A Tutorial and Directory for My Gonzo PostScript Utilities

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

http://www.tinaja.com
don@tinaja.com
(928) 428-4073

Few people appreciate how versatile and how powerful the **PostScript** general purpose computing language is. Or how fast and conveniently and cheaply and intuitively it now can handle an amazing variety of tasks for you.

PostScript is especially superb for…

- Uniquely creating stunning world class graphics.
- Acrobat .PDF file sourcing, editing, and **post editing**.
- Reading or writing most any diskfile in most any language.
- Exploring common or exotic mathematical concepts.
- Robotic or **Santa Claus Machine** controllers and sequences.
- Programmatically writing sourcecode in other computer languages.
- Using "real math" to generate complex charts or graphs.
- Sourcing **real world** badges, bumperstickers, cards, ad specialties, etc…
- Generating **encoders** or reconstruction of antique dialplates.
- Visualizing solutions to complex **electromagnetic field problems**.
- Doing detailed **log file analysis** of website activities.
- Bitmap **distortion correction** and ultra legible **super small lettering**.
- Fast and improved single file **Powerpoint Emulations**.
- Creating **Fractal Ferns** and graphically unique **Fibonacci Sunflowers**.
- Analyzing **Magic Sinewave** energy efficiency developments.
- Performing **word frequency** and grade level analysis on published docs.
- Doing ultra fancy complex **nonlinear graphical transforms**.
- Fast and easy "what if?" exploration of engineering problems.

PostScript is best used when its batch mode one-pass interpreted processing output creates a **graphics output** file, an information **reporting log file**, or one or more new **disk based files** in most any format or language.

It can be useful to compare and contrast PostScript with JavaScript…

STYLE — PS shares the reverse polish, stack oriented, loosely typed extensible heritage of Forth; JS has a more conventional C language class architecture.

REACH — PS can easily read or write any host based disk file in most any language. JS is specifically and absolutely forbidden from ever doing so.

GRAPHICS — PS is a world class superb graphics defining technology; JS graphics as normally used are vastly inferior and frustratingly second rate.

INTERACTIVITY — PS is pretty much limited to batch tasks involving creation of graphics files, log files, and host disk read/writes. JS can be exceptionally real time user interactive.

MATH — PS uses 32 bit math but usually only reports to a **somewhat extendible** six decimal places. JS has a full 64-bit floating point capability.

TRIG — PS works directly in degrees, while JS works directly in radians.

VARIABLE SCOPE — PS variables are normally global unless defined otherwise. JS variables are normally local unless declared otherwise.

THE STACK — Elegant stack manipulation is central to PS capabilities. The JS stack is an optional and seldom used sideshow.

CALCULATED VARIABLES — These are easily done with PS, but require a much more obtuse **form["fa00"]["value"]** approach with JS.

LEARNING CURVE — Much of PS is intuitive and easily picked up. JS details can be mind-boggling and maddeningly infuriating.

PASSING VARIABLES BETWEEN PROCS — Is vastly simpler and more intuitive in PS. JS is far more obtuse but can be more flexible.

FORM REFERENCING — Largely unused and unneeded in PS, but confusingly and centrally essential to JS.

NUMERIC ROUNDOFFS — A rare event in PS, but untreated **3.99999999** results in JS are common and may need programmatic correction.

Getting Started with PostScript

The obvious starting point with **PostScript** is to download the free **PostScript Language Reference Manual**. Follow this up with this tutorial on **Using Acrobat Distiller as a General Purpose Host Computer**.

Next, visit the **PostScript** areas of **My Website**, perhaps starting off with our **PostScript Beginner Projects**, our earlier reprint collection of **PostScript Secrets**, or a truly ancient and outdated demo of our **PostScript Show and Tell**.

For details on pagemaking and more advanced layout concepts, you can study most any of our example **.PSL** source code documents you'll find **here** or **here**.

Here is how you create and run your own PostScript program…

1. **Install Acrobat Distiller or its GhostScript clone. Distiller is included in full Acrobat code, but is NOT provided in free Acrobat Reader only downloads.**
2. **Using any word processor or editor, create and save your PostScript program as an ordinary ASCII textfile.**
3. **Send the program to Distiller.**
4. **View your results as a .PDF file, as a reporting log file, or as a newly created or modified custom disk file.**

The usual way you use PostScript is "not quite" WYSIWIG. Typically, a second or two will be required for visual results to appear. Creative use of split or dual screens can greatly minimize any time delays.

Try it with your very first PostScript program…

```
%!PS
36 sin ==
```

This should report the sin of 36 degrees to you as 0.587785. We'll shortly see another PostScript programming example that is somewhat more complex.

My Gonzo Utilities

Many years ago, I started writing an ongoing series of **Gonzo Utilities** that you can download **here**. These are basically **a set of self-activating dictionaries** that you can place (or run) at the beginning of your own **PostScript** programs. They can enormously simplify and speed up many common **PostScript** tasks.

They are also easily expanded upon to meet your own special needs. You can think of these as a mix of custom combined page making, illustration, analysis, and presentation routines.

What the Gonzo utilities basically do is add many hundreds of new commands to the PostScript language. These commands can be used by themselves, or as tools to generate your own custom and fancier commands. Very often, the gonzo commands let you write PostScript code that is **significantly shorter and faster** than other approaches. Sometimes ridiculously so.

About half of the Gonzo commands involve superb text typesetting features. Such as a premium grade word breaking progressive fill microjustify that can include

bells and whistles like drop caps, kerning, and even hanging punctuation. The other half of gonzo is an eclectic mix of convenience operators, graphing aides, high quality electronic schematic drawing tools, and code that lets you quickly generate real world products.

