on our criteria and what Java has to offer. Before going into detail about these topics, let's first familiarize ourselves with Java fonts background.

Java Fonts Background – Physical Fonts and Logical Fonts

There are two types of fonts in Java – physical fonts and logical fonts. Physical fonts are the actual font libraries consisting of TrueType, PostScript Type 1, or OpenType fonts.[2] For example, "Times New Roman" at point size 12 is the physical font used for rendering the text of this paper. Physical fonts are installed in the host operating system's standard location and are on java runtime's font path. Additional fonts can be installed by the operation system, which will place them in the standard location so that Java run time knows where to find them. The Java 2 platform provides APIs that allow

[1] Currently, there is an ongoing STIX (Scientific Technical Information Exchange) Fonts Project, the mission of which is to create "a comprehensive set of fonts that serve the scientific and engineering community in the process from manuscript creation through final publication, both in electronic and print formats." These fonts will have more than 7,700 glyphs, covering all the symbols in MathML. According to the project's website, these fonts will be available sometime in 2003, and will be free to end users. See http://www.stixfonts.org

[2] They are all outline fonts, which are scalable. Actually, OpenType fonts are either TrueType or Type 1, with a wrapper.

an application to find out what fonts are available to a given runtime and whether or not a given character can be rendered by these fonts.

Logical font names are the font-type names recognized by the java runtime. They are not the actual font libraries installed on your local system, but are mapped to your local font libraries. Java platform since version 1.0 defines five logical font names that every implementation must support: Serif, SansSerif, Monospaced, Dialog, and DialogInput. These logical font names are mapped to local font libraries in implementation dependent ways. Typically one logical font name maps to several local fonts in order to cover a large range of characters. An application using peered AWT components[3] can only use logical font names.

Our Criteria for Selecting Font Approaches

We have 4 criteria for selecting a font approach:

1) It must cover all the mathematical symbols that are frequently in use; these including math operators, Greek symbols, and symbols that are able to stretch such as fences—parenthesis, integral, braces, square, square root, etc;
2) It must work for both Java applications and applets. Our current project is a Java application, but eventually we will build a tool that will be put on web for others to use. So it will be an applet eventually;
3) It must be easy to use for the users. Users should do as little as possible in order to use our tool;

[3] Also called "heavy weight component". Every AWT component has a peer class serving as the interface between Java code and native windowing system, therefore, it depends on native (peer) system for its drawing and rendering. In contrast, a "light weight component" can rely solely on Java code for drawing.

4) When the above three are satisfied, easiness in implementation is also a consideration.

Java's Font Approaches in Brief

What choices does a Java application have in selecting fonts? According to the java documentation for JDK 1.4[15], an application using lightweight components[4] can select fonts in four ways. 1) The oldest one is using logical font names, which came into use since JDK 1.1. 2) A more recent approach is using the name of physical fonts installed locally, which is used since JDK 1.2. 3) A third way involves using the physical fonts that come with Java2 Runtime Environments (J2RE). 4) The newest one comes with JDK 1.3, with which you can use bundled fonts at runtime without locally installing them.

Next, let's go over Java's four font approaches in detail, evaluating them against our criteria, starting with the most recent approach.

Excluding the 4th approach—using bundled physical fonts

Since version 1.3, Java Platform provides a method, Font.createFont(int fontFormat, java.io.InputStream fontStream), with which an application can pack and instantiate TrueType fonts. What this method does is to open/create a new temp folder on your system and to dynamically load TrueType fonts into Java Virtual Machine at runtime. In this way, an application can create a TrueType font without locally installing them.

[4] A light weight component is one written entirely in Java, thus drawn and rendered entirely in Java code rather than depending on native component for its rendering. In contrary, an AWT component is heavy weight.

The advantages include achieving the same look everywhere, having full control over which languages to support, and most importantly, the easiness to use, which best satisfies our criteria No.3. This approach also meets our criteria No.1 by packing all the needed fonts at runtime, provided that the URLs for the math fonts in need are known.

The disadvantages lie in the size of the bundled fonts, which can be very big if they support Asian languages like Chinese, Japanese, and Korean, and the licensing issues that might arise. More importantly, it will be very difficult to satisfy our criteria No.2—the ability to work for Java applets.

For an applet, Font.createFont(int, java.io.InputStream) will throw *java.Lang.SecurityException* if the fonts and the applet in need of them are not on the same server. This is because of the one of the sandbox restrictions imposed on a java applet, the "phone home" rule: "the only host that an applet can establish a network connection to is the one from which it was loaded."[16] The math fonts we are using are *Mathematica* fonts created by Wolfram Research, Inc. They can be freely downloaded from their creator's server, but we cannot put them on our server for downloading because of the licensing issues. There might be ways to escape from the sandbox restrictions, for example, signed applet, but it's out of the scope of this article. For further information in this respect, see chapter 9 of Macgregor, Robert, Durbin, Dave, and Owlett, John, and Yeomans, Andrew, *Java Network Security*, Prentice Hall PTR, New Jersey, 1998. You can also refer to http://java.sun.com/docs/tutorial/security1.1/index.html, which teaches how to produce digital signatures for data and how to verify the authentication of such signatures.

Because our tool will eventually be an applet, and because a java applet has restrictions imposed on it, which are difficulty to circumvent, we excluded using bundled physical fonts.

Excluding the 3rd approach—using the Lucida fonts

The Lucida fonts are a set of TrueType fonts that come with the Java 2 Platform itself. Sun's Java 2 Runtime Environments (J2RE) also contains these fonts. The Java 2 SDK provides three font families: Lucida Sans, Lucida Bright, Lucida Sans TypeWriter, with each family containing four fonts (regular, bold, oblique, and bold oblique), for a total of 12 fonts. The Lucida fonts are stored in the *jre/lib/fonts* subdirectory in the java 2 SDK and in the lib/fonts directory of J2RE. They don't rely on the host operating system.

The benefits that these physical fonts offer are: 1) Consistent look and feel. Your applications will achieve the same look on all the platforms because they use the same font. 2) Rich character sets. These fonts can handle characters of a larger range of languages, especially European and Middle Eastern. For example, the Lucida Sans Regular font covers characters of Hebrew, Arabic, German, and English. You can create fully multilingual applications for the supported languages without having to switch fonts.

There are two disadvantages of this approach. 1) Not all these fonts are supported in all implementations of J2RE—note that these fonts are not platform independent; they are part of J2RE or Java 2 platform; 2) Not all Unicode characters are covered.

Especially, not all math symbols are supported. That is, No.1 of our criteria is not satisfied. Therefore, this is not the font approach we want.

The 2nd font approach—using physical font names

The advantage of using physical font names is that it not only allows an application to take full advantage of all available fonts, but also achieves maximum language coverage. After the math fonts are installed, this approach satisfies all the first three criteria we came up for selecting a font approach. The disadvantage is related to our Criteria No.4, the easiness to implement. Using physical fonts makes it harder to program. We will discuss this issue further in the section of "Implementation".

Now, before we can make our final decision, we need to look at a more traditional font approach of Java, using logical font names.

Excluding the 1st approach—using logical font names

The major advantage of using logical font names is that they are guaranteed to work anywhere. Also, text rendering can be done in at least one language, the one the host operating system is localized for. In addition, a host operating system is often localized for much larger range of languages. With math fonts installed locally, this approach satisfies our first two criteria for using fonts.

The disadvantages include: 1.) An application could have different looks at different places, because the actual fonts used for rendering the text vary from implementation to implementation, from platform to platform, and from locale to locale. 2.) The range of characters that can be rendered is limited by the mapping mechanisms.

For example, Chinese characters can only be rendered on Chinese localized host operating system, not on other localized systems even with Chinese fonts installed. 3.) More importantly, this approach does not satisfy our criteria No.3 and No.4, which is illustrated in the process of handling the font.properties files discussed below.

Overview of the font.properties file

The *font.properties* files are properties files as specified by the Properties class (*java.util.Properties*) and are loaded through that class.[17] They are used in Sun's J2RE to map logical font names to physical fonts. Different host operating system versions and locales have different files to support the mappings. The properties file we are interested in at this stage is the one for English environment, which is the font.properties file with no suffix on its file name: "font.properties".

Names used throughout the font.properties files. The following names are used throughout the properties files:

logical font names: serif, dialog, dialoginput, sansserif, and monospaced

style name: bold, italic, bolditalic, and plain

platform font name: the physical font name such as "Times New Roman"

index number: an integer specifying the order of searching within the set of entries for the same logical font and style for font glyphs with Unicode or java String encoding.

windows charset name: ANSI_CHARSET, SYMBOL_CHARSET

The structure of the font.properties files. The *font.properties* file has entries performing one of the following tasks: font mapping, name aliasing, font file

specification, default font definition, font character encoding, exclusion range information, and the charset specification for text input. Since the name alias, default font definition, and charset for text input are deprecated, we will examine only the rest of the entries.

Font mapping entries: A typical entry that maps a logical font name to a physical font name has the following format:

<logical font name>.<style name>.<index number>=<platform font name>, <windows charset name>, other

For example:

```
serif.0=Times New Roman, ANSI_CHARSET
serif.1=WingDings, SYMBOL_CHARSET, NEED_CONVERTED
serif.2=Symbol, SYMBOL_CHARSET, NEED_CONVERTED

serif.bold.0= Times New Roman, ANSI_CHARSET
serif.bold.1= WingDings, SYMBOL_CHARSET, NEED_CONVERTED
serif.bold.2= Symbol, SYMBOL_CHARSET, NEED_CONVERTED
```

When the style name is not given, plain is used.

Here, logical font "serif" is mapped to three physical fonts: "Times New Roman", "WingDings", and "Symbol". When rendering a character using "serif", the runtime checks these three fonts in the order specified by the indices, with 0 having the highest priority and 2 the least, and uses the first font that can render the character and is not excluded (see below). If java runtime couldn't find a proper font that can render that character, a square or question mark will be drawn in that character's place.

The exclusion range entries. They specify Unicode character ranges that should not be rendered with a given font. This is used when you want to place a font with a large character set early in the search sequence, but want to draw some of the characters with a different font. An entry of this looks like the following:

exclusion.<logical font name>.<index>=char-char, char-char

For example:

```
exclusion.dialogue.0=0500-20ab, 29ad-ffff
```

Font character encoding entries. These entries indicate which character encoding AWT should use when accessing the corresponding fonts. A typical entry looks like this:

```
fontcharset.dialogue.0=sun.io.CharToByteCp1252
fontcharset.diologue.1=sun.awt.windows.CharToByteWingDings
fontcharset.diologue.2=sun.awt.charToByteSymbol
```

where CharToByteCp1252 is the Microsoft Windows Codepage for Latin I characters. The fontcharset entry for dialogue.1 indicates that, to draw a WingDings glyph, the Unicode encoding used in a java string should be converted to another encoding scheme using the sun.awt.windows.CharToByteWingDings converter.

Font file names entries. These entries contain the actual font file names. Listing a complete set of font file names reduces initialization time when mapping is conducted. A typical entry looks like this:

```
filename.Times_New_Roman=TIMES.TTF
```

The above-mentioned four kinds of entries are the ones that need to be changed when we add fonts to the java runtime by changing the *font.properties* file.

Adding fonts to Java run time through the font.properties file

To add fonts to java runtime, we need to perform the following 4 steps:

1) Install the math fonts;
2) Write your own fontcharset Converters;
3) Add your converters and fonts to the font.properties file;
4) Ensure your new converter is visible to the java runtime.

Step 1: Install Math Fonts: Go to the following website, and download fonts from there: http://support.wolfram.com/mathematica/systems/windows/general/MathFonts_TrueType.exe. Or you can go to the CVS repository; I uploaded a zip file of the fonts.

Step 2: Write your own fontcharset Converter. The charset converter converts Unicode, or Java String, encoding, to the encoding (index) of the font. For font drawing, the JDK 1.1 Runtime uses the charset converter that is the subclass of sun.io.CharToByteConverter. Following is an example given by the java documentation on how to create a charset converter.[18] In this example, it is assumed that the font in use contains 256 glyphs indexed from 0x00 to 0xff, and that the font's glyphs correspond to Unicode 0xe000 – 0xe0ff.

```
package mypkg.converter;
import sun.io.CharToByteISO8859_1;
import sun.io.CharToByteConverter;
import sun.io.ConversionBufferFullException;

public class CharToByteMyFont extends sun.io.CharToByteISO8859_1 {
        /**
                This method indicates the range this font converts
        */
        public Boolean canConvert(char ch) {
                if (ch >= 0xe000 && ch <= 0xe0ff)       return true;
                return false;
        }
        /**
                This method converts the Unicode to this font index
        */
        public int convert(char[] input, int inStart, in inEnd, byte[] output, int output, int outEnd)
                        throws ConversionBufferFullException {
                int outIndex = outStart;
                for (int i = inStart; i < inEnd; i++) {
                        char ch = input[i];
                        if (ch >= 0xe000 && ch <= 0xe0ff) {
                                if (outIndex >= outEnd)
                                        throw new ConversionBufferFullException();
                                output[outIndex++] = (byte)(ch – 0xe000);
                }
                return outIndex – outStart;
        }
        /**
                This methods indicates the charset name for this font
        */
        pubic String toString() { return "MyFont";}
}
        -- excerpted from http://java.sun.com/j2se/1.3/docs/guide/intl/fontprop.html
```

For each math font we installed, we have to write a similar converter. We need to read in the font encoding tables, get the information for the Unicode to be drawn and the corresponding font index for it. The difficulty is that the Unicode each math fonts draws does not fall so neatly in one range; it's kind of scattered, and that the number of glyphs each math fonts render are not the same. This makes it harder to write the correct canConvert() and convert() functions.

Step 3: Add your converters and fonts to the Java runtime by specifying the converter classes and font names in the *font.properties* file. We must first add **font mapping entries** in the *font.properties* file by adding an index entry to each logical font names and style. For example, to add a serif font to the *font.properties* file, we should insert the following line:

"serif.3=<Math1>"

Note that the index number for any one font must be continuous, which is a requirement of Java runtime. Had we added a number that was discontinuous, such as serif.5, the Java runtime would not use the entry.

Next, we must add **font character encoding entries** for a math font—this defines the converter for this font. The following line is the fontcharset entry that uses the converter created in the Java code example:

fontcharset.serif.3=mypkg.converter.CharToByteMyfont

We should do the same for each logical font in the font.properties file.

You can also add an entry to the **exclusion range** if you want to limit the range in which glyphs are searched in a physical font. It's also a good idea to add a line to the **font file names entries,** which will reduce initialization time when mapping is conducted: "Filename.Math_1=Math1.TTF"

Step 4: Make your converters visible to the Java runtime—add the class paths to your converters to the classpath of your application. Again take the above example for example, put the converter class under $JDK_HOME/classes/myown_package directory.

Disadvantages of using logical font names:

As we can see from the process of handling the *font.properties* file, our criteria No.3 and No.4 are not met: 1) Easiness to use. The user has to change their *font.properties*; they also have to change classpath; 2) From the implementation point of view, we have to write CharTOByteXXXX Converter for each math font we use, where XXXX is the name of the font that needs a converter.

Due to these advantages, especially the inconvenience to use, we decided to use the physical font name to render math expressions.

Part II: Additional Challenges – Font Related Problems in the Process of Rending a Math Symbol using Physical Font Names

In the process of drawing a math symbol covered by the math fonts we installed, using physical font names, we encountered the following issues:

1.) Why could not we render a symbol by calling "drawString(…)" from Java API ?

2.) How to scale a glyph proportionally? How can we guarantee the legibility and correctness of the after-scaled glyphs?

3.) Can we display a symbol correctly on the computer screen at small sizes (the typical sizes a document usually adopts), which has far lower resolution than a laser printer?

Answering these questions require extensive knowledge about font itself.

1.) Why couldn't our application display any math symbol using drawString() even though I have for math fonts installed?

The *drawstring(String, int x_pos, int y_pos)* method is the most-often called method to render texts in Java. We have used it to render math symbols in native fonts and Lucida fonts. However, we were not able to do so with the math fonts we installed.

To find the reason for this problem, we first looked into what Java's rendering engine does when drawstring() is called, and then looked at what really got installed when we installed a font, in the hope that we could find the gap that needed to be bridged in order to use that method.

According to Java documentation, when drawstring() is called, 4 phases are gone through before that string actually is displayed on the screen: 1) Determine the set of glyphs required for render: the current font is asked to convert Unicode characters to a set of glyphs for representation. 2) The current font is queried to obtain the outline for the indicated glyph. 3) Character outlines are filled. 4) Current *Paint* is queried for a *PaintContext* (e.g. current color).[19]

Of these 4 steps, step 4) just determines the color in which a character is to be displayed; it has nothing to do with whether or not a character can be drawn. Therefore, we can skip examining that. Step 2) and 3) are totally controlled by the shaping and

layout algorithm the selected font implements. Unless the font files is corrupted, chances are very little that something is wrong with these 2 steps, because these fonts have been out there for some time and have been used in quiet a few math software like MathType and Mathemetica. So most likely step 1) is where the problem lied.

Selecting proper glyph according to its name and code is the first step of any font processing algorithm of a printing engine. It is closely related to the font encoding tables. Either this information is not installed[5] or it is installed but Java runtime couldn't find it. A closer look into the content of a true type font file will shed light on this problem, as all the math fonts we installed are TrueType fonts.

According to True Type Specification of Apple, the founder of TrueType technology, any valid TrueType font file must have the following tables[20], among which the 'glyf" table stores the outline descriptions of all the glyphs in that font, 'hmtx', the metrics for layout, and 'cmap', the mapping from Unicode to glyph indices.

Table 1: The required tables

Tag	Table
'cmap'	character to glyph mapping
'glyf'	glyph data
'head'	font header
'hhea'	horizontal header
'hmtx'	horizontal metrics
'loca'	index to location
'maxp'	maximum profile
'name'	Naming
'post'	PostScript

[5] The confusion resulted from my reading of an article about digital type. That article argues that a true type font file consists of two parts: a font outline file and a font metrics file. It does not mention any thing about whether glyph selection info is included. See André, Jacques, "Font Metrics", in Hersch, Roger D., Visual and Technical Aspects of Type, Cambridge University Press, 1993, p.67

That is to say, the font files we installed already have all the information needed for drawing a glyph when its corresponding Unicode is specified. So the mapping information must be invisible to Java run time.

