

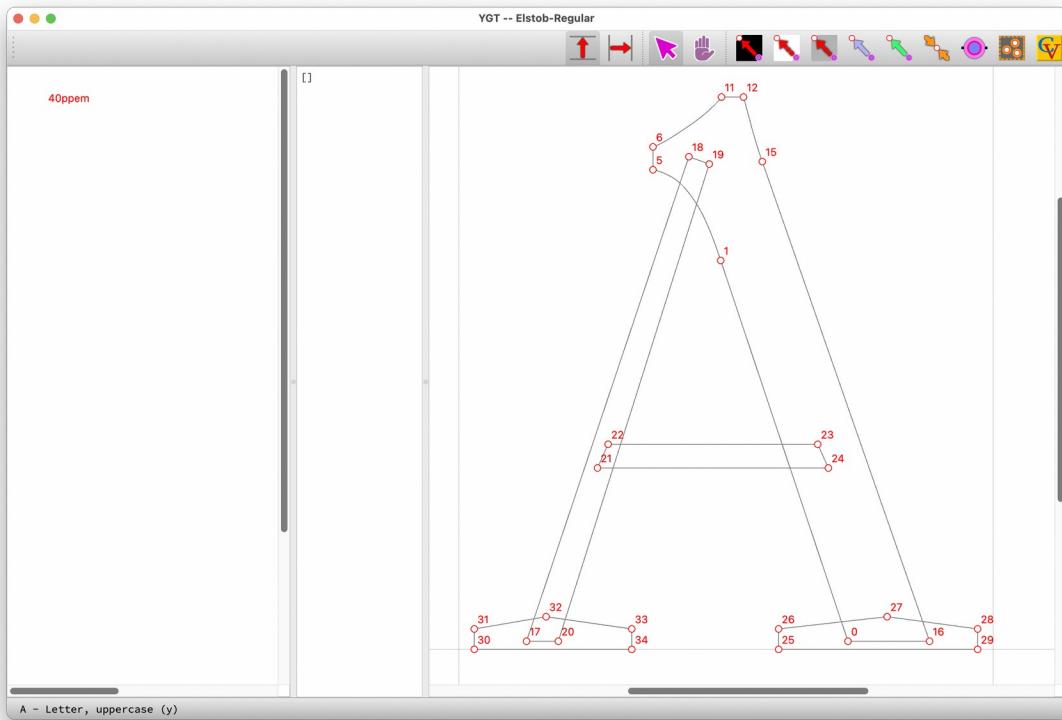
# ygt: A graphical hinting program (a quick-and-dirty guide)

**Ygt** is a program for hinting TrueType fonts. Its interface is similar to those of VTT and Glyphs, but it has somewhat different aims:

- Being written in Python, with PyQt6 and fontTools, it aims to run on any platform.
- It is independent of any font editing program.
- It can either output a hinted font itself or produce a script that can be run from makefiles and the like: it is perfect for use with fontmake.
- In addition to the kinds of hints that other programs generate, it provides an interface for user-written functions and macros, which can greatly simplify the handling of repeated features of glyphs, like serifs.
- It works only with hints and does not touch any other features of a font (for example, it doesn't reposition components the way VTT does).

## Getting started

To get started on a project, start **ygt** and open the TrueType (.ttf) font you want to add hints to. (Any hinting already in the font will be ignored, and later on will be discarded in favor of the hints you add with **ygt**.) You will see a screen like this one:



On the left is the preview panel. A preview of the hinted glyph is not shown or updated automatically; to see it, select “Update Preview” from the “Preview” menu or type Ctrl-U (Cmd-U for the Mac).

In the middle is the code panel, which displays the source code for either the y or the x axis of the current glyph. This source code is in YAML, a very simple data serialization language. **ygt** uses YAML instead of a programming language because it is meant to be descriptive, not imperative: it describes the positions of the points that make up a glyph outline rather than dictating the commands that will position them. The **ygt** hinting language will be described in detail below. You can edit the code in this panel: when you have done so, type Ctrl-R to compile it and display the results in the editing panel to the right.