Here is how you use the Gonzo utilities…

1. **Download a free copy of the Gonzo Utilities from here. Place that copy in an accessible host directory.**
2. **Add a line similar to (C:\\gonzo\\gonzo.ps) run near the beginning of your PostScript program. Note that the FULL pathname must be specified and a DOUBLE reverse slash needs used every time a single reverse slash is to be passed to Windows or a similar host.**
3. **Write and save your PostScript program as an ordinary ASCII textfile in the usual manner.**
4. **Send the program to Distiller. Then view the generated .PDF file, log file, or custom disk file as usual.**

You also have the option of extracting only one or two Gonzo routines and placing them early into your code. Ferinstance, our **mergestr** string merging routine has lots of possible stand-alone uses.

As does **showgrid**, **random**, and dozens of the others.

You might also want to place your own extended Gonzo commands very near the beginning of your PostScript code. Such as a task specific page making or layout routine that expands on the basic justification procs.

A Gonzo proc named **colcheck** is used as your normal link between the internal Gonzo typesetting code and your custom (and often job specific) layout routines.

An Example

Before we look at the Gonzo commands in detail, let's look at a genuine and recent real world problem that shows how fast, simple, and powerful these utilities can be.

Someone on an electronics newsgroup posed the question "What would a waveform look like that had all harmonics present and equal in amplitude?"

We can approach this by doing an actual plot of the target waveform. Starting with a fundamental and adding the needed harmonics.

Here is the annotated Gonzo code to quickly plot the answer…

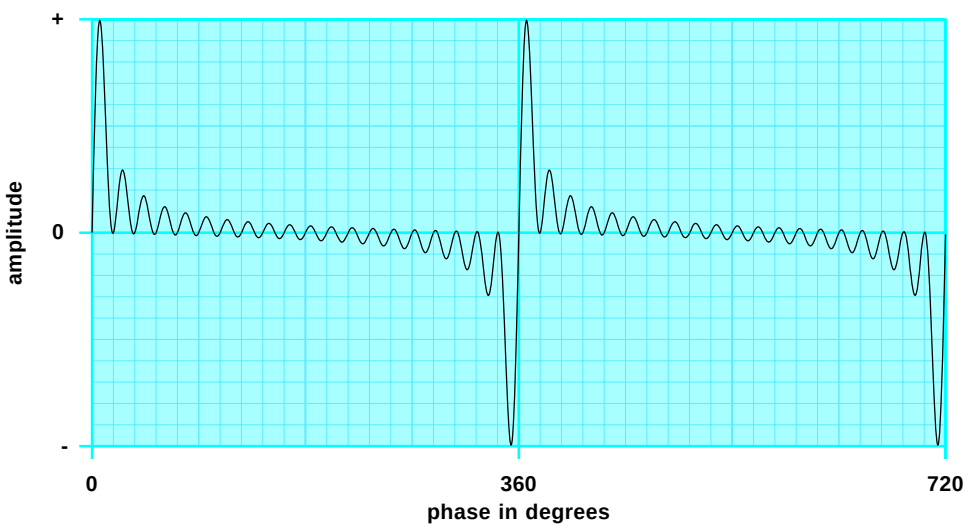
```
%! PS                                  % normal header
% Equal Harmonic gen                  %  title

(C:\\gonzo\\gonzo.ps) run             % run Gonzo

50 50 10 setgrid                      % Create a grid
40 20 showgrid                        % Show part of the grid
/totalharms 20 store                  % Set # of harmonics

0 10 mt                               % Set initial position
0 0.1 720 {/priang exch store         % For one full cycle
  priang 20 mul 90 div                %   Set x position
    0 1 1 totalharms {/curharm        %  For each harmonic,
      exch store priang curharm       %     calculate value
      mul sin add} for
    0.67 mul 10 add lineto} for       %  and set y position
line1 stroke showpage                 % Draw and show
```

And here is what your .PDF file result looks like…



We see that a harmonic limited version would have a narrow positive impulse just beyond zero degrees and a narrow negative impulse just before 360 degrees. This waveform could prove extremely useful in ultrawideband communications apps.

This figure was slightly enhanced by tinting the grid and adding axis callouts. Both tasks are trivial additions when using the **Gonzo Utilities**.

Embedded Gonzo Justification Commands

The text justification portions of the Gonzo Utilities offer you some incredibly sophisticated and advanced features. These include automatic word breaks, progressive two-stage or even three-stage fill justification, left-center-right column or freeform justification, initial drop caps, hanging punctuation, tabbing, global and local character and space kerning, font microsizing, keystoning, overstriking, conditional page anti-orphaning, programmable macros, and dual parallel operation for simultaneous figures and body text.

Being proudly non-WYSIWYG, these features are (a) fully programmable and extensible, and (b) totally device and platform independent. With (c) all source code freely available and easily modified.

Just about any additional text justification feature can be easily added. The justification routines also form the core utilities for page layout or "pagemaking" software of arbitrary complexity.

Gonzo uses the "embedded command" concept of placing markers **inside** text strings. The strings can be independent short **callouts** useful in figures, much longer **document strings** holding up to 65,000 page or multi-page characters, or can use the older PostScript **currentfile** methods without length limits.

Instead of the escape sequences of early typesetters or the **<x>, </x>** marking conventions of HTML, Gonzo often will use alterable markers consisting of a reserved self-delimiting vertical bar followed by a single letter. Gonzo can also embed most any space-delimited PostScript command sequence into your text strings.

An **<esc>** control character or another printing marker can be substituted.

Here is a summary of the more important gonzo embedded text commands…

|0 thru |9
|:, |;, |=, |-, |+,
CHANGE FONT

When used in a Gonzo string, changes to a font predefined as variables **font0** through **font+** using **gonzofont**.

Fonts are predefined in one of two ways by **gonzofont**:.
Use **font0 /Helvetica 10 gonzofont** for normal sizing.
Use **font0 /Helvetica [wide lean climb high xshift yshift] gonzofont** for fancy matrix sizing. For instance, a **yshift** can be used for superscripting or subscripting.

Additional fonts can be defined as needed or block switched in as style groups.