A further investigation into Java's FAQ page on Internationalization gave us some additional clues. When asked why characters cannot be displayed in a certain language even when font for that language has been installed, Sun's answer was "two possibilities": 1) the application may not be selecting the fonts correctly, or 2) the font may be using an encoding that's not supported by the Java 2 Runtime Environment. We were sure that possibility No. 1 was not the reason because we specified the font to use; that left us with possibility No. 2.

As we know, all the characters used in a Java program adopt the Unicode encoding. However, Java does support other encoding mechanisms, for each of which a converter is defined and packed into rt.jar or i18n.jar.[41] The classes *java.io.InputStreamReader*, java.*io.OutputStreamWriter*, and *java.lang.String* can convert between Unicode and a number of other character encoding schemes it supports. Recall the *font.properties* files we discussed in Part I: each font mapping entry specifies a CharToByteXXXX converter, where XXXX is the canonical name of the physical font to which that logical font maps. When a certain font is selected to draw a character encoded in Unicode, the associated converter is called to return the glyph index for that Unicode. That glyph index is in turn used by the font to locate and render the associated glyph with whatever shaping and layout algorithm the font rendering engine implements.

The complete list of encoding sets supported by java (http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html) shows that the math

fonts we installed use encoding mechanisms that are not supported by Java. For example, in font "math1.ttf", abstract character "question mark" is encoded as "0x22", while in Unicode, it should have been encoded as "0x003f"; nor is the character encoding used in "math1.ttf" among the other encoding schemes supported by Java. Therefore, we need to provide Java runtime with a converter, which tells Java what encoding point it should use when it tries to draw a Unicode-encoded character with a math font we installed.

We had two ways to do that: 1) to write a converter for each math font we installed and follow the steps specified in "adding fonts to java runtime" in Part I; in this way, we can use the drawstring() method, but we have to change the *font.properties* file and add the class path of the converter to our application's path, which is exactly the same drawback that discouraged us from using logical font name as our font approach. 2) We still need to do the converting, but instead of changing the *font.properties* file and calling drawstring() to draw math symbol, we created a *GlyphVector*, each element of which encapsulates the font to be used and the index of the glyph to be rendered. We will talk more on the 2nd approach in the section "Implementation".

2.) Can we scale a glyph to whatever size we want without loosing legibility and correctness of the appearance?

Scaling a math symbol happens very often during the process of displaying a math expression. For example, we use superscripts and subscripts a lot in a math expression, the sizes of which should be ¾ as big as that of the hosting symbol. For another example, it would look better if the size of the summation symbol is 1.5 as big as the symbols for the expression after it.

Java's ways of scaling are two: we can either resize the font or use affine transformation[6] to scale the x- and y- coordinate values to desired size. But the question is: will the scaling be done proportionally using the math fonts we installed? How can we be sure of that? We tested some symbols by scaling them to very small and very large sizes and the results were good, but can we say the same for all the symbols scaled to whatever sizes? The challenge for us was to find some theoretical basis. Meeting this challenge requires a deep understanding of the fonts (outline fonts) we are using, the difference between those fonts and bit map fonts, and how outline fonts are rendered (scaled).

Bitmap fonts describe glyphs as a pattern of dots and store them as bit maps. Each map specifies a specific font size, style, and orientation. For example, "12-point Times New Roman characters in the upright orientation". The size of the dot pattern cannot be changed to get larger or smaller character glyphs.[21] Nor can you apply *Affine transformation* to bitmap fonts to get high-quality transformed bitmaps. To get a different sized glyph, a bit map font for that size must be pre-installed.

Outline fonts, on the other hand, describe glyphs by their shapes—points, line segments, and curves. At the lowest level, each glyph is described as a sequence of points on a grid. These points make up line segments (called "on curve"), as well as curves, consisting of "on curve" points as well as "off curve" points (the control points). At a higher level, a glyph is made up of contours, closed shapes defining the glyph. For examples, "B" is made up of 3 contours.

[6] Series of rotation, translation, and scales of a graphic object are affine transformations. Affine transformations preserve parallelism, but not lengths and angles.

More technically speaking, each glyph in an outline font is defined as a mathematical model, Bezier Curve for Type 1 fonts and B-Spline for TrueType fonts, of its outline.[22] "Bezier curve is a parametric cubic (or third order) curve defined by its two end-points and two control-points, which in general are not on the curve…the control-points define the tangents of the curve as it leaves the two end-points. A Bézier curve always remains inside a polygon drawn around all of its points (the "convex hull"). The curves of TrueType are a quadratic (second order) version of Bézier curves, having two end-points but only one off-curve control-point. The single off-curve point then defines *both* tangent vectors, while the curve remains within the triangular convex hull. A very useful property of Bézier curves is that by simply transforming the control points of a curve by a certain matrix, the resulting curve is exactly as though every point *on* the curve was transformed by that same matrix."[7] This is why a glyph in an outline font can be scaled to any size or rotated at any angle by affine transformation without loosing correctness of glyph appearance.[23]

Because the process of rendering an outline font's glyph heavily involves scaling, we will look into that process next.

The rendering of a glyph in an outline font consists of 3 steps.

1) Scale the master outline description of a glyph to a specified size. This process involves changing the device independent em units[7] for the points that make up the glyph outline to the device dependent point numbers[8] representing locations in a device-specific

[7] The units for an em square, the imaginary tablet on which a character is drawn, the area covered by a square whose side is generally equal to the width of character "M". The greater the units, the greater subtlety of a design. Usually, the number of units is a power of 2 for faster scaling. 2048 units per em is very common.

[8] A scaled outline point can occupy any position expressible as a 1/64 of a pixel, that is as a 26.6 fixed point number: 32-bit fixed-point numbers with 6 fractional bits.

pixel grid. The scale used is: (point size * resolution in dpi) / (72 points per inch * units_per_em), where units_per_em is the resolution of the grid on which the master outline was originally defined.

2) Reshape ("grid-fitting") the scaled outline according to associated instructions (also called "hints"). This involves moving the scaled outline points to new locations (the nearest 1/64 pixel on the displaying device), so that symmetries, stem width, and other important glyph features are preserved even when glyphs are displayed in low-resolution devices such as a computer screen. For example, a typical instruction would say: "At 12 ppem (pixels per em) move control point 5 by 1.25 pixels".

3) The reshaped outline is "scan-converted" to produce a bitmap image for raster display[9]. During this step, a set of rules are applied to determine which pixels should be turned on when the glyph image is displayed. (Software doing this is called "scan-converter".)

The math fonts we installed are TrueType fonts. They store a master outline description for each glyph. When our application requests a particular glyph at a specific size for screen display, the above 3 steps are executed and the correct bitmap are created, which is then cached, and copied from the cache whenever it is called for display.[24] Therefore, the proper scaling is taken care of by the math fonts themselves, and we can be sure that the after-scaled glyphs retain their correct appearances.

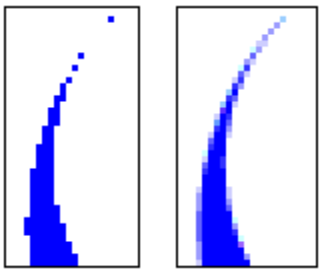
3) Aliasing problem on low-resolution device and the ways on how to improve the legibility on low resolution displaying device like screen

[9] Raster – a rectangular array of points or dots. Scan-conversion takes a primitive (triangle description) and determines which pixels on a scan line (a row of pixels in the raster) to turn on.

Aliasing is the well-known effect on all pixel devices, especially on the ones with low resolution such as computer screen, where diagonal and curved lines appear to be very jagged. When the pixels are large, like on computer screens, some kind of remedy is highly desirable.

Anti-aliasing

Anti-aliasing, naturally enough, is the name for techniques designed to reduce this effect by shading the pixels on the edge of a graph.[25] Rather than just using black or white pixels, printing engines can improve appearance of a glyph by judicious use of gray pixels of various shades - as well as black and white. An edge pixel will become light or dark gray, depending on how much black you'd find if you zoomed in on that pixel. The effect is of a higher resolution, improving aesthetics and, if done with skill, legibility. Anti-aliasing is also called “gray-scale fonts”.[9]



left: standard rendering; right: anti-aliased rendering

Fig 1: Comparison between standard rendering and anti-aliased rendering

In this figure, we can see how anti-aliasing is used to achieve the effect of a higher resolution for the same graphic elements: turning on some additional gray-colored removes the "steps" and restores missing parts of the image.[26]

Java 2D supports antialiasing renders by enabling the pen to fall partially on both of the adjacent pixels. Turning on antialiasing feature of java rendering engine can make math symbols look smoother even the font size is as small as 10, 12, or 14, the popular size a document is in. To turn on anti-aliasing rendering, call the following method:

```
Graphics2D.setRenderingHint(java.awt.RenderingHints.KEY_ANTIALIASING,
                            java.awt.RenderingHints.VALUE_ANTIALIASING_ON).
```

However, when point size is small, anti-aliasing alone won't achieve the desired eligibility. A more advanced and complicated technique called "hinting" is needed.

Hinting

Scaling an outline's control point co-ordinates to the small size of a computer screen can result in the losing of important features of a font: stem weights, crossbar width, serif details, and symmetry. After all, the "mathematically correct" pixels for outlines scaled to a given size do not necessarily mean aesthetics and legibility. Hinting is the technique for restoring, as far as possible, their aesthetics and legibility by equalizing the weight of stems and preventing parts of the glyph from disappearing.

At the lowest level, hinting defines exactly which pixels to turn on to create the best bitmap at small sizes and low resolutions. Since it's a glyph's outline that determines which pixels will be part of a glyph's bitmap, hinting often involves the modification of the outline by moving the control point of the outline's contours. The following figure shows how the outline adjustment is carried out. For more information

about the hinting technology, go to http://www.microsoft.com/typograph/hinting/versus.htm, and http://www.micorsoft.com/typography/hinting/what.htm

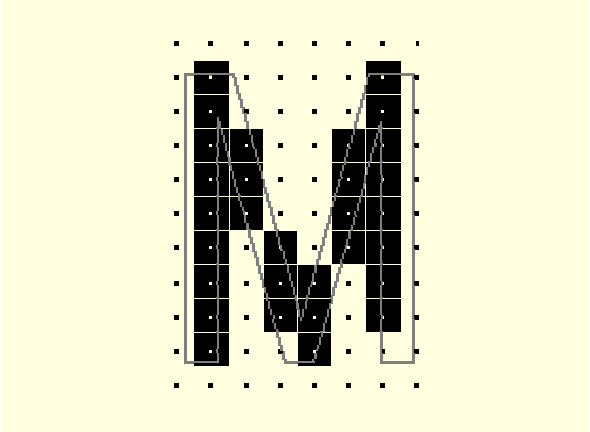


Figure 2a. An outline that hasn't been grid-fitted. Note how poorly the outline corresponds to the pixel pattern, and above all how awkward the bitmap of the M is.

Figure excerpted from http://www.microsoft.com/typography/hinting/what.html

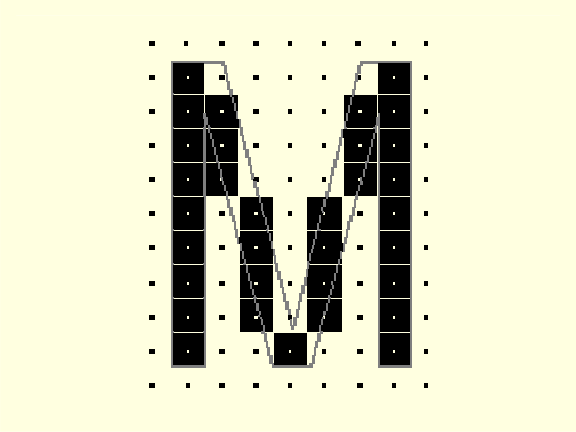


Figure 2b. The same outline grid-fitted. Now the outline has been adjusted to fit snugly around each pixel, ensuring that the correct pixels are turned on.

Figure excerpted from http://www.microsoft.com/typography/hinting/what.html

Part III: Implementation

As we discussed in the introduction, the core task of the current project is to build a Java MathML rendering engine that can display math expressions, with both the expression as a whole and each term in it individually accessible to a user and the document around the expression. To achieve that goal, a lot of things need to be done such as parsing a MathML expression and building a hierarchical structure to layout math objects. However, having the structure alone is not enough; the mansion of the rendering

engine also needs the "building bricks"—the "terminal" math objects that know how to draw themselves to any given size. My implementation supplies such bricks by providing 3 objects: 1) the "MathFonts" object, who knows which font to call and glyph to use when given a Unicode, 2) the "NormalSymbol" object, who can draw itself to the size given to it, and 3) the "StretchySymbol" object, who can stretch itself horizontally to any given length, vertically to any given height, or both horizontally and vertically to any given size.

MathFonts (MathFonts.java)

The main functionalities for a "MathFonts" object are 1) to read in and store font encoding tables, which map a Unicode to a glyph index, for each of the 5 math fonts: "math1.ttf", "math2.ttf", "math3.ttf", "math4.ttf", and "math5.ttf", 2) to find the proper fonts to call and glyphs to use when given a Unicode, and 3) to provide some instructions on the order in which components of a stretchy math symbol are drawn.

Process the font encoding tables

The font encoding tables we used for our project were created by the Mozilla organization. **[27]** A font encoding table in its plain text format consists of 240 entries starting with hexadecimal index 0x20 and ending with 0xFF. One font encoding file exists for each installed font. Each entry in such a file looks as follows: [28]

```
0x20  0x0020  #Space
0x21  0x0021  #Exclamation mark
0x22  0x2200  #For all
...
0x7F ?nbsp
...
- 0x89  0x00D7  #Multiplication sign - see 0xB4
  0x92 >PUA     # &DoubleLongLeftArrow; &Longleftarrow;
```

```
…
0xEC  0x007B:T  #Left curly bracket top
0xED  0x007B:M  #Left curly bracket mid
0xEE  0x007B:B  #Left curly bracket bottom
0xEF  0x007B:G, 0x007D:G  #Curly bracket extender
…
0xFF   0xFFFF
```

As we can see, every entry has 4 columns, separated by white space, with the 1st column for the marker "-", 2nd for the glyph index (i.e., encoding point), 3rd for either a Unicode (with or without annotaions) or ">PUA", and 4th for the comments. MathFonts.java reads in this file, converts 3rd and 2nd column if "-" is not in the 1st column and 3rd column is not ">PUA", and stores the converted result, a pair of **(Unicode, glyphCode)**, in a hash table, where Unicode is first converted to an decimal integer and wrapped in Object Integer, and **glyphCode** is computed as follows:

glyphCode in decimal = glyphIndex in decimal – firstGlyphIndex (0x20) in decimal + 3

For example, the glyph code for the the 3rd entry in the above table, that is, for Unicode "\u2200" is computed as

glyphIndex in decimal = $(0x22)_{16}$ = 2*16 + 2 = $(34)_{10}$;

firstGlyphIndex in decimal = $(0x20)_{16}$ = 2*16 + 0 = $(32)_{10}$;

glyphCode in decimal = 34 – 32 + 3 = 5;

Therefore "5" is the glyph code used when "\u2200" is to be drawn in font math1.

How, you may ask, did we come up with the formula for computing glyph code? As a matter of fact, this formula resulted from out experimentation. When trying to pass the correct code to the drawing procedure of Java API, we found the following pattern:

1. The hexadecimal index given in the font encoding table, for example, "0x22", couldn't be used directly for drawing purposes. That is, to draw "\u2200", the index

"0x22", whether in hexadecimal or in decimal, was not the index expected by the Java drawing procedure;

2. When we passed an array of consecutive integers (each less than 256) to the Java drawing procedure, a group of glyphs were drawn in exactly the same order as shown in the font encoding tables;

3. The glyph code expected by the Java drawing procedure is in the range of [4,), with 4 always be the correct value used to draw the first visible glyph in a font. That is, 4 is the glyph code for the 1st glyph, 5 is the code for the 2nd glyph, 6 is the code for the 3rd glyph, and so on…

From the above patterns, we drew the following conclusions:

1. The correct value expected by the Java drawing procedure to render a character is relative to the **offset** of the associated glyph index (0xXX) from the beginning of the font encoding table (0x20), that is $(0xXX - 0x20)_{10}$;

2. The exact value for the glyph index is the **offset + 3**.

Having these observations, we came up with the formula for computing glyph code. However, there are two exceptions. 1) For "math3.ttf", when a glyph index is greater than "0x7F", all the glyph codes computed should be reduced by 1; 2) For "math5.ttf", all the glyph codes computed should add another 4 to it.

The question remained "Why the formula? Why the exception?" At first, we thought that it's Java's problem, and looked into Java documentation and even source code, for everything related to fonts, glyphs, graphics, and rendering, only to find nothing that could answer the question. Not until recently when I looked into True Type specification (of Apple and Microsoft) did I find some leads.

First Four Glyphs in Fonts

According to Microsoft's specification, the first 4 glyphs in any TrueType font is reserved for the following 4 glyphs at the following glyph index (glyph ID) as shown in the following table:[29]

Glyph ID	Glyph name	Unicode value
0	.notdef	undefined
1	.null	U+0000
2	CR	U+000D
3	Space	U+0020

This specification turns out to be in conformity with Apple's original specification, which lists all the characters that must be assigned to glyph codes 0 and 1.

Glyph index 0 and its Outline

"The first glyph (glyph index 0) *must be* the MISSING CHARACTER GLYPH. This glyph must have a visible appearance and non-zero advance width."[30] The following list of character codes should be treated as Missing Character glyph.[30]

1. 0x0001 START OF HEADING
2. 0x0002 START OF TEXT
3. 0x0003 END OF TEXT
4. 0x0004 END OF TRANSMISSION
5. 0x0005 ENQUIRY
6. 0x0006 ACKNOWLEDGE
7. 0x0007 BELL
8. 0x000A LINE FEED
9. 0x000B VERTICAL TABULATION
10. 0x000C FORM FEED
11. 0x000E SHIFT OUT
12. 0x000F SHIFT IN
13. 0x0010 DATA LINK ESCAPE
14. 0x0011 DEVICE CONTROL 1 (X-On)
15. 0x0012 DEVICE CONTROL 2
16. 0x0013 DEVICE CONTROL 3 (X-Off)
17. 0x0014 DEVICE CONTROL 4
18. 0x0015 NEGATIVE ACKNOWLEDGE
19. 0x0016 SYNCHRONOUS IDLE
20. 0x0017 END OF TRANSMISSION BLOCK
21. 0x0018 CANCEL
22. 0x0019 END OF MEDIUM

23. 0x001A SUBSTITUTE
24. 0x001B ESCAPE
25. 0x001C FILE SEPARATOR
26. 0x001E RECORD SEPARATOR
27. 0x001F UNIT SEPARATOR
28. 0x007F DELETE

This list (together with the next 4) also explains why all the encoding tables start with 0x20 and lack entries for glyph index from 0x00 to 0x1F.

The *.notdef* glyph is very important for giving user feedback that a glyph is not found in the font, therefore, it shouldn't be left without an outline. The commonly used outlines for an undefined character are (the following figure is excerpted from http://www.microsoft.com/typography/otspec/recom.htm):

Fig. 3: glyph shapes for .notdef characters

Glyph index 1 and its outline

"The second glyph (glyph index 1) *must be* the NULL glyph. This glyph must have no contours and zero advance width."[30] The following character code must be map to glyph index 1:

0x0000 NULL
0x0008 BACKSPACE
0x000D CARRIAGE RETURN (in a right-to-left font)
0x001D GROUP SEPARATOR

Glyph index 2 and its outline – no contours and positive advance width[10]

[10] "There are three kinds of glyph in TrueType: simple, composite and zero-contour. **Simple** glyphs contain compressed outline data and hinting instructions. **Composite** glyphs refer to other glyphs for their outlines. Referenced components may be simple or composite themselves. **Zero-contour** glyphs, such as the space character, signify some behavior in text but do not have a printed form. Zero-contour glyphs, like all glyphs, have a left side-bearing and advance width stored in the 'hmtx' table, but these metrics cannot be hinted as they can with simple and component glyphs."—excerpted from http://www.truetype.demon.co.uk/ttoutln.htm

0x000D CARRIAGE RETURN (in a left-to-right font)

Glyph index 3 and its outline – no contours and positive advance width

0x0020 SPACE

Other characters that must map to a glyph with no contours and positive advance width:

1. 0x0009 HORIZONTAL TABULATION
2. 0x00A0 NO-BREAK SPACE (;nbsp)

From the above, we can see that the first 4 glyph indices (0 to 3) are reserved, therefore it is understandable why we need to add a 3 to the offset. The exception with math3 could be explained in this way: NO-BREAK SPACE (in entry *0x7F ?nbsp*) shares the same glyph as SPACE (in entry *0x20 0x0020 #SPACE*), that is, uses same index as SPACE, therefore, all the glyphs after it can move their index forward by 1. This is possible, because in Apple specification, SPACE and No-BREAK SPACE are put in the same group – 0 contours and positive width.

As to the exception with math5, we noticed that at the beginning of the font table (see Fig. 11 on page 74), in addition to SPACE (0x0020, whose index is 3) and BACKSPACE, which are mapped to index 1 as specified by Apple spec), there are 4 additional glyphs for the keys HOME, PGUP, PGDN, END on a computer's keyboard. These glyphs share the same attributes as 0-contours glyphs. That is, they don't have print forms, but signify certain behaviors in a text, and therefore should also have positive advance widths. In reality, as we know, these four keys have different functionalities that can not be replaced by one another, therefore, they cannot share glyphs. So my guess is that glyph index 4, 5, 6, 7 are reserved for these 4 characters, and therefore, all the following glyphs have to add another 4, altogether 7 to their offsets.

NormalSymbol (NormalSymbol.java)

Object *NormalSymbol* is responsible for drawing itself to specified size. There were two major issues to consider: 1) what drawing procedure of java API to use, and 2) how to scale a character to a proper size without loosing legibility, correctness of shape, and aesthetics. The capability to scale a character properly is totally up to the font itself, which we have discussed in Part II in great detail. Therefore, we will focus on the first issue only.

GlypyVector vs. *drawstring()*

As we've already mentioned previously, the *drawstring()* methods are the most often used methods to render texts. We can make use those methods if we want to, so long as we provide a CharToByteFontName converter for each math fonts we installed. As a matter of fact, writing those Converters are no difficulty than writing the MathFonts.java, most likely, a lot easier. However, a simple converter alone cannot make full use of the font encoding tables; for entries with same Unicode but different annotations, only one entry, thus one glyph can be used.

Recall the font encoding table we discussed before. The 3rd field is either >PUA or Unicode with or without annotations (0xNNNN:0, 0xNNNN:1…). One usage of annotations is to specify sizes for glyphs associated with the same Unicode: An annotation with digit '0' means "normal-size", therefore that glyph can be used in a normal text run. Digit '1' means 'big' (Tex's terminology), '2' means 'Big", '3' menas 'bigg', '4' means 'Bigg', and so on. The ordering is relative to the font family.

With a CharToByteFontName converter, one Unicode can only map to 1 glyph index, therefore, we lose all the other choices we have. But more importantly, with a simple converter, we are not able to draw a stretchy symbol (discussed in next section), which requires all its parts accessible. These parts have the same Unicode, but with different annotations (0xNNNN:T, 0xNNNN:G…). A converter cannot handle this situation. We need some procedures that can deal with glyph index directly. Fortunately, Java has APIs that can handle with glyph and a vector of glyphs directly.

Out of the above consideration, we decided not to write a converter, and use the methods that can muddle with glyph indices directly, even though the *drawstring()* methods are enough for drawing normal math symbols.

How to use methods that handle glyph indices to draw a normal symbol

Using *GlyphVector* related methods to draw a symbol is much more complicated than using drawstring(), which just involves 2 lines of code. To actually display a symbol using these methods, you must do the following:

1) specify the font:

```
Font font = new Font (name, style, size)
```

2) create an array of glyph codes:

```
int[] gCode = { …},
```

where each glyph code is the index of the glyph to be rendered, as computed with the formula on page 30;

3) create a *FontRenderContext*:

```
FontRenderContext frc = Graphics2D_object.getFontRenderContext();
```

4) create the *GlyphVector*:

```
font.createGlyphVector (FontRenderContext, int []gCode),
```

5) *drawGlyphVector()* at once OR loop to draw glyphs in that vector one by one;

5a) If draw the vector at once:

```
Graphics2D_object.drawGlyphVector (GlyphVector,
                                   float x_coordinator,
                                   float float y_coordinator)
```

5b) If loop, for each element (loop index i) in the *GlyphVector* (gv):

1. get its position (origin where the glyph is to be drawn):

```
Point2D p = gv.getGlyphPosition(i);
```

2. create an *Affine transformation*, which converts user space coordinate system to device dependent coordinate system, to adjust the origin where this glyph is to be drawn:

```
AffineTransform at = AffineTransform.getTranslateInstance (p.getX(),
                                                           p.getY());
```

3. get the glyph outline:

```
Shape glyph = gv.GetGlyphOutline(i);
```

4. create the transformed shape (outline):

```
Shape newGlyph = at.createTransformedShape (glyph);
```

5. fill the shape (outline):

```
Graphics2D_object.fill (newGlyph);
```

The disadvantage of using drawGlyphVector() is that you have no control of where each glyph should be drawn; Java will do the layout according to the metrics information stored in the font file for that font.

StretchySymbol (Stretchy Symbol.java)

What's a stretchy symbol?

A stretchy symbol is a math symbol that can be rendered by assembling several, usually 3, but sometime 2 or 4, parts together either from top to down or from left to right. For example, the integral sign "∫" can be drawn by putting these 3 parts together: "top glyph" " ⌠ " on the top, "extender" " ⎮ " right under it, and "bottom glyph", "⌡ " at the bottom. This is often done when the height of "∫" at a normal size is not big enough and needs to be extended. The extender "|" can be repeatedly drawn as many times as needed. Similarly, "14243" can be drawn by placing "left glyph""1", "glue glyph" (extender) "4", "middle glyph" "2", another "glue glyph" "4", and "right glyph" "3" next to each other when needs arise for "14243" to be lengthened to a certain length. By repeating drawing the "extender", in this case " ⎮ " and "4", as many times as needed, we can extend a stretchy symbol as long/high as we need.

The parts constitute a stretchy symbol are represented in the font encoding table as annotated Unicode: 0xNNNN:T, 0xNNNN:B, 0xNNNN:L, 0xNNNN:R, 0xNNNN:M, 0xNNNN:G, where "T" stands for "TOP", "B" for "Bottom", "L" for "LEFT", "R" for "RIGHT", "M" for "MIDDLE" and "G" for "GLUE". They respectively mean that the glyph should be used as top, bottom, left, right, middle, or as repeatable glue when assembled by parts.[24]

In a sense, "T" (top) and "L" (left) are the same, and so are "B" (bottom) and "R" (right), when it comes to the order of drawing. For stretchy symbols consisting of 3 parts, "T" or "L" is always drawn first once, followed by "G" drawn several times, and "B" or "R" is always drawn once in the end. The same holds for stretchy symbols of 4 parts, with the exception of symbol square root.

How is a stretchy symbol rendered?

To a certain degree, the rendering of a stretchy symbol depends on the order in which the glyph codes for its parts are stored. The storing of glyph codes for the parts of a stretchy symbol can be categorized as 4 cases: (Below, what's actually stored is the glyph index corresponding to each part, not those letters themselves)

1. General case – 3 parts. No matter they are "T"-"G"-"B", or "L"-"G"-"B", they can be added to a vector in that order.
2. 4 parts. Similarly, no matter the symbol is vertically stretchy or horizontally stretchy, the parts can be stored in the order "T"("L")—"G"—"M"—"B"("R"). Example: for parenthesis, " ⎧ ", the glyph codes for the four parts are added to a vector in this order: "⎛", "⎜", "⎝", "⎞".
3. 2 parts. "G" and "B", or "G" and "R" are stored in that order. For example, "↓" in the order of the extender, " ⏐ ", and the bottom glyph, "↓", and "⟶" in the order of "—" and "→"
4. 2 parts. "T" and "G", or "L" and "G" are stored in that order.
5. Square root – 4 parts, stored in this order: "G" (horizontal extender), "T", "G" (vertical extender), "B": "‾", " ⎡ ", "⏐", and "√".

When it comes to the actually drawing, we can just get the needed glyph indices out of the vector in the order they were stored. For Case 1, general case, 1st element, be it the top part (vertically stretchy) or the left part (horizontal stretchy), is always got out and drawn first; then 2nd element, the stretchy part, and finally, the 3rd element, either the

bottom or the right part is rendered. The 2nd element of the vector can be repeated as many times as needed. That whether these elements are assembled from top to down or from left to right is dependent on a flag.

For case 2, we can repeat what we did in case 1, except that the number of times for repeating 2nd element, the extender, should be reduced by half, and that after 3rd element is drawn, 2nd element needs to be drawn the same number of times as it was before 3rd element was drawn. And finally, the 4th element needs to be drawn once. Similarly, there is a flag that decides whether all the parts are packed from left to right or from top to bottom.

For case 3, the 1st element needs to be repeated, and then 2nd element needs only to be drawn once. A flag with show whether the parts should be put next to each other, or on top of each other.

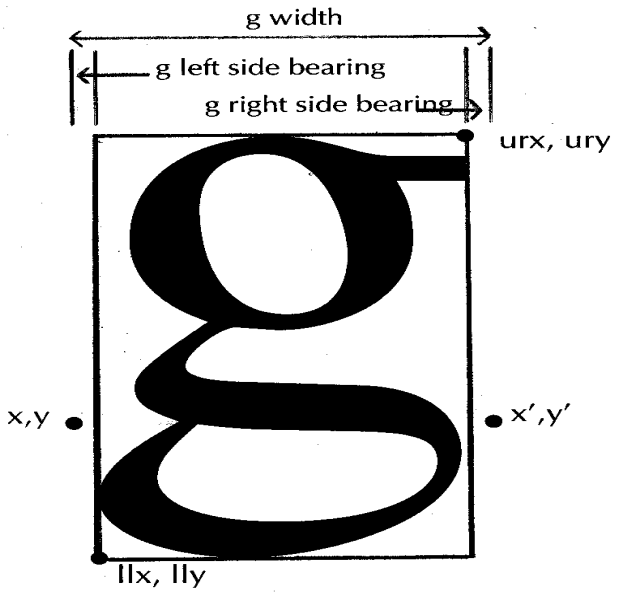
For case 4, the 1st element needs to be drawn once, followed by the 2nd elements being repeated for a number of times. A flag will show how these parts should be assembled.

For case 5, the square root, the 1st element, "−", is repeated as many times as specified (horizontally), then the 2nd element, "┌", is drawn once, the 3rd element, "╷", is repeated (vertically) as many times as specified, and finally the 4th element, "┘" is drawn.

3) Why, when using the same drawing procedures, we can display some stretchy symbols without visible gaps between glue and terminal glyphs, but cannot for other ones? How did we correct the error?

The way we assembled the parts of a stretchy symbol was to move the origin to the right by the width of the tightest bounding box of the glyph just drawn, or down by the height of the same box, hoping to patch all the parts seamlessly. This approach worked for some of the stretchy symbols, but for some, it did not. We couldn't understand why until we had a better understanding of the font metrics for a given glyph.

Consider the font metrics shown for an ordinary Latin letter:



x,y – coordinates of the origin of the character
x'y' – coordinates of the origin of the next character
w_x, w_y – character width
ur_x, ur_y, ll_x, ll_y – coordinates of the bounding box
left side bearing (lsb) = llx;
right side bearing (rsb) = w_x - ur_x

Fig 4: main metric value for painting and spacing a character[31]

Here, current pen position (x,y), is used as the origin of the glyph for "g", and (x',y') is the position where the next glyph's origin lies. Except for the 2 points that define the bounding box, you can see that two more measurements are used: lsb and rsb.

LSB (left side-bearing) is "the offset from a glyph's horizontal origin to its minimum x-coordinate. Usually, with the character origin at zero, it is exactly the same

as the minimum x-coordinate (xMin). The advance width and LSB (the metrics) for each glyph are stored in TrueType's 'hmtx' table. These metrics can be adjusted by hinting to control character spacing at low resolution."[9]

Things I did not know and corrections made to display a stretchy symbol seamlessly

1. An origin does not necessarily lie right on the bounding box; it could be to the left of or inside the box. This means that an "lsb" can be positive or negative, and that lsbs are not necessarily the same for every glyph in the same font file. However, my drawing procedure for horizontally stretched characters assumed that origin is at the left side of the bounding box, that is, lsb = 0; this caused the problem of overlapping between some parts, if lsb for the glyph just drawn is less than that for the glyph to be drawn next, but gaps between others, if otherwise. Therefore, not only should we move the origin by $xMin + w$, where $xMin$ is the x-coordinate value of the up-left corner of the bounding box, and w is the width of the bounding box of the glyph just drawn, but also should we do the following after the moving: the origin by: 1) if $xMin > x_{origin}$, move origin left by $|xMin - x_{origin}|$, which removes gap, 2) if $xMin < x_{origin}$, move origin right by $|xMin - x_{origin}|$, which removes overlap (we didn't do this in code).
2. Baseline always lies at the same line where the origin lies;
3. The coordinate system used in our Java program is the one with y-axis extending downwards, while most part of a glyph is drawn to the opposite direction as far as y-axis is concerned. We can see this fact from the yMin[11] value returned from the Java API for getting the upper-left corner of a glyph's tightest bounding box:

[11] yMin is the y-coordinate value of the upper-left corner of the bounding box.

yMin is always negative. This means that we need to know the height of a glyph and move down the origin by that many units **before** that glyph is actually drawn; otherwise, the glyph will overlap the previous glyph (if wasn't moved down by enough units) or a gap will be produced between the two glyphs.

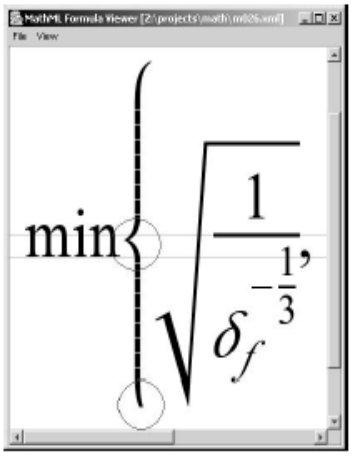
4. Because the origin does not necessarily lie on the lower left point of the bounding box, the actual number of pixels moving down is not necessarily the height of the bounding box as we originally used; it must be carefully re-calculated every time a move is to be made. Otherwise, when origin is inside the bounding box—moving down by the height means that two much space is left for that glyph, a gap will form; when it is outside—overlap. The details of how to calculate that value is elaborated as follows: 1) move the origin down by *h'*, where *h'* is the distance between the baseline and the upper boundary of the bounding box of the glyph to be drawn, 2) draw the glyph, and 3) move the origin down by *h''*, where *h''* is the distance between the baseline and the lower boundary of the bounding box.

4) Repeat assembling extender vs. using *"AffineTransform"*

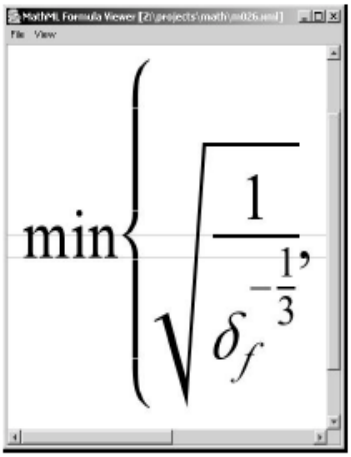
After we corrected our code according to the observations we made above, we can draw all the stretchy glyphs without visible gaps, including the square root, which requires both horizontal stretch and vertical stretch for one symbol. However, sometimes, the terminal glyph may lose some pixels at the end, and a middle glyph might not be aligned right (see Fig 5).

Two reasons might have caused that imprecision: 1) our drawing procedure only corrected the "gap" problem, but ignored the "overlapping" part; 2) the rounding errors of the data on the bounding box may build up when the extender glyph is to be drawn several times. Although current devices are using 26.6 fixed number system (32 bits with 6 bits used for precision), and the pixels can be measured in float units, there still exits rounding errors. The more repetition of patching parts, the more building up of rounding errors, the more chances of lose precision.