Self delimiting.

|a thru |f MACRO COMMANDS

When used in a Gonzo string, executes your predefined **amacro** through **fmacro** commands at actual print time.

Macros are particularly useful for indenting and outdenting titles while changing their fonts and sizes. These usually will end up document specific. And defined in the document header.

There are six other macros of upper case **|U** thru **|Z**.
More are easily added.

Commands are case sensitive and self-delimiting.
The macros must **not** disturb stack values!

|/anystuff INSERT POSTSCRIPT COMMAND

When used in a Gonzo string, a defined PostScript proc whose name does not conflict with any other Gonzo proc can be executed. The proc must **not** disturb stack values!

The command **must** be followed by a space that will be automatically consumed in processing. Lack of this space will cause a word or line collision. Or an error message.

This proc is normally executed **immediately** during the Gonzo processing time. If the proc is to end up **deferred** so it executes during print time, a **printlist exch 3 index exch put exch 1 add exch** must be properly built into the correctly deferred proc definition as noted below.

Note that only the name of the proc should get executed **between** Gonzo strings. The name of the proc preceded by a bar and a slash should get executed **inside** Gonzo strings.

An **undefined** embedded PostScript command (such as **|/xxxx**) can be used as an **error trapping debugger**. This can be most useful to find out exactly **where** in a long or a complex doc another error problem took place.

|h HALF NEGATIVE LINEFEED

When used in a Gonzo string, **increases** the value of **ypos** by one half of **yinc**.

When used on an in-string line by itself, adds one half a line of white space ledding between any paragraphs. This often gives you the best viewability for modern text presentation.

Self-delimiting.

|i PROVIDE INITIAL DROP CAP

Knocks out a text hole to make room for an initial drop cap. Does this by setting **indententcount** to the number of lines the cap will take up; by setting **dropflag** to true so that indentcount lines will be indented by **dropindent**; and by resetting the vertical position to the initial top line.

The initial character font is subscripted as needed so its **top** will be flush with the top of the top text line.

A raised initial cap is much simpler and can be done by placing one larger, bolder, and colored initial character.

Self-delimiting.

|j POSITIVE UNIT KERN

When used in a Gonzo string, **adds** one **kern** unit of whitespace between current adjacent characters.

kern is normally 1.0 for full size or 0.1 when on a 10X grid.

Used to "add a little daylight" to anything that looks a tad too crowded. Especially useful before or after parenthesis.

Self-delimiting. Global kerning is separately available by using **cstretch** and **sstretch**.

|k NEGATIVE UNIT KERN

When used in a Gonzo string, **subtracts** one **kern** unit of whitespace between current adjacent characters.

kern is normally 1.0 for full size or 0.1 when on a 10X grid.

Used to "close in" adjacent characters that appear too far apart. Useful for converting "AWARD" to "AWARD" and such.

Self-delimiting. Global kerning is separately available by using **cstretch** and **sstretch**.

|l FULL POSITIVE LINEFEED

When used in a Gonzo string, **decreases** the value of **ypos** by a full **yinc** of ledding .

Useful to efficiently "tab downward", replacing repeated carriage returns. Normally should **not** be used in middle of a text line.

Self-delimiting.

|n PARAGRAPH NOBREAK ORPHAN ELIMINATOR

When placed in a Gonzo string, checks to see if you are within five lines of the page bottom. And then forces a column or page change. Useful to eliminate one or two line orphans at the top of the next column or page. Also prevents a header from appearing too far down the page. Self-delimiting.

|o OVERSTRIKE CHARACTER

Useful for placing two characters on top of each other without repositioning. For heading on down the cañyon, or for those $\overline{Q}$ complementary logic notations.

Uses **overstrikechar** to select the secondary character and **overstrikeht** to set the **vertical offset** of the secondary character. These values will have to be redefined for each **different** overstrike use. Self-delimiting.

|p NORMAL PARAGRAPH INDENT

Sets or resets paragraph indentation to the **pm** value. Alternate to |z which zeros any paragraph indent. Self-delimiting.

|s STOP COLCHECK

Sets a high negative value to **ybot** so that relative in-figure positioning will not trip a main text column or page action. Usually internal to a **save-restore** context. Self-delimiting.

|y FULL NEGATIVE LINEFEED

When used in a Gonzo string, **increases** the value of **ypos** by **yinc**. When used on an in-string line by itself, removes one line of white space between text lines.

Useful to "cancel out" a line of pure commands so they leave no text ledding in their position. Self-delimiting.

|z ZERO PARAGRAPH INDENT

Sets or resets paragraph indentation to zero, ignoring the **pm** value. Complements |p which sets or resets any paragraph indent. Self-delimiting.

|C SWITCH TO CENTER JUSTIFY

Selects the center justify mode for all text to follow. Alternate choices are |F for fill, |L for left, and |R for right.

|F **SWITCH TO FILL JUSTIFY**

Selects the fill justify mode for all text to follow. Alternate choices are **|C** for center, **|L** for left, and **|R** for right.

|L **SWITCH TO LEFT JUSTIFY**

Selects the left justify mode for all text to follow. Alternate choices are **|C** for center, **|F** for fill, and **|R** for right.

|P **SWITCH TO CUSTOM JUSTIFY "P"**

Selects a custom justify macro that the user has preprogrammed to a name of **justP**. Useful for menu justify, supertabbing, keystoning, or other specialized text formatting apps.

|Q **SWITCH TO CUSTOM JUSTIFY "Q"**

Selects a custom justify macro that the user has preprogrammed to a name of **justQ**. Useful for menu justify, supertabbing, keystoning, or other specialized text formatting apps.

|R **SWITCH TO RIGHT JUSTIFY**

Selects the right justify mode for all text to follow. Alternate choices are **|C** for center, **|F** for fill, and **|L** for left.

|U thru |Z **MACRO COMMANDS**

When used in a Gonzo string, executes your predefined **Umacro** through **Zmacro** commands at actual print time.