AfffineTransform, on the other hand, can improve the precision of the glyph by scaling the extender instead of patching the extender one by one. If an extender needs to be redrawn10 times, with affine transformation, you only need to enlarge the y-coordinator by 10: *AffineTransform.scale (1.0, 10.0).* What affine transformation does is to convert device independent coordinator system to device's coordinator system. Thus by supplying a scale factor of (1.0, count), where count is the number of times an extender needs to be drawn, the extender is extended to the desired length. Therefore, when integrating my code into the big project, Anton Dragunov used *AffineTransform*.scale() to extend the extender instead of repeated drawing. The following figure shows the improvement of rendering when affine transformation is used.[2]



a) Stretching is done by repetition of the extender glyph. Notice the inappropriate alignment of the center part of the left brace and a cut-off of its bottom part.



b) Stretching is done by stretching the extender glyph using affine transformations of the shape. The alignment and the size of the brace is precise.

Fig 5: comparison of using "assembling parts" and "affine transformation" to draw stretchy symbols

Contributions

My major contributions include the following:

1) Provided a *MathFont* class (733 lines of code) that can

   1a) read in and store 5 font encoding tables in an array of hash tables, which are keyed by a Unicode and store the associate glyph index,

   1b) find the proper fonts to call and glyphs to use when given a Unicode, and

   1c) add the glyph codes for each stretchy symbol in a vector in proper order, and provide API to return that vector when a Unicode is passed.

2) Provided a *NormalSymbol* class (400 lines) that can draw it self at specified size or scale to specified size.

3) Provided a *StretchySymbol* class (566 lines) that can stretch a math symbol to specified width, or height, or both.

4) Provided a Constant interface (51 lines) that defines all the constants used in the project.

5) Other java classes for testing purpose (*Symbol.java* (39 lines), *SymbolFrame.java* (170 lines), *DrawPanel.java* (98 lines), *MathGlyph.java* (28 lines), *MathSymbol.java* (42 lines))

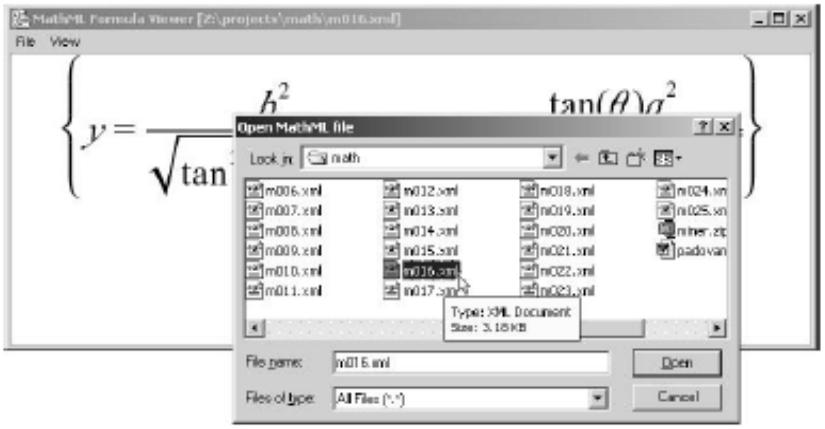
6) Help improve the understanding in the following area:

6a) Unicode, character encoding, font encoding, glyph, and their relations

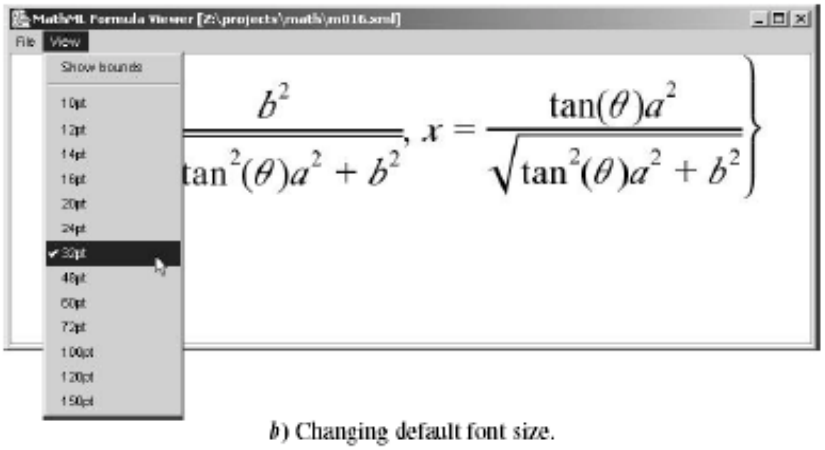
6b) Java font approaches, supported encodings and converters, the *font.properties* file handling, text rendering process

6c) Typography: bitmap fonts vs. outline fonts; outline fonts scaling (rendering), font metrics, font files of TrueType fonts, anti-aliasing, and hinting

The first 4 java classes have been integrated into our Java MathML Rendering Tool. Following is the snapshot of the output of this software:[2]



a) Loading a MathML file for viewing.



b) Changing default font size.

Fig 6: snap shot of the math expression drawn using our MathML rendering tool with my code integrated

Part IV: Future Work

Create a converter for each font file, and write a script to change the *font.propeties* file and to add that converter to the classpath of Java runtime. Glyph vector related methods are too complicated for drawing normal symbols; When drawString() is enough to do the job, we should make it possible to use the easy method.

Appendix 1: Apple's requirements and recommendations for Character Code to Glyph Mapping

Character Code/Glyph Mapping Requirements and Recommendations[26]

The following is a list of required characteristics for glyphs and their corresponding character codes in Apple TrueType fonts.

1. The first glyph (glyph index 0) *must be* the MISSING CHARACTER GLYPH. This glyph must have a visible appearance and non-zero advance width.
2. The second glyph (glyph index 1) *must be* the NULL glyph. This glyph must have no contours and zero advance width.
3. All characters in the appropriate Macintosh character set(s) *must* be present in fonts to be used with proper Macintosh applications. Certain specified characters, however, are mapped to the

MISSING CHARACTER GLYPH as stated below. This is not true for Unicode; there is no requirement for Unicode coverage except that it *should* at least be the complete set of Unicode equivalents of characters in at least one Macintosh character set.

4. If a font has multiple cmaps, then corresponding characters in the respective character sets *should be* mapped to the same glyphs.
5. The following character codes must be mapped to the first glyph (glyph index 0, MISSING CHARACTER GLYPH):

    3. 0x0001 START OF HEADING
    4. 0x0002 START OF TEXT
    5. 0x0003 END OF TEXT
    6. 0x0004 END OF TRANSMISSION
    7. 0x0005 ENQUIRY
    8. 0x0006 ACKNOWLEDGE
    9. 0x0007 BELL
    10. 0x000A LINE FEED
    11. 0x000B VERTICAL TABULATION
    12. 0x000C FORM FEED
    13. 0x000E SHIFT OUT
    14. 0x000F SHIFT IN
    15. 0x0010 DATA LINK ESCAPE
    16. 0x0011 DEVICE CONTROL 1 (X-On)
    17. 0x0012 DEVICE CONTROL 2
    18. 0x0013 DEVICE CONTROL 3 (X-Off)
    19. 0x0014 DEVICE CONTROL 4
    20. 0x0015 NEGATIVE ACKNOWLEDGE
    21. 0x0016 SYNCHRONOUS IDLE
    22. 0x0017 END OF TRANSMISSION BLOCK
    23. 0x0018 CANCEL
    24. 0x0019 END OF MEDIUM
    25. 0x001A SUBSTITUTE
    26. 0x001B ESCAPE
    27. 0x001C FILE SEPARATOR
    28. 0x001E RECORD SEPARATOR
    29. 0x001F UNIT SEPARATOR
    30. 0x007F DELETE

6. The following characters must map to the second glyph (glyph index 1, the NULL glyph):

    31. 0x0000 NULL
    32. 0x0008 BACKSPACE
    33. 0x000D CARRIAGE RETURN (in a right-to-left font)
    34. 0x001D GROUP SEPARATOR

7. Each of the following characters must map to a glyph with no contours and positive advance width:

    35. 0x0009 HORIZONTAL TABULATION
    36. 0x000D CARRIAGE RETURN (in a left-to-right font)
    37. 0x0020 SPACE
    38. 0x00A0 NO-BREAK SPACE

8. The following groups of characters must have the same width

39. 0x0009 (HORIZONTAL TABULATION) and 0x0020 (SPACE)

The following list of recommendations are for the glyphs mapped to the character codes stated. They are intended to aid in line layout

1. Lining numbers should be monospaced. These include 0x0030 through 0x0039 (digits zero through nine) and 0x00CA (Unicode U+2007) FIGURE SPACE. Old-style figures need not be monospaced.
2. In cursive fonts, glyphs should overlap to allow glyphs to join together in device-independent text. a 5% overlap is recommended.
3. White space should be evenly distributed between the left- and right-side bearings of glyphs. Extra space should be placed on the right if grid-fitting results in an odd number of pixels.

Appendix 2: Font Encoding Table for font “Math1”[25]

0x20 0x0020 #Space
0x21 0x0021 #Exclamation mark
0x22 0x2200 #For all
0x23 0x0023 #Number sign
0x24 0x2203 #There exists
0x25 0x0025 #Percent sign
0x26 0x0026 #Ampersand
0x27 0x220D #Small contains as member
0x28 0x0028 #Left parenthesis
0x29 0x0029 #Right parenthesis
0x2A 0x002A #Asterisk
0x2B 0x002B #Plus sign
0x2C 0x002C #Comma
0x2D 0x002D #Hyphen-minus
0x2E 0x002E #Full stop
0x2F 0x002F #Forward slash (solidus)

0x30 0x0030 #Digit zero
0x31 0x0031 #Digit one
0x32 0x0032 #Digit two
0x33 0x0033 #Digit three
0x34 0x0034 #Digit four
0x35 0x0035 #Digit five
0x36 0x0036 #Digit six
0x37 0x0037 #Digit seven
0x38 0x0038 #Digit eight
0x39 0x0039 #Digit nine
0x3A 0x003A #Colon
0x3B 0x003B #Semicolon
0x3C 0x003C #Less-than sign
0x3D 0x003D #Equals sign
0x3E 0x003E #Greater-than sign
0x3F 0x003F #Question mark

0x40 0x2245 #Approximately equal to
0x41 0x0391 #Greek capital letter Alpha
0x42 0x0392 #Greek capital letter Beta
0x43 0x03A7 #Greek capital letter Chi
0x44 0x0394 #Greek capital letter Delta
0x45 0x0395 #Greek capital letter Epsilon
0x46 0x03A6 #Greek capital letter Phi
0x47 0x0393 #Greek capital letter Gamma
0x48 0x0397 #Greek capital letter Eta
0x49 0x0399 #Greek capital letter Iota
0x4A 0x03D1 #Greek theta symbol
0x4B 0x039A #Greek capital letter Kappa
0x4C 0x039B #Greek capital letter Lamda
0x4D 0x039C #Greek capital letter Mu
0x4E 0x039D #Greek capital letter Nu
0x4F 0x039F #Greek capital letter Omicron

0x50 0x03A0 #Greek capital letter Pi
0x51 0x0398 #Greek capital letter Theta

0x52 0x03A1 #Greek capital letter Rho
0x53 0x03A3 #Greek capital letter Sigma
0x54 0x03A4 #Greek capital letter Tau
0x55 0x03A5 #Greek capital letter Upsilon
0x56 0x03C2 #Greek small letter final sigma
0x57 0x03A9 #Greek capital letter Omega
0x58 0x039E #Greek capital letter Xi
0x59 0x03A8 #Greek capital letter Psi
0x5A 0x0396 #Greek capital letter Zeta
0x5B 0x005B #Left square bracket
0x5C 0x2234 #Therefore
0x5D 0x005D #Right square bracket
0x5E 0x22A5 #Perpendicular
0x5F 0x005F #Low line

0x60 0x005E #Circumflex accent
0x61 0x03B1 #Greek small letter alpha
0x62 0x03B2 #Greek small letter beta
0x63 0x03C7 #Greek small letter chi
0x64 0x03B4 #Greek small letter delta
0x65 0x03B5 #Greek small letter epsilon
0x66 0x03C6 #Greek small letter phi
0x67 0x03B3 #Greek small letter gamma
0x68 0x03B7 #Greek small letter eta
0x69 0x03B9 #Greek small letter iota
0x6A 0x03D5 #Greek phi symbol
0x6B 0x03BA #Greek small letter kappa
0x6C 0x03BB #Greek small letter lamda
0x6D 0x03BC #Greek small letter mu
0x6E 0x03BD #Greek small letter nu
0x6F 0x03BF #Greek small letter omicron

0x70 0x03C0 #Greek small letter pi
0x71 0x03B8 #Greek small letter theta
0x72 0x03C1 #Greek small letter rho
0x73 0x03C3 #Greek small letter sigma
0x74 0x03C4 #Greek small letter tau
0x75 0x03C5 #Greek small letter upsilon
0x76 0x03D6 #Greek pi symbol
0x77 0x03C9 #Greek small letter omega
0x78 0x03BE #Greek small letter xi
0x79 0x03C8 #Greek small letter psi
0x7A 0x03B6 #Greek small letter zeta
0x7B 0x007B #Left curly bracket
0x7C 0x007C #Vertical line
0x7D 0x007D #Right curly bracket
0x7E 0x007E #Tilde
0x7F ?nbsp

0x80 0x2010 #Hyphen
0x81 ?lost   #Alias delimiter [STIX: &AliasDelimiter;]
0x82 0x25A0 #Black square
0x83 0x25A1 #White square
0x84 0x22F1 #Down right diagonal ellipsis
0x85 0x2043 #Hyphen bullet
0x86 0x226A #Much less-than

```
 0x87  0x226B  #Much greater-than
 0x88  0x2423  #Space indicator
-0x89  0x00D7  #Multiplication sign - see 0xB4
 0x8A  0x2A75  #TWO CONSECUTIVE EQUALS SIGNS, &Equal;
 0x8B  0x0060  #Grave accent
 0x8C  0x0027  #Apostrophe
 0x8D  0x0022  #Quotation mark
 0x8E  0x0303  #Tilde embellishment
 0x8F  0x0331  #Macron below embellishment

 0x90  0x21D5  #Up down double arrow
 0x91  0x21D5:G, 0x21D1:G, 0x21D3:G  #double vertical arrow extender
 0x92 >PUA     # &DoubleLongLeftArrow; &Longleftarrow; &xlArr;
 0x93  0x21D0:G, 0x21D2:G, 0x21D4:G  #double horizontal arrow extender
 0x94 >PUA     # &DoubleLongRightArrow; &Longrightarrow; &xrArr;
-0x95  0x21D4  #double left right arrow - ?same as 0xDB
 0x96 >PUA     # &DoubleLongLeftRightArrow; &Longleftrightarrow; &xhArr;
 0x97  0x2195  #Up down arrow
 0x98 >PUA     # &LongLeftArrow; &longleftarrow;  &xlarr;
 0x99 >PUA     # &LongRightArrow; &longrightarrow; &xrarr;
 0x9A >PUA     # &LongLeftRightArrow; &longleftrightarrow; &xharr;
 0x9B  0x2254  #COLON EQUALS, colone, coloneq, Assign
 0x9C  0x22C4  #DIAMOND OPERATOR, diam, diamond, Diamond
 0x9D  0x2287  #Superset of or equal to
 0x9E  0x0040  #Commercial at
 0x9F  0x25AA  #Black small square

-0xA0  0x002E  #Full stop - ?same as 0x2E
 0xA1  0x03D2  #Greek upsilon with hook symbol
 0xA2  0x2032  #Prime
 0xA3  0x2264  #Less-than or equal to
-0xA4  0x002F  #Slash, Solidus - slightly more slanted than 0x2F
 0xA5  0x221E  #Infinity
 0xA6  0x29EB  #Rule delayed
 0xA7  0x2663  #Black club suit
 0xA8  0x2662  #White diamond suit
 0xA9  0x2661  #White heart suit
 0xAA  0x2660  #Black spade suit
 0xAB  0x2194  #Left right arrow
 0xAC  0x2190  #Leftwards arrow
 0xAD  0x2191  #Upwards arrow
 0xAE  0x2192  #Rightwards arrow
 0xAF  0x2193  #Downwards arrow

 0xB0  0x00B0  #Degree sign
 0xB1  0x00B1  #Plus-minus sign
 0xB2  0x2033  #Double prime
 0xB3  0x2265  #Greater-than or equal to
 0xB4  0x00D7  #Multiplication sign - sligthly bigger than 0x89 - seems better on the web/screen
 0xB5  0x221D  #Proportional to
 0xB6  0x2202  #Partial differential
 0xB7  0x2022  #Bullet
 0xB8  0x00F7  #Division sign
 0xB9  0x2260  #Not equal to
 0xBA  0x2261  #Identical to
 0xBB  0x2248  #Almost equal to
```

0xBC 0x2026 #Horizontal ellipsis
0xBD 0x2191:G, 0x2193:G, 0x2195:G #Vertical arrow extender
0xBE 0x2190:G, 0x2192:G, 0x2194:G #Horizontal arrow extender
0xBF 0x21B5 #Downwards arrow with corner leftwards

0xC0 0x2135 #Alef symbol
0xC1 0x2111 #Fraktur capital I
0xC2 0x211C #Fraktur capital R
0xC3 0x2118 #Weierstrass elliptic symbol
0xC4 0x2297 #Circled times
0xC5 0x2295 #Circled plus
0xC6 0x2205 #Empty set
0xC7 0x22C3 #N-ary union
0xC8 0x22C2 #N-ary intersection
0xC9 0x2283 #Superset of
0xCA ?nbsp
0xCB 0x2284 #Not a subset of
0xCC 0x2282 #Subset of
0xCD 0x2286 #Subset of or equal to
0xCE 0x2208 #Element of
0xCF 0x2209 #Not an element of

0xD0 0x2220 #Angle
0xD1 0x2207 #Gradient (nabla)
0xD2 0x00AE #Registered sign
0xD3 0x00A9 #Copyright sign
0xD4 0x2122 #Trade mark sign
0xD5 0x220F #N-ary product
0xD6 0x221A #Radical symbol
0xD7 0x00B7 #Middle dot
0xD8 0x00AC #Not sign
0xD9 0x2227 #Logical and
0xDA 0x2228 #Logical or
0xDB 0x21D4 #double left right arrow
0xDC 0x21D0 #double left arrow
0xDD 0x21D1 #double up arrow
0xDE 0x21D2 #double right arrow
0xDF 0x21D3 #double down arrow