Macros are particularly useful for indenting and outdenting titles while changing their fonts and sizes. These usually will end up document specific. And defined in the document header.

There are six other macros of lower case **|a** thru **|f**.
More are easily added.

Commands are case sensitive and self-delimiting.
The macros must **not** disturb stack values!

Gonzo Text Justification Commands

There are a few service routines built into the Gonzo text justification. These are normally **used directly as PostScript commands**, rather than being embedded into strings for later evaluation.

Here are some of the more common or more useful text variables and procs…

-xpos- -ypos- (text string) cc **CALLOUT CENTER JUSTIFY**

Center justifies a text string on the **-xpos-** value, starting with a vertical text baseline of **-ypos-**. Normally increments downward as **-yinc-** and is evaluated by **colcheck**. Does **NOT** respond to **txtwide**. This allows wider titles than text columns.

Use internal **\n** or actual newline chars for multiple lines.

-xpos- -ypos- (text string) cf **CALLOUT FILL JUSTIFY**

Fill justifies a text string on the left **-xpos-** value, starting with a vertical text baseline of **-ypos-**. Normally increments downward as **-yinc-** and is evaluated by **colcheck**. Width of flush column is set by **txtwide**.

-xpos- -ypos- (text string) cl **CALLOUT LEFT JUSTIFY**

Left justifies a text string on the left **-xpos-** value, starting with a vertical text baseline of **-ypos-**. Normally increments downward as **-yinc-** and is evaluated by **colcheck**. Width of ragged right column is maximum set by **txtwide**.

/colcheck {custom user proc} store **COLUMN TESTER FOR PAGE LAYOUT**

colcheck is the crucial link between the Gonzo justification procs and any user defined custom page layout or formatting code.

Normally, **colcheck** will attempt to start a new line that is **yinc** below the previous one. If **ybot** is negative, then a custom action is taken to move on to the next column or the next page.

The user normally places their **colcheck** code early in their program but **after** the Gonzo procs are installed and run.

A custom **colcheck** routine will normally adjust **xpos** and **ypos** to go to the next column, or will do a **showpage** and a setup as needed for the following page.

-xpos- -ypos- (text string) cr **CALLOUT RIGHT JUSTIFY**

Right justifies a text string on the right **-xpos-** value, starting with a vertical text baseline of **-ypos-**. Normally increments downward as **-yinc-** and is evaluated by **colcheck**. Width of ragged left column is maximum set by **txtwide**.

/cstretch 0.15 store **SET GLOBAL CHARACTER KERNING**

Adds a small amount of white space to each and every character. Useful to "lighten" text or to prevent collisions on very small point sizes. Global space kerning is separately set with **sstretch**.

A little of this goes a long way, except for intentional special effects. Use 0.1 or 0.15 for normal text or 0.01 or 0.015 when you are on a 10X grid.

/fname /family 10 gonzofont **DEFINE FONT FOR GONZO USE**

~ or ~

/fname /family [wide lean climb high xshift yshift] gonzofont

Defines and creates fonts for internal Gonzo use. **fname** is typically **font0** through **font-**. **/family** is the normal exact PostScript font name, such as **/Helvetica**.

In the first instance, one numeric sets the point size. Points can be fractional such as **9.557** points or microsized if desired.

In the second instance, the full font array is used to set the font characteristics. The first value is the **width** of the font. The second value is the amount of font **lean** and is normally used to create italic effects. The third value is the amount of font **climb** and is normally reserved for rotations, isometric, or any other distortions. The fourth value is the **height** of the font.

The fifth value is the amount of **horizontal offset** of the font, and almost always will remain at zero. The sixth value is the amount of **vertical offset** of the font. This is enormously useful when creating superscripts or subscripts.

Note that **gonzofont** only **creates** a Gonzo font. The font must be separately activated. Perhaps by an immediate **font1** or by embedding a **|0** in the string currently being justified.

(list string) cck **KEYSTONE JUSTIFY**
(list string) clk
(list string) crk

Attempts to do a keystone justify by "unifying" the progressive widths in a sequential list. Per **this example**.

cck does a centered keystone. **clk** will do a flush left keystone. **clr** does a flush right keystone. The relative lengths of the early

list entries versus the later ones determines whether the keystone will be fatter at the top or bottom. Considerable adjustment of the text line kerning may be needed to get decent results.

/lastlinestretch 0.2 store **LIGHTEN LAST FJ PARAGRAPH LINE**

A typical line in a fill justified paragraph will normally be stretched somewhat. If the final line is **not** stretched, it may appear "too dark". This somewhat arcane adjustment stretches final fill justified paragraph lines that are less than 80 percent of maximum. Use a value of 0.0 to defeat.

-xpos- -ypos- (text string) mj **MENU JUSTIFY**

Does a "menu justify" by replacing multiple spaces with constant pitch dots. Not included in this version, but is found **here**.

[{proc1}{proc2}...{procn}] **THE PRINTLIST**

Fancy text justification is inherently a two step process. Each line first needs analyzed to find out what is needed in the way of font changes, kerning, stretches, and such. Each and every internally consistent element is then placed into a **printlist** for a sequential **forall** evaluation at print time.

Many embedded commands will automatically be deferred till print time. However, any special effects (such as making one word **red**) that are defined as PostScript procs may execute **immediately** instead of at the deferred print time.

We will see how to insert new commands into the printlist below. The magic incantation you will need is **printlist exch 3 index exch put exch 1 add exch**.

/tabs [10 20 30 40] store **THE TAB LIST**

Defines the position of simple tabbing used in the left justify modes. A single **|t** moves you absolutely to the next tab value. A double **|t |t** moves you absolutely to the second tab value, and so on. Tab values need not be sequential.

/txtwide 43 store **SET JUSTIFICATION TEXT WIDTH**

Sets the maximum column width of all justify modes except **cc**.

/cmacro (znhL7) stringmacro def **STRINGMACRO**

A convenience tool to aid in macro building. Executes every character in the string as an **individual** embedded command.

Ferinstance, in the above title positioning macro, **(z)** defeats any paragraph margins, **(n)** prevents starting a title at the very bottom of the screen or page, **(h)** does a half linefeed upwards to improve ledding, **(L)** switches to a left justify, and **(7)** picks **font7**. Other PostScript procs can precede or follow stringmacro use.

/sstretch 0.15 store **SET GLOBAL SPACE KERNING**

Adds a small amount of white space to each and every space. Useful to "lighten" text or to prevent collisions on very small sizes. Global character kerning is separately set with **cstretch**.

A little of this goes a long way, except for intentional special effects. Use 0.1 or 0.15 for normal text or 0.01 or 0.015 when youare on a 10X grid.

-xpos- -ypos- (data string) cst **SUPER TABBING**

Does an exotic "supertabbing" where individual column entries can have their own fonts, their own justifications, and their own special effects. Any occurrance of **two or more** sequential spaces in the data string is treated as a "move to next column" tab command. A **stab** array sets each column's values.

Not included in this version, but is found **here**.

/xpos 45 store **SET TEXT HORIZONTAL POSITION**

xpos is normally used automatically to set the start of a text line. It may also be preset or manually overridden.

/ybot 0 store **SET COLUMN BOTTOM LIMIT**

ybot is normally used automatically to set the bottom of page trip point for a new column or page move. This may also be preset or manually overridden.

/yinc 10 store **SET TEXT LINE VERTICAL SPACING**

yinc is normally used automatically to set the ledding between text lines. It may also be set or overridden.

/ypos 45 store **SET TEXT VERTCAL POSITION**

ypos is normally used automatically to set the vertical position of a text line. It may also be preset or rewritten.

Additional Gonzo Text Justification Variables

Most of the other variables used in Gonzo text justification are somewhat self-explanatory. Here is a summary of many of these commands…

```
/altescapechar 124 def          % alternate "escape" key ( | )
/dropflag false def             % use a drop cap?
/dropcount 3 def                % drop cap lines indented
/dropindent 40 def              % drop cap width reserved
/escapechar 27 def              % original "escape" key
/hangflag true def              % allow hanging punctuation?
/hangfract 0.6 def              % hung punctuation hang amount
/justifylastline false def      % fill justify last paragraph line?
/justx (justL) def              % running justification mode
/kern 1 def                     % default individual kern amount
/oktoadvance true def           % don't go to next line if false
/oktoprint true def             % print suppression flag
/overstrikechar (—) def         % overstrike character
/overstrikeht 5.5 def           % overstrike vertical shift
/pm 10 def                      % normal paragraph indent
/rslashchar 92 def              % "reverse slash" key -alterable-
/rslashok true def              % allow reverse slash processing?
/stringmode false def           % string or currentfile source?
/spacecharratio 6 def           % fj ratio of space to char stretch
/sstretch -0.3 def              % minimum space kerning
/txtwide 350 def                % width of column
/ypara 0 def                    % extra paragraph end v space
/ybot -9999 def                 % default bottom reference
```

Deferring Execution Until Print Time

When setting fancy text, Gonzo normally does a whole line at a time. To do this, there is a current Gonzo **printlist**. The printlist is an array of executable procs that are done in strict sequential order as a **forall** loop.

Each proc typically might be a group of words having the same font, point size, and weight. Several procs might be needed per line if there are size, kerning, italic, bold, or special effect changes in the middle of the line.

The exact current xpos position on the line may not be known ahead of time! Especially with a fill justify. And its exact position may depend highly on what has already been put down.

There are all sorts of sneaky and powerful things you can do by inserting an additional proc or two **inside** your printlist. For instance, you might want to make one word **red**. Or you might like to place an emphasis box or fancy graphics underneath a few words.

Or, most importantly, you might want to set an "action block" or a PDFMarking Acrobat **ANN link** underneath a url. One that **automatically tracks** the printed url name and length. Regardless of where on the line the url text is or how much post editing is done. And needing no later manual intervention.

When creating a Gonzo text proc, a pointer to the last used printlist entry starts off as the **second** stack element. It becomes the **third** stack element when you put your new deferred proc on the stack.

Here is the general method of inserting a top-of-stack proc into your printlist....

1. Create an executable (but NOT executing!) array.
Then make the array executable.

2. Add this magic command sequence...

```
printlist exch 3 index exch   % stuff into printlist
put exch 1 add exch           % incrementlist count
```

What this does is take your top-of-stack proc, adds it as the newest array element to your printlist, and then increments and properly replaces the printlist pointer.

Here is how you would make one word red and the rest of the line black again...

Place this code before your text strings...

```
/setred {mark 1 0 0 /setrgbcolor cvx ] cvx printlist
 exch 3 index exch put exch 1 add exch} store

/setblack {mark 0 0 0 /setrgbcolor cvx ] cvx printlist
 exch 3 index exch put exch 1 add exch} store
```

And use it like this...

```
100 200 (This |/setred word|/setblack  is red.) cl
```

Note that everything in a PostScript array executes immediately. If you want to create any **PostScript** proc that executes **later** rather than immediately, you have

to use a roundabout generation method. You thus have to use an indirect approach of **/setrgbcolor cvx** (which defers) compared to **setrgbcolor** (which does not).

Yes, we could also have done our red word by defining a pair of our twelve macro commands. And this route would be self-deferring and self-delimiting without any **printlist** hassles. But there are many advantages to space delimited and named PostScript procs that are **printlist** insertable. Especially when, say, lots and lots of url's or exotic (Why did I do that?) commands may be involved.

Speaking of which, a much fancier example that does auto-positioning and auto-tracking for Acrobat web url's appears **here**. Autotracking emphasis boxes can use similar techniques.