0xE0 0x25CA #LOZENGE, loz, lozenge
0xE1 0x2329 #left-pointing angle bracket
0xE2 0x2146 #DOUBLE-STRUCK ITALIC SMALL D, DifferentialD, dd
0xE3 0x2147 #DOUBLE-STRUCK ITALIC SMALL E, ExponentialE, exponentiale, ee
0xE4 0x2148 #DOUBLE-STRUCK ITALIC SMALL I, ImaginaryI, ii
0xE5 0x2211 #N-ary summation
0xE6 0x0028:T #left parenthesis top
0xE7 0x0028:G #left parenthesis extender
0xE8 0x0028:B #left parenthesis bottom
0xE9 0x005B:T #left square bracket top
0xEA 0x005B:G #left square bracket extender
0xEB 0x005B:B #left square bracket bottom
0xEC 0x007B:T #left curly bracket top
0xED 0x007B:M #left curly bracket middle
0xEE 0x007B:B #left curly bracket bottom
0xEF 0x007B:G, 0x007D:G #curly bracket extender

0xF0 0x2317 #VIEWDATA SQUARE - not to confuse with 0x22D5 equal and parallel to
0xF1 0x232A #right-pointing angle bracket
0xF2 0x222B #Integral
0xF3 0x222B:B
0xF4 0x222B:G
0xF5 0x222B:T
0xF6 0x0029:T #right parenthesis top
0xF7 0x0029:G #right parenthesis extender
0xF8 0x0029:B #right parenthesis bottom
0xF9 0x005D:T #right square bracket top
0xFA 0x005D:G #right square bracket extender
0xFB 0x005D:B #right square bracket bottom
0xFC 0x007D:T #right curly bracket top
0xFD 0x007D:M #right curly bracket middle
0xFE 0x007D:B #right curly bracket bottom
0xFF ? #?Unknown character

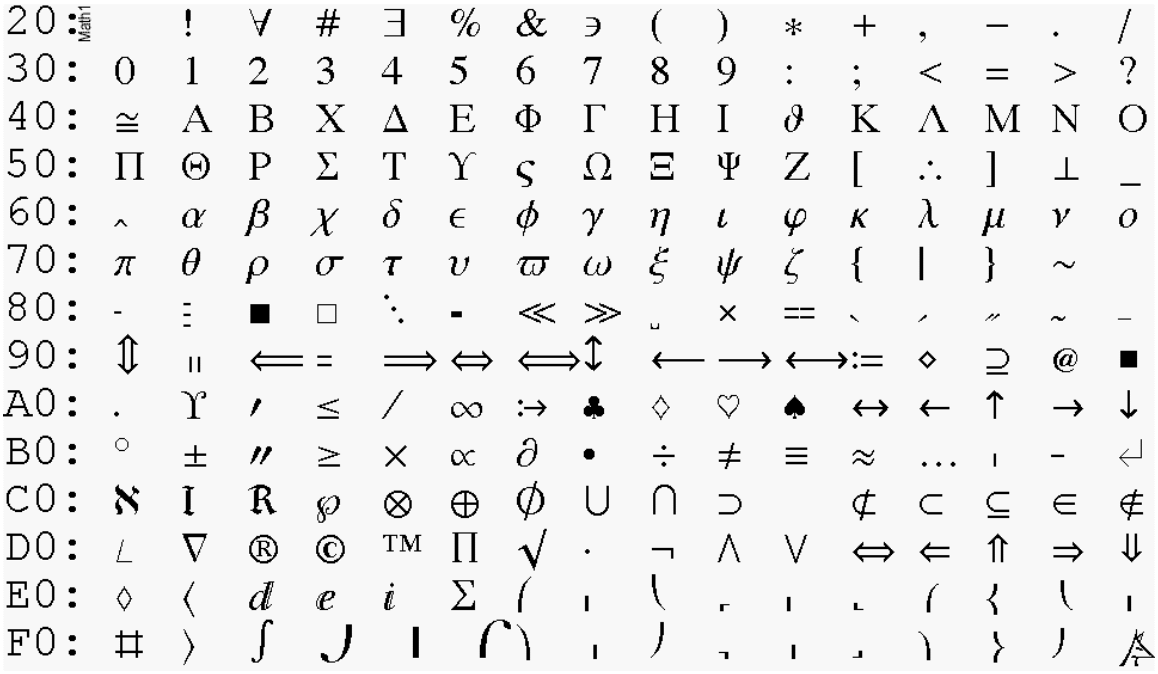


Fig 7: Font Encoding Table for Font “Math1”[32]

Appendix 3: Font Encoding Table for font “Math2”[33]

```
 0x20  0x0020    #Space
0x21  0x221A:?  #Square root, radic, Sqrt
0x22  0x221A:?  #Square root, radic, Sqrt
0x23  0x221A:?  #Square root, radic, Sqrt
0x24  0x221A:?  #Square root, radic, Sqrt
0x25  0x221A:?  #Square root, radic, Sqrt
0x26  0x221A:?  #Square root, radic, Sqrt
0x27  0x221A:?  #Square root, radic, Sqrt
0x28  0x221A:?  #Square root, radic, Sqrt
0x29  0x221A:?  #Square root, radic, Sqrt
0x2A  0x221A:?  #Square root, radic, Sqrt
0x2B  0x221A:?  #Square root, radic, Sqrt
0x2C  0x221A:?  #Square root, radic, Sqrt
0x2D  0x221A:?  #Square root, radic, Sqrt
0x2E  0x221A:?  #Square root, radic, Sqrt
0x2F  0x221A:?  #Square root, radic, Sqrt

0x30  0x221A:0  #Square root, radic, Sqrt
0x31  0x221A:1  #Square root, radic, Sqrt
0x32  0x221A:2  #Square root, radic, Sqrt
0x33  0x221A:3  #Square root, radic, Sqrt
0x34  0x221A:B  #Square root, radic, Sqrt
0x35  0x221A:G  #Square root, radic, Sqrt
0x36  0x221A:T  #Square root, radic, Sqrt
0x37  0x221A:?  #Square root, radic, Sqrt
0x38  0x007B:0  #Left curly bracket
0x39  0x007B:1  #Left curly bracket
0x3A  0x007B:2  #Left curly bracket
0x3B  0x007B:3  #Left curly bracket
0x3C  0x007D:0  #Right curly bracket
0x3D  0x007D:1  #Right curly bracket
0x3E  0x007D:2  #Right curly bracket
0x3F  0x007D:3  #Right curly bracket

0x40  0x005B:0  #Left square bracket
0x41  0x005B:1  #Left square bracket
0x42  0x005B:2  #Left square bracket
0x43  0x005B:3  #Left square bracket
0x44  0x005D:0  #Right square bracket
0x45  0x005D:1  #Right square bracket
0x46  0x005D:2  #Right square bracket
0x47  0x005D:3  #Right square bracket
0x48  0x0028:0  #Left parenthesis
0x49  0x0028:1  #Left parenthesis
0x4A  0x0028:2  #Left parenthesis
0x4B  0x0028:3  #Left parenthesis
0x4C  0x0029:0  #Right parenthesis
0x4D  0x0029:1  #Right parenthesis
0x4E  0x0029:2  #Right parenthesis
0x4F  0x0029:3  #Right parenthesis

0x50  0x301A:0  #Left white square bracket
0x51  0x301A:1  #Left white square bracket
```

0x52 0x301A:2 #Left white square bracket
0x53 0x301A:3 #Left white square bracket
0x54 0x301B:0 #Right white square bracket
0x55 0x301B:1 #Right white square bracket
0x56 0x301B:2 #Right white square bracket
0x57 0x301B:3 #Right white square bracket
0x58 0x2329:0 #Left-pointing angle bracket
0x59 0x2329:1 #Left-pointing angle bracket
0x5A 0x2329:2 #Left-pointing angle bracket
0x5B 0x2329:3 #Left-pointing angle bracket
0x5C 0x232A:0 #Right-pointing angle bracket
0x5D 0x232A:1 #Right-pointing angle bracket
0x5E 0x232A:2 #Right-pointing angle bracket
0x5F 0x232A:3 #Right-pointing angle bracket

0x60 0x2308:0 #Left ceiling
0x61 0x2308:1 #Left ceiling
0x62 0x2308:2 #Left ceiling
0x63 0x2308:3 #Left ceiling [tuning: map this to: 0x2308:T]
0x64 0x230A:0 #Left floor
0x65 0x230A:1 #Left floor
0x66 0x230A:2 #Left floor
0x67 0x230A:3 #Left floor [tuning: map this to: 0x230A:B]
0x68 0x2308:G, 0x230A:G
0x69 0x0028:T #Left parenthesis
0x6A 0x0028:G #Left parenthesis
0x6B 0x0028:B #Left parenthesis
0x6C 0x007B:T #Left curly bracket
0x6D 0x007B:M #Left curly bracket
0x6E 0x007B:B #Left curly bracket
0x6F 0x007B:G, 0x007D:G #curly bracket extender

0x70 0x2309:0 #Right ceiling
0x71 0x2309:1 #Right ceiling
0x72 0x2309:2 #Right ceiling
0x73 0x2309:3 #Right ceiling [tuning: map this to: 0x2309:T]
0x74 0x230B:0 #Right floor
0x75 0x230B:1 #Right floor
0x76 0x230B:2 #Right floor
0x77 0x230B:3 #Right floor [tuning: map this to: 0x230B:B]
0x78 0x2309:G, 0x230B:G
0x79 0x0029:T #Right parenthesis
0x7A 0x0029:G #Right parenthesis
0x7B 0x0029:B #Right parenthesis
0x7C 0x007D:T #Right curly bracket
0x7D 0x007D:M #Right curly bracket
0x7E 0x007D:B #Right curly bracket
0x7F ?nbsp

0x80 0x005B:T #Left square bracket
0x81 0x005B:G #Left square bracket
0x82 0x005B:B #Left square bracket
0x83 0x005D:T #Right square bracket
0x84 0x005D:G #Right square bracket
0x85 0x005D:B #Right square bracket
0x86 0x301A:B #Left white square bracket

0x87 0x301A:G #Left white square bracket
0x88 0x301A:T #Left white square bracket
0x89 0x301B:B #Right white square bracket
0x8A 0x301B:G #Right white square bracket
0x8B 0x301B:T #Right white square bracket
0x8C ? :? #?Left ceiling
0x8D ?
0x8E ? :? #?Right ceiling
0x8F 0x221A:? #Square root, radic, Sqrt

0x90 0x002F:0 #Forward slash (solidus)
0x91 0x002F:1 #Forward slash (solidus)
0x92 0x002F:2 #Forward slash (solidus)
0x93 0x002F:3 #Forward slash (solidus)
0x94 0x2216:0 #Back slash (Set minus)
0x95 0x2216:1 #Back slash (Set minus)
0x96 0x2216:2 #Back slash (Set minus)
0x97 0x2216:3 #Back slash (Set minus)
0x98 0x222E:0 #Contour integral
0x99 0x222F:0 #Surface integral
0x9A 0x2233:0 #Anticlockwise contour integral
0x9B 0x2232:0 #Clockwise contour integral (arrow on right)
0x9C 0x2A16:0 #?QUATERNION INTEGRAL OPERATOR, quatin, [#Integral with square]
0x9D 0x228E:0 #Multiset union
0x9E 0x2293:0 #Square cap
0x9F 0x2294:0 #Square cup

0xA0 0x2223:0 #Left vertical bar
0xA1 0x2223:1 #Left vertical bar
0xA2 0x2223:2 #Left vertical bar
0xA3 0x2223:3 #Left vertical bar
0xA4 ? :? #Right vertical bar
0xA5 ? :? #Right vertical bar
0xA6 ? :? #Right vertical bar
0xA7 ? :? #Right vertical bar
0xA8 0x222E:1 #Contour integral
0xA9 0x222F:1 #Surface integral
0xAA 0x2233:1 #Anticlockwise contour integral
0xAB 0x2232:1 #Clockwise contour integral (arrow on right)
0xAC 0x2A16:1 #?QUATERNION INTEGRAL OPERATOR, quatin, [#Integral with square]
0xAD 0x228E:1 #Multiset union
0xAE 0x2293:1 #Square cap
0xAF 0x2294:1 #Square cup

0xB0 ? :? #Left double vertical bar
0xB1 ? :? #Left double vertical bar
0xB2 ? :? #Left double vertical bar
0xB3 ? :? #Left double vertical bar
0xB4 ? :? #Right double vertical bar
0xB5 ? :? #Right double vertical bar
0xB6 ? :? #Right double vertical bar
0xB7 ? :? #Right double vertical bar
0xB8 0x222E:2 #Contour integral
0xB9 0x222F:2 #Surface integral
0xBA 0x2233:2 #Anticlockwise contour integral
0xBB 0x2232:2 #Clockwise contour integral (arrow on right)

0xBC 0x2A16:2 #?QUATERNION INTEGRAL OPERATOR, quatin, [#Integral with square]
0xBD 0x228E:2 #Multiset union
0xBE 0x2293:2 #Square cap
0xBF 0x2294:2 #Square cup

0xC0 0x2225:0 #?left double vertical bar
0xC1 0x2225:1 #?left double vertical bar
0xC2 0x2225:2 #?left double vertical bar
0xC3 0x2225:3 #?left double vertical bar
0xC4 0x2223:G #?left double vertical bar extender
0xC5 0x2225:G #?left vertical bar extender
0xC6 0x02DA #RING ABOVE, ring
0xC7 0x02C7 #caron, Hacek
0xC8 0x2758:0 #?LIGHT VERTICAL BAR, ❘
0xC9 0x2758:1 #?LIGHT VERTICAL BAR, ❘
0xCA ?nbsp
0xCB 0x2758:2 #?LIGHT VERTICAL BAR, ❘
0xCC 0x2758:3 #?LIGHT VERTICAL BAR, ❘
0xCD 0x2A0D:0 #Finite part integral, fpartint, Integral with slash
0xCE 0x2A0D:1 #Finite part integral, fpartint, Integral with slash
0xCF 0x2A0D:2 #Finite part integral, fpartint, Integral with slash

0xD0 0x0308 #combining diaeresis [#Two dots above]
0xD1 0x20DB #Three dots above embellishment
0xD2 0x21BC, 0x21BC:L, 0x294E:L #Left harpoon (barb up)
0xD3 0x21C0, 0x21C0:R, 0x294E:R #Right harpoon (barb up)
0xD4 0x294E #Left barb up right barb up harpoon, LeftRightVector
0xD5 0x20D6, 0x20D6:L, 0x20E1:L #Left arrow embellishment
0xD6 0x20D6:G, 0x20D7:G, 0x20E1:G, 0x21BC:G, 0x21C0:G, 0x294E:G #Horizontal embellishment extender
0xD7 0x20D7, 0x20D7:R, 0x20E1:R #Right arrow embellishment
0xD8 0x20E1 #Left right arrow embellishment
0xD9 0x222B:0 #Integral
0xDA 0x2211:0 #N-ary summation
0xDB 0x220F:0 #N-ary product
0xDC 0x22C3:0 #N-ary union
0xDD 0x22C2:0 #N-ary intersection
0xDE 0x22C1:0 #N-ary logical or [?0x2228:1 #Logical or]
0xDF 0x22C0:0 #N-ary logical and [?0x2227:1 #Logical and]

0xE0 0x222B:1 #Integral
0xE1 0x222B:2 #Integral
0xE2 0x2211:1 #N-ary summation
0xE3 0x2211:2 #N-ary summation
0xE4 0x220F:1 #N-ary product
0xE5 0x220F:2 #N-ary product
0xE6 0x22C3:1 #N-ary union
0xE7 0x22C3:2 #N-ary union
0xE8 0x22C2:1 #N-ary intersection
0xE9 0x22C2:2 #N-ary intersection
0xEA 0x22C1:1 #N-ary logical or
0xEB 0x22C1:2 #N-ary logical or
0xEC 0x22C0:1 #N-ary logical and
0xED 0x22C0:2 #N-ary logical and
0xEE 0x2228:0 #Logical or
0xEF 0x2227:0 #Logical and

0xF0 0x030F #Double grave accent
0xF1 0x02D8 #Breve
0xF2 0x0311 #Combining inverted breve, DownBreve
0xF3 0x02DC:0 #Small tilde, tilde, DiacriticalTilde
0xF4 0x02DC:1 #Small tilde, tilde, DiacriticalTilde
0xF5 0x02DC:2 #Small tilde, tilde, DiacriticalTilde
0xF6 0x02DC:3 #Small tilde, tilde, DiacriticalTilde
0xF7 0x2032 #Prime
0xF8 0x2033 #Double prime
0xF9 0x2035 #Reversed prime
0xFA 0x2036 #Reversed double prime
0xFB ? :? #?Vertical bar
0xFC ? :? #?Double vertical bar
0xFD ? :? #?Vertical bar
0xFE ? :? #?Double vertical bar
0xFF 0x2210:0 #N-ary coproduct

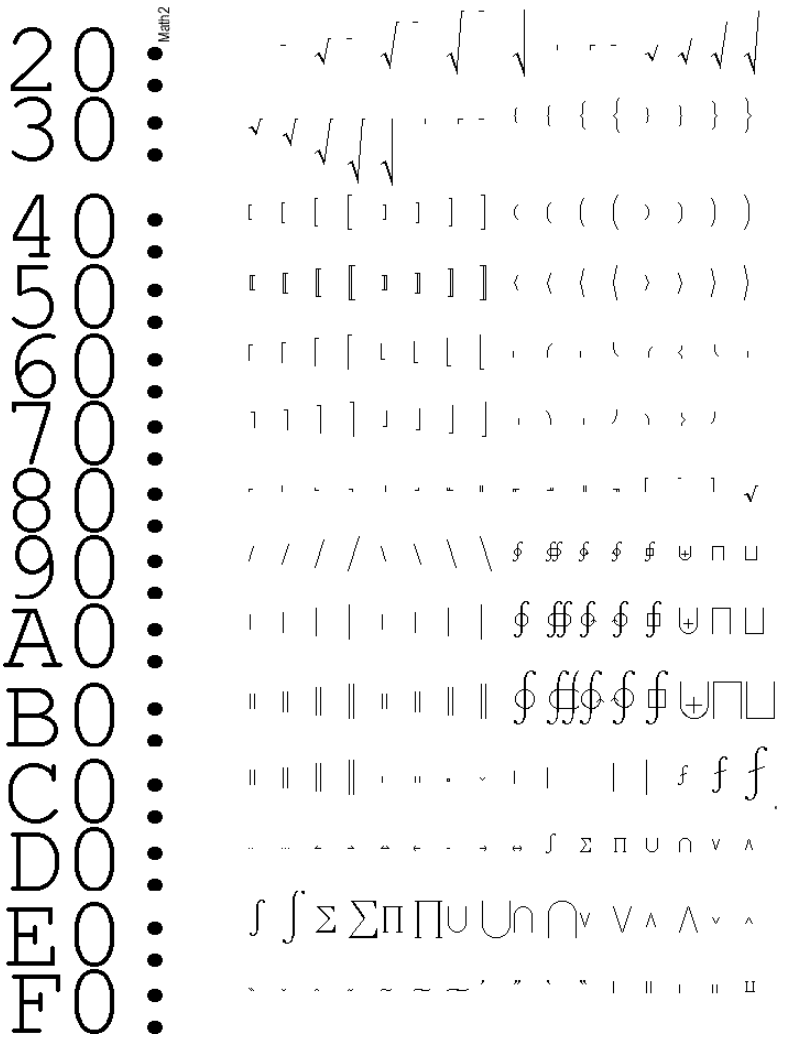


Fig 8: Font Encoding Table for Font “Math 2”[34]