Gonzoing Gonzo

A very few characters in Gonzo are usually highly reserved. So showing them (as is needed in this Gonzo tutorial) can be tricky.

For instance, a reverse slash inside a **PostScript** string really has to be shown as a **double** reverse slash. This gets important in a hurry when trying to write or read a host diskfile. For a printed or screen display, it will take **two** Gonzo slashes to equal one PostScript slash and **two** PostScript slashes to pass one real slash on to Windows or whatever. To show double reverse slashes on screen, **eight** reverse slashes are needed in the original Gonzo string!

Showing a vertical bar can be tricky if this is also how we identify embedded commands. Two workarounds are to redefine **rslashcar**. Or (as we've done here), simply draw the vertical slash as an embedded PostScript graphic line.

Page Layouts

Usually, you will want to combine fancier page layout code with all of these fundamental justification procs. This can be done by adding new commands **early** in your document but **after** Gonzo is first run. Detailed examples can be found in the **sourcecode** for this **GuruGram**, or by studying most any of the **.PSL** source codes found **here**, **here**, and elsewhere on **my website**.

Now for the Rest of Gonzo

The precision justification procs we just looked at are only a small portion of the **Gonzo Utilities**. Here are some of the many other available commands, arranged by group…

(A) SERVICE UTILITIES —

These service utilities greatly simply a wide range of common PostScript tasks…

-xside- -hypotenuse- acos — Finds trig inverse cosine

-xside- -hypotenuse- asin —	Finds trig inverse sine
backwards	Prints backwards
bestgray	Best gray for lores printers
blackflash	Drum conditioner for older printers
-count- copies	Set number of print copies
feetfirst	Eject print feet first
flushends	Flush path ends
flushjoins	Flush path joins
GEniejul	6-digit to Julian date converter
-xstart- -ystart- -yend- -linewidth- hrule	fixed horizontal rule
inch	Inches
indiagray	India ink wash at 300 DPI
landscape	Landscape format of 8-1/2 x 11
listfonts	List installed fonts
longjob	Lengthen job timeout
manual	Select manual feed on printer
(string1) (string2) mergestr	Merge strings to stack top
negative	Negative printing
outline	Find character path
pi	As in 3.1415926
pixel	Points to 300 DP pixels
positive	Restore positive printing
printfonts	Prints currently installed fonts
-max+1- random	Random integer 0 to max
report	Report top of stack to host
reprogray	Reprogray at 300 DPI
-num- romnum	Convert 0-99 num to Roman string
roundends	Round path ends
roundjoins	Round path joins

-rad- [x1 y1 x2 y2 ... xn yn] roundpath Rounds path except ends

-xpos- -ypos- -xwid- -yhgt- -crad- roundbox Build a rounded box

snoop Activates superexec

stockends Normal (extended) path ends

stockjoins Normal (extended) path joins

-delayvalue- stall Stall at roughly 1000 per second

stopwatchon Reset and start stopwatch

stopwatchoff Stop and report stopwatch

-#repts- -spacing- (char) stringdown South character repeats

-#repts- -spacing- (char) stringleft West character repeats

-#repts- -spacing- (char) stringright East character repeats

-#repts- -spacing- (char) stringup North character repeats

[wd1 gr1 wd2 gr2 ... wdn grn] superstroke Mulltiple full strokes

[wd1 gr1 ... wdn grn] superinstidestroke Mulltiple clipped strokes

-angle- tan Find tangent of angle in degrees

tray Turn off printer manual feed

-xstart- -ystart- -yend- -linewidth- vrule fixed vertical rule

white Print in white

(string) width Find width of string

(B) LAYOUT GRIDS —

Much of the early grid layout code was aimed at providing exceptional graphics quality on first generation 300 DPI printers. While some of this portion of Gonzo is dated, these two commands (and their internal support procs) remain useful…

-xposn- -yposn- -scale- setgrid Create a scaled layout grid

Working on a magnified grid can greatly simplify layouts. You are dealing with smaller numbers whose relationships are more intuitive and more obvious. A command of **50 60 10 setgrid** moves you to a point 50 points right and 60 points up from the **lower left** page corner. This establishes the **0,0** or "home" position of your grid. The invisible grid is scaled 10X and extends infinitely in all directions.

It is important to define such values as **cstretch** and **sstretch** well **after** you create the grid. Defined font sizes should also be later and proportionally smaller.

Use of **setgrid** leaves an "open" **gsave** graphics state which may need closed later.

-#ofxblocks- -#ofyblocks- showgrid	Show a portion of the grid

This command can be used for layout visualization or as a final design element of a graph or chart. By changeable default, there is presently a slight emphasis of each **fifth** grid line and a stronger emphasis of each **tenth** grid line. The easiest way to turn a grid on or off is to **comment** the above line. you can do this by placing or removing a **"%"** at its start.

A detailed example of using a layout grid to create book covers appears **here**.

(C) ILLUSTRATION AIDES —

The whole point of the Gonzo Utilities is to give you **absolute and total control** of your design and layout projects. Very often resulting in code that yields much higher quality in files that are exceptionally short and very fast running. Yes, point-by-point entry of graphics instructions can get rather tedious. But this approach very often can give you stunning results not easily available elsewhere.

Many of these illustration aides are simply convenience operators that require far fewer keystrokes than "raw" PostScript. You can easily further expand and customize them for your own needs…

-xpos- -ypos- mt	absolute moveto shorthand
-xpos- -ypos- rm	relative moveto shorthand
-xpos- -ypos- rl	relative lineto shorthand

These routines set linewidths…

line1	A "normal" grid line
line2	A "bold" grid line
line3	A "very heavy" grid line

These routines draw a line…

-distance- d	relative line to the south
-distance- l	relative line to the west
-distance- l+	relative line to the northwest
-distance- l-	relative line to the southwest
-distance- r	relative line to the east
-distance- r+	relative line to the northeast
-distance- r-	relative line to the southeast
-distance- u-	relative line to the north

These routines append to a path…

-distance- pd	relative path to the south
-distance- pl	relative path to the west

-distance- pl+	relative path to the northwest
-distance- pl-	relative path to the southwest
-distance- pr	relative path to the east
-distance- pr+	relative path to the northeast
-distance- pr-	relative path to the southeast
-distance- pu-	relative path to the north

These routines draw a line while "erasing" any lines that are crossed. This is quite useful in an electronic schematic for "lines crossing but not connected". …

-distance- dx	erase & draw to the south
-distance- lx	erase & draw to the west
-distance- rx	erase & draw to the east
-distance- ux	erase & draw to the north

If you really want your electrical lines connected, you can add a dot…

-xpos- -ypos- mdot	dot two crossed lines

Here's some circles and some arrows…

-xpos- -ypos- darrow	arrow points south
-xpos- -ypos- larrow	arrow points west
-xpos- -ypos- rarrow	arrow points east
-xpos- -ypos- uarrow	arrow points north
-xpos- -ypos- circ1	small circle
-xpos- -ypos- circ2	terminal sized circle
-xpos- -ypos- circ3	LED sized circle
-xpos- -ypos- circ4	test point sized circle

A pair of very versatile repeat utilities. …

[{proc} -spacing- -#trips-] xrpt	Repeat proc horizontally
[{proc} -spacing- -#trips-] yrpt	Repeat proc vertically

Finally, a "liquid paper" path whiteout…

whitefill	erase anything under icon

(D) ELECTRONIC SYMBOLS —

These electronic symbols offer significantly higher quality than is found in most schematic drawing packages. They are one example of the many possibilities that positionable "**opaque blob icons on strings**" offer. Note that you can slide the wires "under" the icons simply by placing them **earlier** in your code…

micro	Show /Symbol font3 mu for capacitors
ohms	Show /Symbol font3 Omega for resistors
-xpos- -ypos- tstpt	Draw electronic test point opaque circle

-xpos- -ypos- rinverter	Right facing logic inverter
-xpos- -ypos- linverter	Left facing logic inverter
-xpos- -ypos- hresistor	Horizontal resistor
-xpos- -ypos- vresistor	Vertical resistor
-xpos- -ypos- lpot	Vertical potentiometer
-xpos- -ypos- vcap	Vertical capacitor
-xpos- -ypos- uvcap	Inverted vertical capacitor
-xpos- -ypos- hcap	horizontal capacitor
-xpos- -ypos- schmitt	Schmitt trigger hysteresis symbol
-xpos- -ypos- dpdt	Double pole double throw switch
-xpos- -ypos- spdt	Single pole double throw switch
-xpos- -ypos- diode	Diode pointing east
-xpos- -ypos- udiode	Diode pointing north
-xpos- -ypos- ddiode	Diode pointing south
-xpos- -ypos- led	LED diode pointing south
-xpos- -ypos- negpulse	Negative going pulse waveform
-xpos- -ypos- pospulse	Positive going pulse waveform
-xpos- -ypos- 5vdc	Vertical stub and terminal
-xpos- -ypos- hxtal	Horizontal crystal
-xpos- -ypos- sensor	Sensor
ground	Normal south pointing ground
uground	North pointing ground
lground	West pointing ground
rground	East pointing ground
-xpos- -ypos- edgecon	Edge connector pin
-xpos- -ypos- cell	One vertical battery cell
-xpos- -ypos- -#loops- winding	Horizontal inductor or winding
-xpos- -ypos- -#loops- vwinding	Vertical primary winding
-xpos- -ypos- -#loops- vrwinding	Vertical secondary winding
-xpos- -ypos- phonejack	Full size phone jack
-xpos- -ypos- lilphonejack	Smaller phone jack
-xpos- -ypos- varistor	Varistor
-xpos- -ypos- piezo	Piezo transducer
-xpos- -ypos- pctab	Printed circuit terminal

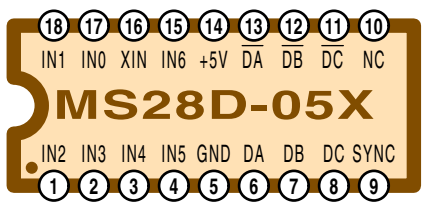
-xpos- -ypos- npn	East facing NPN transistor
-xpos- -ypos- npn	West facing NPN transistor
-xpos- -ypos- pnp	East facing PNP transistor
-xpos- -ypos- pnp	West facing PNP transistor

Many more electronic symbols are easily added. Hundreds of improved (for color options) and more specialized examples may be found by viewing the **.PSL** source code files **here**.

(E) THE DIPDRAW PROC —

In general, I've been hesitant to upgrade portions of the **Gonzo Utilities**. Based on "iffen it ain't broke, don't fix it". Thus, ongoing patches and add-on mods are the norm, compared to core revisions.

The **dipdraw** was an example of an ultra fancy programmable icon whose use kept recurring in my electronics stories…



Here is what the calling code looks like…

```
14 6 moveto
18
(MS28D-05X)
(IN1 IN0 XIN IN6 +5V /DA /DB /DC NC)
(IN2 IN3 IN4 IN5 GND DA DB DC SYNC)
dipdraw
```

Any forward slashes are used to force a complement bar. An improved colorized version of the Dipdraw code can be found in the **source code** for this **GuruGram** . Actual chip details can be found **here** .

(F) STEP AND REPEATS —

Quite a few very useful real world products need a **step and repeat** capability. In which more than one item gets imaged per page. With or without sequential numbering or custom data base access. Examples include business cards, tickets, address labels, badges, bumperstickers, and great heaping bunches more.

A very versatile **stepandrepeat** capability is built into the **Gonzo Utilities**. Two new elements are needed to use this proc. The first is a named and predefined 9 entry array that goes in a **stepnrptparams** dictionary. The second is a predefined **repeatproc** that contains the artwork code for **one** of the items being repeated.