Appendix 4: Font Encoding Table for font "Math3"[35]

```
 0x20  0x0020  #Space
 0x21  0x222B  #Integral
 0x22  0x222E  #Contour integral
 0x23  0x222F  #Surface integral
 0x24  0x2233  #Anticlockwise contour integral
 0x25  0x2232  #Clockwise contour integral (arrow on left)
 0x26  0x2A16  #?QUATERNION INTEGRAL OPERATOR, &quatin;
 0x27  0x2229  #Intersection
 0x28  0x222A  #Union
 0x29  0x228E  #Multiset union
 0x2A  0x2293  #Square cap
 0x2B  0x2294  #Square cup
-0x2C  0x2227  #big logical and
 0x2D  0x2227  #Logical and
-0x2E  0x2228  #big logical or
 0x2F  0x2228  #Logical or

 0x30  0x22B2  #Normal subgroup of
 0x31  0x22B4  #Normal subgroup of or equal to
-0x32 >PUA     #Normal subgroup of with bar
 0x33  0x22EA  #Not normal subgroup of
 0x34  0x22EC  #Not normal subgroup of or equal to, &nltrie; &ntrianglelefteq; &NotLeftTriangleEqual;
-0x35 >PUA     #Not normal subgroup of with bar
-0x36 >PUA     #Bar operator [#?2223 Divides]
-0x37 >PUA     #Double bar operator [?#2225 Parallel To]
-0x38 >PUA     #Triple bar operator
-0x39 >PUA     #Bar operator [#?2223 Divides]
-0x3A >PUA     #Double bar operator [?#2225 Parallel To]
-0x3B ?vertical single bar extender
 0x3C >PUA     #0x2225-0xFE00x, PARALLEL, SLANTED, &parsl;
 0x3D  0x2247  #Neither approximately nor actually equal to
 0x3E  0x2243  #Asymptotically equal to
 0x3F  0x2242  #Minus tilde

 0x40  0x22B3  #Contains as normal subgroup
 0x41  0x22B5  #Contains as normal subgroup or equal to
-0x42 >PUA     #Contains as normal subgroup with bar
 0x43  0x22EB  #Does not contain as normal subgroup
 0x44  0x22ED  #Does not contain as normal subgroup or equal
-0x45 >PUA     #Does not contain as normal subgroup with bar
-0x46 >PUA     #Not bar operator
-0x47 >PUA     #Not double bar operator
-0x48 >PUA     #Not triple bar operator
-0x49 >PUA     #Not bar operator
-0x4A >PUA     #Not double bar operator
-0x4B ?vertical double bar extender
 0x4C  0x2241  #Not tilde
 0x4D  0x2249  #Not almost equal to
 0x4E  0x2244  #Not asymptotically equal to
 0x4F >PUA     #0x2242-0x0338, NOT EQUAL OR SIMILAR, &NotEqualTilde; &nesim;

 0x50  0x224F  #Difference between
 0x51  0x224E  #Geometrically equivalent to
```

```
 0x52 >PUA     #0x224F-0x0338, NOT BUMPY SINGLE EQUALS, &nbumpe; &NotHumpEqual;
 0x53 >PUA     #0x224E-0x0338, NOT BUMPY EQUALS, &NotHumpDownHump; &nbump;
 0x54  0x2262  #Not identical to
 0x55  0x2250  #Approaches the limit
 0x56  0x2970  #RIGHT DOUBLE ARROW WITH ROUNDED HEAD, &RoundImplies;
 0x57  0x003D  #Equals sign
-0x58 >PUA     #Triple equal
-0x59 >PUA     #Frown over bar
-0x5A >PUA     #Smile under bar
 0x5B  0x2322  #Frown
 0x5C  0x2323  #Smile
 0x5D ?
 0x5E  0x224D  #Equivalent to
 0x5F  0x226D  #Not equivalent to

 0x60  0x226A  #Much less-than
 0x61  0x2AA1  #DOUBLE NESTED LESS-THAN, &LessLess;
 0x62  0x2264  #LESS-THAN OR EQUAL TO, ≤ &leq;
 0x63  0x2266  #Less-than over equal to
 0x64  0x2272  #Less-than or equivalent to
 0x65  0x226E  #Not less-than
 0x66 >PUA     #U0226A-00338-0FE00, NOT MUCH LESS THAN, VARIANT, &nLtv; &NotLessLess;
 0x67 >PUA     #U024A1-00338, NOT DOUBLE LESS-THAN SIGN, &NotNestedLessLess;
 0x68  0x2270  #Neither less-than nor equal to
-0x69  0x2270  #Neither less-than nor equal to
 0x6A  0x2268  #Less-than but not equal to
 0x6B  0x2274  #Neither less-than nor equivalent to
 0x6C  0x228F  #Square image of
 0x6D  0x2291  #Square image of or equal to
 0x6E >PAU     #U0228F-00338, NOT, SQUARE SUBSET, &NotSquareSubset;
 0x6F  0x22E2  #Not square image of or equal to

 0x70  0x226B  #Much greater-than
 0x71  0x2AA2  #DOUBLE NESTED GREATER-THAN, &GreaterGreater;
 0x72  0x2265  #GREATER-THAN OR EQUAL TO, ≥ &GreaterEqual; &geq;
 0x73  0x2267  #Greater-than over equal to
 0x74  0x2273  #Greater-than or equivalent to
 0x75  0x226F  #Not greater-than
 0x76 >PUA     #U0226B-00338-0FE00 NOT MUCH GREATER THAN, VARIANT, &nGtv;
&NotGreaterGreater;
 0x77 >PUA     #U024A2-00338, NOT DOUBLE GREATER-THAN SIGN, &NotNestedGreaterGreater;
 0x78  0x2271  #Neither greater-than nor equal to
-0x79  0x2271  #Neither greater-than nor equal to
 0x7A  0x2269  #Greater-than but not equal to
 0x7B  0x2275  #Neither greater-than nor equivalent to
 0x7C  0x2290  #Square original of
 0x7D  0x2292  #Square original of or equal to
 0x7E >PAU     #U02290-00338 NOT SQUARE SUPERSET, &NotSquareSuperset;
 0x7F ?nbsp

 0x80  0x227A  #Precedes
-0x81  0x227C  #Precedes or equal to
 0x82  0x227C  #Precedes or equal to
 0x83  0x227E  #Precedes or equivalent to
 0x84  0x2280  #Does not precede
```

-0x85 0x22E0 #Does not precede or equal
 0x86 0x22E0 #Does not precede or equal
 0x87 0x22E8 #Precedes but not equivalent to
 0x88 0x2276 #Less-than or greater-than
 0x89 0x2278 #Neither less-than nor greater-than
 0x8A 0x22DA #Less-than equal to or greater-than
 0x8B 0x2235 #Because or since
 0x8C 0x22E3 #Not square original of or equal to
 0x8D 0x2288 #Neither a subset of nor equal to
 0x8E 0x2285 #Not a superset of
 0x8F 0x2289 #Neither a superset of nor equal to

 0x90 0x227B #Succeeds
-0x91 0x227D #Succeeds or equal to
 0x92 0x227D #Succeeds or equal to
 0x93 0x227F #Succeeds or equivalent to
 0x94 0x2281 #Does not succeed
-0x95 0x22E1 #Does not succeed or equal
 0x96 0x22E1 #Does not succeed or equal
 0x97 0x22E9 #Succeeds but not equivalent to
 0x98 0x2277 #Greater-than or less-than
 0x99 0x2279 #Neither greater-than nor less-than
 0x9A 0x22DB #Greater-than equal to or less-than
 0x9B 0x2237 #Proportion
 0x9C 0x220B #Contains as member
 0x9D 0x220C #Does not contain as member
 0x9E 0x2296 #Circled minus
 0x9F 0x2299 #Circled dot operator

 0xA0 0x22D5 #EQUAL AND PARALLEL TO, ⋕
 0xA1 0x2213 #Minus-plus sign
 0xA2 0x22A2 #Right tack
 0xA3 0x22A8 #True
 0xA4 0x22A3 #Left tack
 0xA5 0x2AE4 #VERTICAL BAR DOUBLE LEFT TURNSTILE, ⫤ ⫤
 0xA6 0x22A5 #Perpendicular
 0xA7 0x22A4 #Down tack
 0xA8 0x22BA #intercalate [could also be mapped to 0x2351 #APL FUNCTION SYMBOL UP TACK
OVERBAR]
 0xA9 0x226C #BETWEEN, twixt, between
 0xAA 0x2240 #Wreath product
 0xAB 0x22C8 #Bowtie (join, relational database theory)
 0xAC 0x221F #Right angle
 0xAD 0x2221 #Measured angle
 0xAE 0x2222 #Spherical angle
 0xAF 0x2205 #Empty set

 0xB0 0x2127 #Inverted ohm sign
 0xB1 0x2204 #There does not exist
 0xB2 0x03DC #Greek letter digamma
 0xB3 0x03DA #Greek letter stigma
 0xB4 ?
 0xB5 0x03E0 #Greek letter sampi
 0xB6 0x03B5 #Greek small letter epsilon
 0xB7 0x03F1 #Greek rho symbol

```
 0xB8  0xED03  #Greek small letter digamma
 0xB9  0x03C2  #Greek small letter final sigma
 0xBA ?
 0xBC  0x2136  #Bet symbol
 0xBD  0x2137  #Gimel symbol
 0xBE  0x2138  #Dalet symbol
 0xBF  0x03F0  #Greek kappa symbol

 0xC0  0x00DD  #Latin capital letter Y with acute
 0xC1  0x00FD  #Latin small letter y with acute
 0xC2  0x00D0  #Latin capital letter Eth
 0xC3  0x00F0  #Latin small letter eth
 0xC4  0x00DE  #Latin capital letter Thorn
 0xC5  0x00FE  #Latin small letter thorn
 0xC6  0x0141  #Latin capital letter L with stroke
 0xC7  0x0142  #Latin small letter l with stroke
 0xC8  0x00BF  #Inverted question mark
 0xC9  0x00A2  #Cent sign
 0xCA  0x212B  #Angstrom sign
 0xCB  0x00A3  #Pound sign
 0xCC  0x00A1  #Inverted exclamation mark
 0xCD  0x00A7  #Section sign
 0xCE  0x00B6  #Paragraph sign (pilcrow)
 0xCF  0x00A5  #Yen sign

 0xD0  0x00B5  #Micro sign
 0xD1  0x210F  #Planck constant over two pi
 0xD2  0x0131  #imath *normal style* - Latin small letter dotless i
-0xD3  0x0131  #imath italic - Latin small letter dotless i
 0xD4 >PUA     #jmath *normal style* - Latin small letter dotless j, &jmath;
-0xD5 >PUA     #jmath italic - Latin small letter dotless j, &jmath;
 0xD6  0x2020  #Dagger
 0xD7  0x2021  #Double dagger
 0xD8  0x00B0  #Degre sign
 0xD9  0x266D  #Music flat sign
 0xDA  0x266E  #Music natural sign
 0xDB  0x266F  #Music sharp sign
 0xDC  0x2370  #Boxed question mark
 0xDD  0x2353  #Boxed up caret
 0xDE  0x00C5  #Latin capital letter A with ring above
 0xDF  0x019B  #Latin small letter lambda with stroke

 0xE0  0x25A0  #Black square
 0xE1  0x25A1  #White square
 0xE2  0x25FE  #BLACK MEDIUM SMALL SQUARE, FilledSmallSquare
 0xE3  0x25FD  #WHITE MEDIUM SMALL SQUARE, EmptySmallSquare
 0xE4  0x25AA  #Black small square
 0xE5  0x25AB  #White small square
-0xE6 >PUA     #Large black circle
 0xE7  0x25EF  #Large white circle, xcirc, bigcirc
 0xE7  0x25CF  #Black circle
 0xE8  0x25CB  #White circle
 0xE9  0x2219  #Bullet
 0xEA  0x25E6  #Composition
 0xEB ?        #Large black diamond
```

0xEC ? #Large white diamond
0xED 0x25C6 #Black diamond
0xEF 0x25C7 #White diamond

0xF0 0x25AE #Black vertical rectangle
0xF1 0x25AF #White vertical rectangle
0xF2 0x25B2 #Black up-pointing triangle
0xF3 0x25B3 #White up-pointing triangle
0xF4 0x25BC #Black down-pointing triangle
0xF5 0x25BD #White down-pointing triangle
0xF6 0x29EB #FILLED LOZENGE, lozf, blacklozenge
0xF7 0x2736 #Six pointed black star
0xF8 0x2605 #Black star
0xF9 ?
0xFA ?
0xFB 0x0190 #Latin capital letter open E
-0xFC >PUA #DOUBLE-STRUCK ITALIC Imaginary j
-0xFD >PUA #DOUBLE-STRUCK ITALIC Blackboard-bold greek small letter gamma
-0xFE >PUA #DOUBLE-STRUCK ITALIC Blackboard-bold greek small letter pi
0xFF 0x2145 #DOUBLE-STRUCK ITALIC CAPITAL D, CapitalDifferentialD, DD

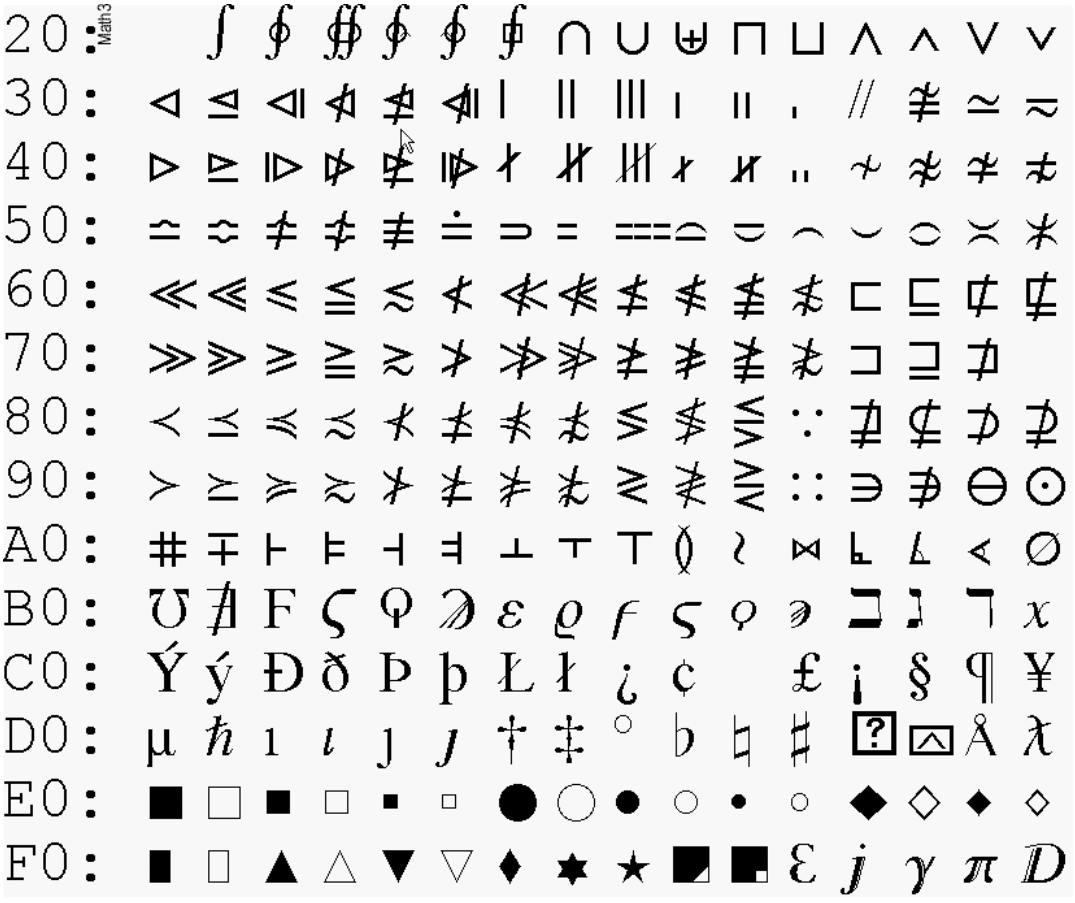


Fig 9: Font Encoding Table for Font “Math3”[36]

Appendix 5: Font Encoding Table for font “Math4”[37]

0x20 0x0020 #Space
0x21 0x21E5 #Rightwards arrow to bar
0x22 0x21A4 #Leftwards arrow from bar
0x23 0x21A6 #Rightwards arrow from bar
0x24 0x2912 #UPWARDS ARROW TO BAR, UpArrowBar
0x25 0x2913 #DOWNWARDS ARROW TO BAR, DownArrowBar
0x26 0x21A5 #Upwards arrow from bar
0x27 0x21A7 #Downwards arrow from bar
0x28 0x2952 #LEFTWARDS HARPOON WITH BARB UP TO BAR, LeftVectorBar
0x29 0x2953 #RIGHTWARDS HARPOON WITH BARB UP TO BAR , RightVectorBar
0x2A 0x2956 #LEFTWARDS HARPOON WITH BARB DOWN TO BAR, DownLeftVectorBar
0x2B 0x2957 #RIGHTWARDS HARPOON WITH BARB DOWN TO BAR, DownRightVectorBar
0x2C 0x295A #LEFTWARDS HARPOON WITH BARB UP FROM BAR, LeftTeeVector
0x2D 0x295B #RIGHTWARDS HARPOON WITH BARB UP FROM BAR, RightTeeVector
0x2E 0x295E #LEFTWARDS HARPOON WITH BARB DOWN FROM BAR, DownLeftTeeVector
0x2F 0x295F #RIGHTWARDS HARPOON WITH BARB DOWN FROM BAR, DownRightTeeVector

0x30 0x21BC #Left harpoon (barb up)
0x31 0x21C0 #Right harpoon (barb up)
0x32 0x294E #LEFT BARB UP RIGHT BARB UP HARPOON, LeftRightVector
0x33 0x21BD #Left harpoon (barb down)
0x34 0x21C1 #Right harpoon (barb down)
0x35 0x2950 #LEFT BARB DOWN RIGHT BARB DOWN HARPOON, DownLeftRightVector
0x36 0x2951 #UP BARB LEFT DOWN BARB LEFT HARPOON, LeftUpDownVector
0x37 0x294F #UP BARB RIGHT DOWN BARB RIGHT HARPOON, RightUpDownVector
0x38 0x2958 #UPWARDS HARPOON WITH BARB LEFT TO BAR, LeftUpVectorBar
0x39 0x2959 #DOWNWARDS HARPOON WITH BARB LEFT TO BAR, LeftDownVectorBar
0x3A 0x2954 #UPWARDS HARPOON WITH BARB RIGHT TO BAR , RightUpVectorBar
0x3B 0x2955 #DOWNWARDS HARPOON WITH BARB RIGHT TO BAR, RightDownVectorBar
0x3C 0x2960 #UPWARDS HARPOON WITH BARB LEFT FROM BAR, LeftUpTeeVector
0x3D 0x2961 #DOWNWARDS HARPOON WITH BARB LEFT FROM BAR, LeftDownTeeVector
0x3E 0x295C #UPWARDS HARPOON WITH BARB RIGHT FROM BAR, RightUpTeeVector
0x3F 0x295D #DOWNWARDS HARPOON WITH BARB RIGHT FROM BAR, RightDownTeeVector

0x40 0x21BF, 0x2960:T #Up harpoon (barb left)
0x41 0x2191:G, 0x2193:G
0x42 0x21C3, 0x2961:B #Down harpoon (barb left)
-0x43 0x21BE #Up harpoon (barb right) with lbearing = 0 - see also 0x55
-0x44 #vertical extender for arrows with barb right and with lbearing = 0
-0x45 0x21C2 #Down harpoon (barb right) with lbearing = 0 - see also 0x64
0x46 0x21CC #Right harpoon over left harpoon
0x47 0x21CB #Left harpoon over right harpoon
0x48 0x21CC:L
0x49 0x21CC:R
0x4A 0x21CB:L
0x4B 0x21CB:R
0x4C 0x296E #UPWARDS HARPOON WITH BARB LEFT BESIDE DOWNWARDS HARPOON WITH BARB RIGHT, udhar, UpEquilibrium

0x4D 0x296F #DOWNWARDS HARPOON WITH BARB LEFT BESIDE UPWARDS HARPOON WITH BARB RIGHT, duhar, ReverseUpEquilibrium
0x4E 0x21A9 #Leftwards arrow with hook
0x4F 0x21AA #Rightwards arrow with hook

0x50 0x2190, 0x2190:L #LEFTWARDS ARROW, larr, leftarrow, LeftArrow
0x51 0x2190:G, 0x2192:G, 0x2194:G, 0x21A4:G, 0x21A6:G
0x52 0x21A4:R
0x53 0x21A6:L
0x54 0x2192 #RIGHTWARDS ARROW, rarr, rightarrow, RightArrow
0x55 0x21BE #see also 0x43 #UPWARDS HARPOON WITH BARB RIGHTWARDS , uharr, upharpoonright, RightUpVector
0x56 0x21C4 #Rightwards arrow over leftwards arrow
0x57 0x21C6 #Leftwards arrow over rightwards arrow
0x58 0x21C4:L
0x59 0x21C4:R
0x5A 0x21C6:R
0x5B 0x21C6:L
0x5C 0x21C5 #Upwards arrow leftwards of downwards arrow
0x5D 0x21F5 #DOWNWARDS ARROW LEFTWARDS OF UPWARDS ARROW, duarr, DownArrowUpArrow
0x5E ?single line vertical extender
0x5F ?single line horizontal extender

0x60 0x2191, 0x2191:T, 0x21A5:T, 0x295C:T, 0x2960:T #Upwards arrow
0x61 0x21A5:B
0x62 0x21A7:T
0x63 0x2193, 0x2193:B, 0x21A7:B #Downwards arrow
0x64 0x21C2 #see also 0x45 #DOWNWARDS HARPOON WITH BARB RIGHTWARDS, dharr, RightDownVector, downharpoonright
0x65 0x296E:B
0x66 0x296E:G, 0x296F:G # double vertical extender for arrows with heads
0x67 0x296F:T
0x68 0x296E:T
0x69 0x21C5:G, 0x21F5:G #vertical extender for arrows with barb left - MFPF: arrows with bars, \u2951.G, \u21BF.G, \u21C3.G ...
0x6A 0x296F:B
0x6B 0x21C5:B
0x6C 0x21F5:T
0x6D 0x21C5:T
0x6E 0x21F5:B
0x6F 0x21CC:G, 0x21CB:G, 0x21C4:G, 0x21C6:G #double horizontal extender of arrows with barb up and down

0x70 0x23B4:0 #TOP SQUARE BRACKET, tbrk, OverBracket
0x71 0x23B4:1
0x72 0x23B4:2
0x73 0x23B4:3
0x74 0x23B4:L
0x75 0x23B4:G
0x76 0x23B4:R
0x77 >PUA #[Upwards arrowhead] SHORT UP ARROW 2303-FE00, ↑
0x78 >PUA #[Downwards arrowhead] SHORT DOWN ARROW 2304-FE00, ↓
0x79 >PUA #[Leftwards arrowhead] SHORT LEFT ARROW 2190-0FE00, ←
0x7A >PUA #[Rightwards arrowhead] SHORT RIGHT ARROW 2192-FE00, →
0x7B 0x21E4 #Leftwards arrow to bar

```
0x7C  0x2012  #EN DASH
0x7D  0x2013  #EM DASH
0x7E  0x2014  #HORIZONTAL BAR = QUOTATION DASH
0x7F ?nbsp

0x80  0x23B5:0  #BOTTOM SQUARE BRACKET, bbrk, UnderBracket
0x81  0x23B5:1
0x82  0x23B5:2
0x83  0x23B5:3
0x84  0x23B5:L
0x85  0x23B5:G
0x86  0x23B5:R
0x87  0x2199:0  #South west arrow
0x88  0x2196:0  #North west arrow
0x89  0x2197:0  #North east arrow
0x8A  0x2198:0  #South east arrow
0x8B  0x2199:1  #South west arrow
0x8C  0x2196:1  #North west arrow
0x8D  0x2197:1  #North east arrow
0x8E  0x2198:1  #South east arrow
0x8F  0x231E  # downward left corner, &dlcorn; \llcorner

0x90  0xFE35:0  #PRESENTATION FORM FOR VERTICAL LEFT PARENTHESIS, OverParenthesis
0x91  0xFE35:1
0x92  0xFE35:2
0x93  0xFE35:3
0x94  0xFE35:L
0x95  0xFE35:G
0x96  0xFE35:R
0x97  0xFE37:0  #PRESENTATION FORM FOR VERTICAL LEFT CURLY BRACKET, OverBrace
0x98  0xFE37:1
0x99  0xFE37:2
0x9A  0xFE37:3
0x9B  0xFE37:L
0x9C  0xFE37:G
0x9D  0xFE37:M
0x9E  0xFE37:R
0x9F  0x2035  #backprime - reverse prime, &bprime;

0xA0  0xFE36:0  #PRESENTATION FORM FOR VERTICAL RIGHT PARENTHESIS,
UnderParenthesis
0xA1  0xFE36:1
0xA2  0xFE36:2
0xA3  0xFE36:3
0xA4  0xFE36:L
0xA5  0xFE36:G
0xA6  0xFE36:R
0xA7  0xFE38:0  #PRESENTATION FORM FOR VERTICAL RIGHT CURLY BRACKET, UnderBrace
0xA8  0xFE38:1
0xA9  0xFE38:2
0xAA  0xFE38:3
0xAB  0xFE38:L
0xAC  0xFE38:G
0xAD  0xFE38:M
0xAE  0xFE38:R
0xAF  0x2032  #prime or minute, ′
```

```
 0xB0  0x0302:0  # hat
 0xB1  0x0302:1
 0xB2  0x0302:2
 0xB3  0x0302:3
 0xB4  0x2026  #Horizontal ellipsis
 0xB5  0x22EF  #Math-axis ellipsis
 0xB6  0x22EE  #Vertical ellipsis
 0xB7  0x22F0  #Up right diagonal ellipsis
 0xB8  0x22F1  #Down right diagonal ellipsis
-0xB9  0x2026  #same as 0xB4 #Horizontal ellipsis
-0xBA  0x22EF  #same as 0xB5 #Math-axis ellipsis
-0xBB  0x22EE  #same as 0xB6 #Vertical ellipsis
 0xBC  0x0307  #Dot
 0xBD  0x00B7  #\cdotp
-0xBE  0x2035  #Reversed prime
-0xBF  0x2036  #Reversed double prime

 0xC0  0x02C7:0  #caron, Hacek
 0xC1  0x02C7:1
 0xC2  0x02C7:2
 0xC3  0x02C7:3
 0xC4  0x25B4  #BLACK UP-POINTING SMALL TRIANGLE, utrif, blacktriangle
 0xC5  0x25B8  #BLACK RIGHT-POINTING SMALL TRIANGLE, rtrif, blacktriangleright
 0xC6  0x25BE  #BLACK DOWN-POINTING SMALL TRIANGLE, dtrif, blacktriangledown
 0xC7  0x25C2  #BLACK LEFT-POINTING SMALL TRIANGLE, ltrif, blacktriangleleft
 0xC8  0x21A9:R  #hook (side:up) for left arrow #see also 0xC7 hook (side:down)?
 0xC9  0x21AA:L  #hook (side:up) for right arrow #see also 0xC6 hook (side:down)?
 0xCA ?non-breaking space
 0xCB  0x22A0  #Squared times
 0xCC  0x263A  #Smily
 0xCD ?       #Smily
 0xCE  0x2639  #Smily
 0xCF ?       #Smily

 0xD0 ?
-0xD1  0x002F:0  #Forward slash (solidus)
-0xD2  0x002F:1  #These are much smaller than those in Math2
-0xD3  0x002F:2
-0xD4  0x002F:3
-0xD5  0x002F:4
 0xD6  0x25AA   #BLACK SMALL SQUARE, squf, squarf, blacksquare
 0xD7 ?      :R  #see also 0xC6 #hook (side:down) for left arrow? left harpoon?
 0xD8 ?      :L  #see also 0xC7 #hook (side:down) for right arrow? right harpoon?
 0xD9  Not/Applicable here
 0xDA  N/A
 0xDB  N/A
 0xDC  N/A
 0xDD  0x231A  #WATCH
 0xDE  0x0327  #COMBINING CEDILLA
 0xDF ?

 0xE0  Not/Applicable here
 0xE1  N/A
 0xE2  N/A
 0xE3  N/A
```

0xE4 N/A
0xE5 N/A
0xE6 N/A
0xE7 N/A
0xE8 N/A
0xE9 N/A
0xEA N/A
0xEB N/A
0xEC N/A
0xED 0x2318 #PLACE OF INTEREST = COMMAND KEY
0xEE N/A
0xEF 0x2423 #Space indicator

0xF0 N/A
0xF1 N/A
0xF2 N/A
0xF3 N/A
0xF4 N/A
0xF5 N/A
0xF6 N/A
0xF7 N/A
0xF8 N/A
0xF9 N/A
0xFA N/A
0xFB N/A
0xFC ? :B #hook for up arrow? up harpoon?
0xFD ? :B #hook for up arrow? up harpoon?
0xFE ? :T #hook for down arrow? down harpoon?
0xFF ? :T #hook for down arrow? down harpoon?

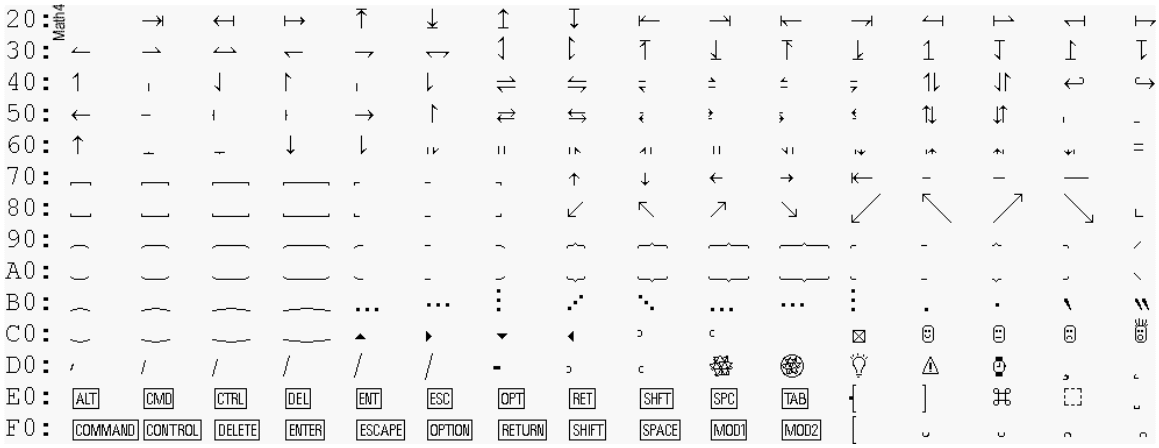


Fig 10: Font Encoding Table for Font “Math4”[38]

Appendix 6: Font Encoding Table for font “Math5”[39]

0x20 0x0020 #Space
0x21 Not/Applicable here
0x22 N/A
0x23 N/A
0x24 N/A
0x25 N/A
0x26 ?nbsp
0x27 ?nbsp
0x28 ?nbsp
0x29 ?nbsp
0x2A ?nbsp
0x2B ?nbsp
0x2C ?nbsp
0x2D ?nbsp
0x2E ?nbsp
0x2F ?nbsp

0x30 ?
0x31 ?nbsp
0x32 ?nbsp
0x33 ?nbsp
0x34 ?nbsp
0x35 ?nbsp
0x36 ?nbsp
0x37 ?nbsp
0x38 ?nbsp
0x39 ?nbsp
0x3A ?nbsp
0x3B ?nbsp
0x3C ?nbsp
0x3D ?nbsp
0x3E ?nbsp
0x3F ?nbsp

0x40 ?nbsp
0x41 0x1D49C #MATHEMATICAL SCRIPT CAPITAL A, 𝒜
0x42 0x212C #Script capital B
0x43 0x1D49E #MATHEMATICAL SCRIPT CAPITAL C, 𝒞
0x44 0x1D49F #MATHEMATICAL SCRIPT CAPITAL D, 𝒟
0x45 0x2130 #Script capital E
0x46 0x2131 #Script capital F
0x47 0x1D4A2 #MATHEMATICAL SCRIPT CAPITAL G, 𝒢
0x48 0x210B #Script capital H
0x49 0x2110 #Script capital I
0x4A 0x1D4A5 #MATHEMATICAL SCRIPT CAPITAL J, 𝒥
0x4B 0x1D4A6 #MATHEMATICAL SCRIPT CAPITAL K, 𝒦
0x4C 0x2112 #Script capital L
0x4D 0x2133 #Script capital M
0x4E 0x1D4A9 #MATHEMATICAL SCRIPT CAPITAL N, 𝒩
0x4F 0x1D4AA #MATHEMATICAL SCRIPT CAPITAL O, 𝒪