The usual access is…

```
/repeatproc_name stepandrepeat
```

Nineteen repeat patterns have already been preprogrammed into the stock Gonzo **stepnrptparams** dictionary…

/admitonetick	**45 tickets**
/babybumper	**20 very small bumperstickers**
/badgeaminit	**6 badges**
/bigbumpstick	**3 bumperstickers**
/buscard	**12 business cards**
/busenvelope	**1 business envelope**
/eightlabel	**1/8 page 2x4 labels**
/fulllandpage	**One entire landscape page**
/fullportpage	**One entire portrait page**
/lilbumpstick	**5 medium blumperstickers**
/quadsplit	**1/4 page 2x2 labels**
/readerserv	**300 reader service numbers**
/shiplabel	**4 custom labels**
/sixlabel	**1/6 page 2x3 labels**
/stdplabel	**11 stock data processing labels**
/tenlabel	**1/10 page2x5**
/videospline	**13 VHS cassette splines**
/3.5disklabel	**6 disk labels 3.5 inch**
/5.25disklabel	**7 disk labels 5.25 inch**

Many detailed step-and-repeat use examples are found **here**. For sequential numbering, existing **startnum** and **runnum** variables can be used. These same or similar numbers can extract names and addresses from an externally generated data array. Such an array can be built into your code or else **run** just like you did the **Gonzo Utilities**.

Some step-and-repeat projects will have to be carefully matched to the stock being printed. Especially when shear or fold points are built into the material. One useful source of such forms is **Blanks USA**.

If you wish to add or modify your own repeat pattern data as a **stepnrptparams** dictionary entry, the needed nine array values are…

```
/myrepeatdata [
        #hrepeats       % number of horizontal repeats
        #vrepeats       % number of vertical repeats
        hspacing        % proc to proc horizontal spacing
        vspacing        % proc to proc vertical spacing
        hstart          % first proc horizontal offset
        vstart          % first proc vertical offset
        ticklength      % registration tick length if used
        useticks        % true-false registration tick flag
        uselandscape    % true-false landscape orientation
                ] store
```

(G) CURVETRACING —

High quality smooth continuous curves are crucial for typography, animation, and many similar graphical tasks. These are normally created by using **cubic splines**, often by using the PostScript **curveto** and **rcurveto** operators. Some detailed expositions of the underlying math appear in our **cubic spline** library.

Here is our **Puss de Resistance** done entirely in manually entered cubic splines…

Getting the ends of the continuing custom splines to meet and match is both non-trivial and tricky. The original **Gonzo Utilities** provided a rather tedious approach to quality spline generation. In which you entered the position and the angle of each chosen end point as three data values…

```
[ x1 y1 ang1  x2 y2 ang2 ... xn yn angn ] curvetrace
```

Each selected point above a three point minimum had its **horizontal position**, its **vertical position**, and its **slope angle** entered into your array. Thus giving you total control of appearance at the spline joints. **x1=0** and **y1=0** appends path.

The actual splines used were intentionally somewhat weaker than a theoretical optimum. The quality of your results are highly dependent on how accurately you entered your position and angle information. And upon how far you tried to go with each single spline. And, of course, how patient you were with manual data entry of more complex or multiple paths.

An additional variable called the **tension** gave you an additional "smoothness" control. With the usual "best" tension being somewhere around **2.8**. Higher tensions gave you "straighter" curves between end points, while lower ones became moderately or extensively "loopy".

An optional point showing **showtick** true-false flag was included for debugging.

A major change and improvement in the Gonzo curvetracing utilities was made **here**. You can now optionally **enter angle values of 999** for **internal** points. The code will make an amazingly sophisticated "guess" as to your best angle each time. And thus eliminating much (but certainly not all) of the tedium.

A demo of the new curvetracing code appears **here** .

This "**guess angle**" algorithm is based on taking three sequential points and then attempting to temporarily **draw a circle through them**. The slope at the midpoint is often a good and possibly optimally correct fit for your curve of interest.

(H) CIRCULAR ARC JUSTIFY —

An **Arc Justify** routine sets text along a **circular** path. This is useful for labels, for logos, badges and for large names on the back of jackets or T-shirts. A fairly fancy arc justify routine is included in the **Gonzo Utilities**. For top appearance, both local and global kerning is supported.

Here is how you activate a Gonzo arc justify…

```
-xpos- -ypos- radius (your message) karcjustify
```

A **positive** radius value causes the message to appear **clockwise across the top**. A **negative** radius value causes the message to appear **counterclockwise across the bottom**. In either case, the radius will be to the **baseline** of the text. Thus, **your upper and lower radius will normally differ by the average character height**.

Upper and lower messages are normally **centered along a vertical axis**. Any repositioning can be done by adding leading or trailing spaces. Since each and every character is separately positioned, the earlier and fancier Gonzo Justification features are **not** available for an arc justify.

A predefined **arckern** variable will add positive kerning to each and every letter and space. A custom **customkern** variable will create a (usually negative) space every time a **customkernchar** is called. Typically, arckern might be +1, your customkern -1, and customkernchar a **(~)** string. To reduce the space between two printing characters, you **place a "~" between them**.

Several detailed examples of arc justify projects appear **here**.

An Important Update

Starting with Acrobat 8.1, the ability to read and write disk files **has been disabled** as a default. To use our Gonzo utilities, this read and write ability **must** be restored.

In Windows, the Distiller file read and write ability is restored by using a command line run command of…

```
acrodist -F
```

The read and write enabling will remain active so long as an executable version of Distiller is on or under the desktop. To use, you simply drag and drop your Gonzo routines into this active instance of Distiller.

For More Help

Similar tutorials and additional support materials are found on our **PostScript** and our **GurGram** library pages. As always, **Custom Consulting** is available on a cash and carry or contract basis. As are seminars.

For details, you can email **don@tinaja.com**. Or call **(928) 428-4073**.