0x50 0x1D4AB #MATHEMATICAL SCRIPT CAPITAL P, 𝒫
0x51 0x1D4AC #MATHEMATICAL SCRIPT CAPITAL Q, 𝒬

```
0x52  0x211B  #Script capital R
0x53  0x1D4AE  #MATHEMATICAL SCRIPT CAPITAL S, &Sscr;
0x54  0x1D4AF  #MATHEMATICAL SCRIPT CAPITAL T, &Tscr;
0x55  0x1D4B0  #MATHEMATICAL SCRIPT CAPITAL U, &Uscr;
0x56  0x1D4B1  #MATHEMATICAL SCRIPT CAPITAL V, &Vscr;
0x57  0x1D4B2  #MATHEMATICAL SCRIPT CAPITAL W, &Wscr;
0x58  0x1D4B3  #MATHEMATICAL SCRIPT CAPITAL X, &Xscr;
0x59  0x1D4B4  #MATHEMATICAL SCRIPT CAPITAL Y, &Yscr;
0x5A  0x1D4B5  #MATHEMATICAL SCRIPT CAPITAL Z, &Zscr;
0x5B ?nbsp
0x5C ?nbsp
0x5D ?nbsp
0x5E ?nbsp
0x5F ?nbsp

0x60 ?nbsp
0x61  0x1D4B6  #MATHEMATICAL SCRIPT SMALL a, &ascr;
0x62  0x1D4B7  #MATHEMATICAL SCRIPT SMALL b, &bscr;
0x63  0x1D4B8  #MATHEMATICAL SCRIPT SMALL c, &cscr;
0x64  0x1D4B9  #MATHEMATICAL SCRIPT SMALL d, &dscr;
0x65  0x212F  #Script small e
0x66  0x1D4BB  #MATHEMATICAL SCRIPT SMALL f, &fscr;
0x67  0x210A  #Script small g
0x68  0x1D4BD  #MATHEMATICAL SCRIPT SMALL h, &hscr;
0x69  0x1D4BE  #MATHEMATICAL SCRIPT SMALL i, &iscr;
0x6A  0x1D4BF  #MATHEMATICAL SCRIPT SMALL j, &jscr;
0x6B  0x1D4C0  #MATHEMATICAL SCRIPT SMALL k, &kscr;
-0x6C  0x2113  #Script small l - 1-like - see also 0x7B
0x6D  0x1D4C2  #MATHEMATICAL SCRIPT SMALL m, &mscr;
0x6E  0x1D4C3  #MATHEMATICAL SCRIPT SMALL n, &nscr;
0x6F  0x2134  #Script small o

0x70  0x1D4C5  #MATHEMATICAL SCRIPT SMALL p, &pscr;
0x71  0x1D4C6  #MATHEMATICAL SCRIPT SMALL q, &qscr;
0x72  0x1D4C7  #MATHEMATICAL SCRIPT SMALL r, &rscr;
0x73  0x1D4C8  #MATHEMATICAL SCRIPT SMALL s, &sscr;
0x74  0x1D4C9  #MATHEMATICAL SCRIPT SMALL t, &tscr;
0x75  0x1D4CA  #MATHEMATICAL SCRIPT SMALL u, &uscr;
0x76  0x1D4CB  #MATHEMATICAL SCRIPT SMALL v, &vscr;
0x77  0x1D4CC  #MATHEMATICAL SCRIPT SMALL w, &wscr;
0x78  0x1D4CD  #MATHEMATICAL SCRIPT SMALL x, &xscr;
0x79  0x1D4CE  #MATHEMATICAL SCRIPT SMALL y, &yscr;
0x7A  0x1D4CF  #MATHEMATICAL SCRIPT SMALL z, &zscr;
0x7B  0x2113  #Script small l - \ell-like - see also 0x6C
0x7C ?nbsp
0x7D ?nbsp
0x7E ?nbsp
0x7F ?nbsp

0x80 ?nbsp
0x81 ?nbsp
0x82  0x212D  #Fraktur capital C
0x83  0x1D507  #MATHEMATICAL FRAKTUR CAPITAL D, &Dfr;
0x84  0x1D508  #MATHEMATICAL FRAKTUR CAPITAL E, &Efr;
0x85  0x1D509  #MATHEMATICAL FRAKTUR CAPITAL F, &Ffr;
0x86  0x1D50A  #MATHEMATICAL FRAKTUR CAPITAL G, &Gfr;
```

0x87 0x210C #Fraktur capital H
0x88 0x2111 #Fraktur capital I
0x89 0x1D50D #MATHEMATICAL FRAKTUR CAPITAL J, 𝔍
0x8A 0x1D50E #MATHEMATICAL FRAKTUR CAPITAL K, 𝔎
0x8B 0x1D50F #MATHEMATICAL FRAKTUR CAPITAL L, 𝔏
0x8C 0x1D510 #MATHEMATICAL FRAKTUR CAPITAL M, 𝔐
0x8D ?nbsp
0x8E ?nbsp
0x8F ?nbsp

0x90 ?nbsp
0x91 0x211C #Fraktur capital R
0x92 0x1D516 #MATHEMATICAL FRAKTUR CAPITAL S, 𝔖
0x93 0x1D517 #MATHEMATICAL FRAKTUR CAPITAL T, 𝔗
0x94 0x1D518 #MATHEMATICAL FRAKTUR CAPITAL U, 𝔘
0x95 0x1D519 #MATHEMATICAL FRAKTUR CAPITAL V, 𝔙
0x96 0x1D51A #MATHEMATICAL FRAKTUR CAPITAL W, 𝔚
0x97 0x1D51B #MATHEMATICAL FRAKTUR CAPITAL X, 𝔛
0x98 0x1D51C #MATHEMATICAL FRAKTUR CAPITAL Y, 𝔜
0x99 0x2128 #Fraktur capital Z
0x9A 0x1D51E #MATHEMATICAL FRAKTUR SMALL a, 𝔞
0x9B 0x1D51F #MATHEMATICAL FRAKTUR SMALL b, 𝔟
0x9C 0x1D520 #MATHEMATICAL FRAKTUR SMALL c, 𝔠
0x9D ?nbsp
0x9E ?nbsp
0x9F 0x1D523 #MATHEMATICAL FRAKTUR SMALL f, 𝔣

0xA0 0x1D524 #MATHEMATICAL FRAKTUR SMALL g, 𝔤
0xA1 0x1D525 #MATHEMATICAL FRAKTUR SMALL h, 𝔥
0xA2 0x1D526 #MATHEMATICAL FRAKTUR SMALL i, 𝔦
0xA3 0x1D527 #MATHEMATICAL FRAKTUR SMALL j, 𝔧
0xA4 0x1D528 #MATHEMATICAL FRAKTUR SMALL k, 𝔨
0xA5 0x1D529 #MATHEMATICAL FRAKTUR SMALL l, 𝔩
0xA6 0x1D52A #MATHEMATICAL FRAKTUR SMALL m, 𝔪
0xA7 0x1D52B #MATHEMATICAL FRAKTUR SMALL n, 𝔫
0xA8 0x1D52C #MATHEMATICAL FRAKTUR SMALL o, 𝔬
0xA9 0x1D52D #MATHEMATICAL FRAKTUR SMALL p, 𝔭
0xAA 0x1D52E #MATHEMATICAL FRAKTUR SMALL q, 𝔮
0xAB 0x1D52F #MATHEMATICAL FRAKTUR SMALL r, 𝔯
0xAC 0x1D530 #MATHEMATICAL FRAKTUR SMALL s, 𝔰
0xAD 0x1D531 #MATHEMATICAL FRAKTUR SMALL t, 𝔱
0xAE 0x1D532 #MATHEMATICAL FRAKTUR SMALL u, 𝔲
0xAF 0x1D533 #MATHEMATICAL FRAKTUR SMALL v, 𝔳

0xB0 0x1D534 #MATHEMATICAL FRAKTUR SMALL W, 𝔴
0xB1 0x1D535 #MATHEMATICAL FRAKTUR SMALL X, 𝔵
0xB2 0x1D536 #MATHEMATICAL FRAKTUR SMALL Y, 𝔶
0xB3 0x1D537 #MATHEMATICAL FRAKTUR SMALL Z, 𝔷
0xB4 0x1D504 #MATHEMATICAL FRAKTUR CAPITAL A, 𝔄
0xB5 0x1D505 #MATHEMATICAL FRAKTUR CAPITAL B, 𝔅
0xB6 0x1D511 #MATHEMATICAL FRAKTUR CAPITAL N, 𝔑
0xB7 0x1D512 #MATHEMATICAL FRAKTUR CAPITAL O, 𝔒
0xB8 0x1D513 #MATHEMATICAL FRAKTUR CAPITAL P, 𝔓
0xB9 0x1D514 #MATHEMATICAL FRAKTUR CAPITAL Q, 𝔔
0xBA 0x1D521 #MATHEMATICAL FRAKTUR SMALL d, 𝔡
0xBB 0x1D522 #MATHEMATICAL FRAKTUR SMALL e, 𝔢

0xBC 0x1D542 #MATHEMATICAL DOUBLE-STRUCK CAPITAL K, 𝕂
0xBD ?nbsp
0xBE ?nbsp
0xBF ?nbsp

0xC0 0x1D538 #MATHEMATICAL DOUBLE-STRUCK CAPITAL A, 𝔸
0xC1 0x1D539 #MATHEMATICAL DOUBLE-STRUCK CAPITAL B, 𝔹
0xC2 0x2102 #Blackboard-bold capital C
0xC3 0x1D53B #MATHEMATICAL DOUBLE-STRUCK CAPITAL D, 𝔻
0xC4 0x1D53C #MATHEMATICAL DOUBLE-STRUCK CAPITAL E, 𝔼
0xC5 0x1D53D #MATHEMATICAL DOUBLE-STRUCK CAPITAL F, 𝔽
0xC6 0x1D53E #MATHEMATICAL DOUBLE-STRUCK CAPITAL G, 𝔾
0xC7 0x210D #Blackboard-bold capital H
0xC8 0x1D540 #MATHEMATICAL DOUBLE-STRUCK CAPITAL I, 𝕀
0xC9 0x1D541 #MATHEMATICAL DOUBLE-STRUCK CAPITAL J, 𝕁
0xCA ?nbsp
0xCB 0x1D543 #MATHEMATICAL DOUBLE-STRUCK CAPITAL L, 𝕃 Ƶ
0xCC 0x1D544 #MATHEMATICAL DOUBLE-STRUCK CAPITAL M, 𝕄
0xCD 0x2115 #Blackboard-bold capital N - with double struck diagonal bar - see also 0xF4
0xCE 0x1D546 #MATHEMATICAL DOUBLE-STRUCK CAPITAL O, 𝕆
0xCF 0x2119 #Blackboard-bold capital P

0xD0 0x211A #Blackboard-bold capital Q
0xD1 0x211D #Blackboard-bold capital R
0xD2 0x1D54A #MATHEMATICAL DOUBLE-STRUCK CAPITAL S, 𝕊
0xD3 0x1D54B #MATHEMATICAL DOUBLE-STRUCK CAPITAL T, 𝕋
0xD4 0x1D54C #MATHEMATICAL DOUBLE-STRUCK CAPITAL U, 𝕌
0xD5 0x1D54D #MATHEMATICAL DOUBLE-STRUCK CAPITAL V, 𝕍
0xD6 0x1D54E #MATHEMATICAL DOUBLE-STRUCK CAPITAL W, 𝕎
0xD7 0x1D54F #MATHEMATICAL DOUBLE-STRUCK CAPITAL X, 𝕏
0xD8 0x1D550 #MATHEMATICAL DOUBLE-STRUCK CAPITAL Y, 𝕐
0xD9 0x2124 #Blackboard-bold capital Z
0xDA 0x1D552 #MATHEMATICAL DOUBLE-STRUCK SMALL a, 𝕒
0xDB 0x1D553 #MATHEMATICAL DOUBLE-STRUCK SMALL b, 𝕓
0xDC 0x1D554 #MATHEMATICAL DOUBLE-STRUCK SMALL c, 𝕔
0xDD 0x1D555 #MATHEMATICAL DOUBLE-STRUCK SMALL d, 𝕕
0xDE 0x1D556 #MATHEMATICAL DOUBLE-STRUCK SMALL e, 𝕖
0xDF 0x1D557 #MATHEMATICAL DOUBLE-STRUCK SMALL f, 𝕗

0xE0 0x1D558 #MATHEMATICAL DOUBLE-STRUCK SMALL g, 𝕘
0xE1 0x1D559 #MATHEMATICAL DOUBLE-STRUCK SMALL h, 𝕙
0xE2 0x1D55A #MATHEMATICAL DOUBLE-STRUCK SMALL i, 𝕚
0xE3 0x1D55B #MATHEMATICAL DOUBLE-STRUCK SMALL j, 𝕛
0xE4 0x1D55C #MATHEMATICAL DOUBLE-STRUCK SMALL k, 𝕜
0xE5 0x1D55D #MATHEMATICAL DOUBLE-STRUCK SMALL l, 𝕝
0xE6 0x1D55E #MATHEMATICAL DOUBLE-STRUCK SMALL m, 𝕞
0xE7 0x1D55F #MATHEMATICAL DOUBLE-STRUCK SMALL n, 𝕟
0xE8 0x1D560 #MATHEMATICAL DOUBLE-STRUCK SMALL o, 𝕠
0xE9 0x1D561 #MATHEMATICAL DOUBLE-STRUCK SMALL p, 𝕡
0xEA 0x1D562 #MATHEMATICAL DOUBLE-STRUCK SMALL q, 𝕢
0xEB 0x1D563 #MATHEMATICAL DOUBLE-STRUCK SMALL r, 𝕣
0xEC 0x1D564 #MATHEMATICAL DOUBLE-STRUCK SMALL s, 𝕤
0xED 0x1D565 #MATHEMATICAL DOUBLE-STRUCK SMALL t, 𝕥
0xEE 0x1D566 #MATHEMATICAL DOUBLE-STRUCK SMALL u, 𝕦
0xEF 0x1D567 #MATHEMATICAL DOUBLE-STRUCK SMALL v, 𝕧

0xF0 0x1D568 #MATHEMATICAL DOUBLE-STRUCK SMALL w, 𝕨
0xF1 0x1D569 #MATHEMATICAL DOUBLE-STRUCK SMALL x, 𝕩
0xF2 0x1D56A #MATHEMATICAL DOUBLE-STRUCK SMALL y, 𝕪
0xF3 0x1D56B #MATHEMATICAL DOUBLE-STRUCK SMALL z, 𝕫
-0xF4 0x2115 #Blackboard-bold capital N - with double struck left foot - see also 0xCD
0xF5 ?nbsp
0xF6 ?nbsp
0xF7 0x2019 #Right single quotation mark
0xF8 0x2018 #Left single quotation mark
0xF9 0x201D #Right double quotation mark
0xFA 0x201C #Left double quotation mark
0xFB 0x25CB #White circle
0xFC 0x25A1 #White square
0xFD 0x25B3 #White up-pointing triangle
0xFE N/A #Mathe
0xFF N/A #matica(R)

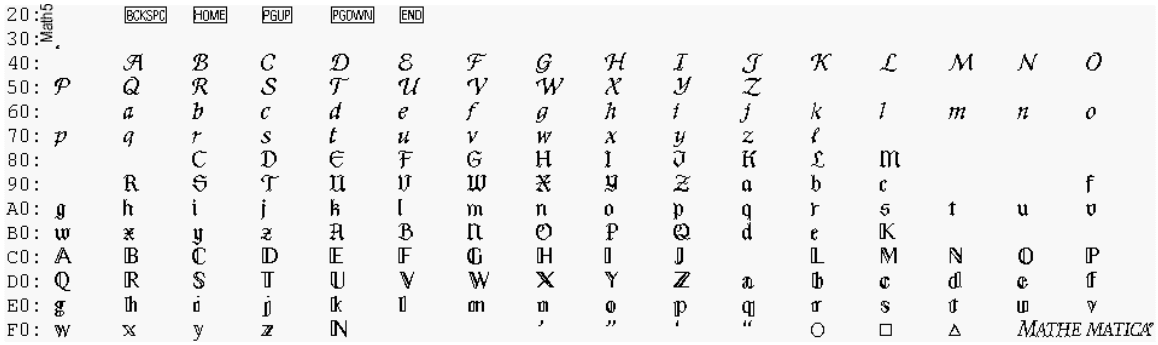


Fig 11: Font Encoding Table for Font “Math5”[40]

References

1. **http://www.w3.org/Math/.**
2. **Dragunov, Anton N. "Java Implementation of a MathML Rendering Engine", Master of Science Degree project report, Oregon State University, December 2002.**
3. **http://www.w3.org/Math/implementations.html.**
4. **http://www.dessci.com/en/products/webeq/webeq.asp.**
5. **http://www.stixfonts.org/faq.html#Eleven.**
6. **http://www.stixfonts.org/faq.html#Seven.**
7. **Barbow, Geoff and Eccles, Simmon, Typesetting & Composition, 2nd Edition, BluePrint, New York 1992, p9.**
8. **Barbow, Geoff and Eccles, Simmon, Typesetting & Composition, 2nd Edition, BluePrint, New York 1992, p2.**
9. **http://www.truetype.demon.co.uk/ttglossg.htm.**
10. **http://www.unicode.org/glossary.**
11. **Boualem, Malek, Leisher, Mark, and Ogden, Bill, "Encoding script-specific writing rules based on the Unicode character set", http://crl.nmsu.edu/~mleisher/contextnew.pdf.**
12. **www.unicode.com glossary.**
13. **http://www.w3.org/TR/REC-CSS2/fonts.html#encoding.**
14. **http://www.stixfonts.org/charactertable.html or http://www.unicode.org/charts/.**
15. **http://java.sun.com/j2se/1.4/docs/guide/intl/faq.html#Text Rendering.**

16. **Macgregor, Robert, Durbin, Dave, and Owlett, John, and Yeomans, Andrew, Java Network Security, Prentice Hall PTR, New Jersey, 1998, p.27.**
17. **http://java.sun.com/j2se/1.4/docs/guide/intl/fontprop.html#loading.**
18. **http://java.sun.com/j2se/1.3/docs/guide/intl/fontprop.html.**
19. **"Text Operations of Graphics2D Objects". http://java.sun.com/j2se/1.3/docs/api/java/awt/Graphics2D.html.**
20. **http://developer.apple.com/fonts/TTRefMan/RM06/Chap6.html#Overview.**
21. **Sullivan, David, Sullivan, J. Wesley, and Sullivan, William L., Desk Top Publishing: Writing and Publishing in the Computer Age, Houghtan Mifflin Company, 1989, p359.**
22. **True Type outline. http://www.truetype.demon.co.uk/ttoutln.htm.**
23. **http://developer.apple.com/fonts/TTRefMan/RM01/chap1.html.**
24. **http://developer.apple.com/fonts/TTRefMan/RM01/chap2.html.**
25. **http://www.microsoft.com/typography/default.asp click on link "font smoothing".**
26. **http://www.truetype.demon.co.uk/ttalias.htm.**
27. **http://www.mozilla.ort/project/font/encoding.**
28. **http://www.mozilla.org/projects/mathml/fonts/encoding/math1.html.**
29. **http://www.microsoft.com/typography/otspec/recom.htm.**
30. **http://developer.apple.com/fonts/TTRefMan/RM07/appendixB.html.**
31. **André, Jacques, "Font Metrics", in Hersch, Roger D., Visual and Techical Aspects of Type, Cambridge University Press, 1993.**
32. **http://www.mozilla.org/projects/mathml/fonts/encoding/math1.gif.**

33. **http://www.mozilla.org/projects/mathml/fonts/encoding/math2.html.**

34. **http://www.mozilla.org/projects/mathml/fonts/encoding/math2.gif.**

35. **http://www.mozilla.org/projects/mathml/fonts/encoding/math3.html.**

36. **http://www.mozilla.org/projects/mathml/fonts/encoding/math3.gif.**

37. **http://www.mozilla.org/projects/mathml/fonts/encoding/math4.html.**

38. **http://www.mozilla.org/projects/mathml/fonts/encoding/math4.gif.**

39. **http://www.mozilla.org/projects/mathml/fonts/encoding/math5.html.**

40. **http://www.mozilla.org/projects/mathml/fonts/encoding/math5.gif.**

41. **http://java.sun.com/j2se/1.4/docs/guide/intl/encoding.doc.html**