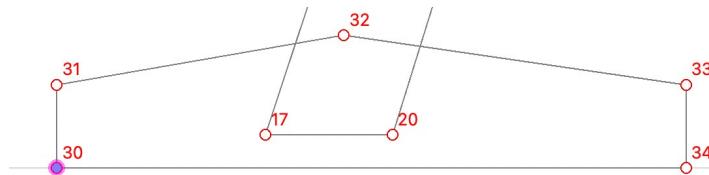
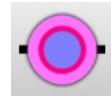
The editing panel displays an outline of the current glyph with the points that define it. On-curve points are displayed as red circles and off-curve points as smaller blue circles. As hints are mostly applied to on-curve points, you may wish to hide the off-curve points. Right-click anywhere in the panel and select “Hide off-curve points” from the context menu to do this. On the same context menu you can decide whether or not to display point numbers.

Above the editing panel is a toolbar with various tools for editing the glyph’s hints.

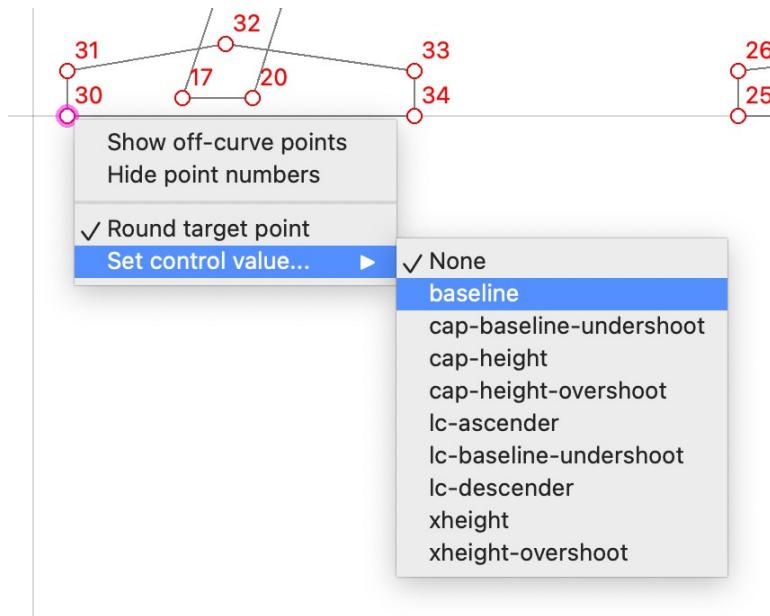
## Basic hinting

To add a hint, select one or more points (click on them or drag the mouse over them) and either click on a button in the toolbar or type a shortcut key. (The shortcut keys are displayed in the tooltips that appear when you move the mouse over the tool buttons.)

For example, to anchor a point to a position on the grid (which you should do first whenever beginning a sequence of hints), select a point and either click the “Anchor” button (at right) or type “A.” The anchored point will acquire a thick pink outline:

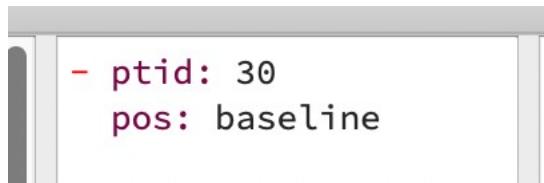


Now you can right-click on this point, select “Set control value” from the context menu, and choose the control value describing the place on the grid where the point should be positioned:

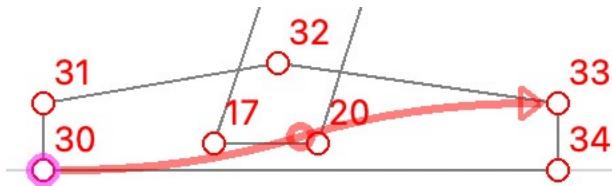


Control values (numbers that specify **positions** or **distances** on the grid) are named rather than numbered in ygt, making the selection quick and easy. Control values are alphabetized in the menu (think about this when you’re choosing names for them), which will display those that are most appropriate for the type of hint you’re adding them to.

As you add hints in the editor, the code panel is automatically updated:



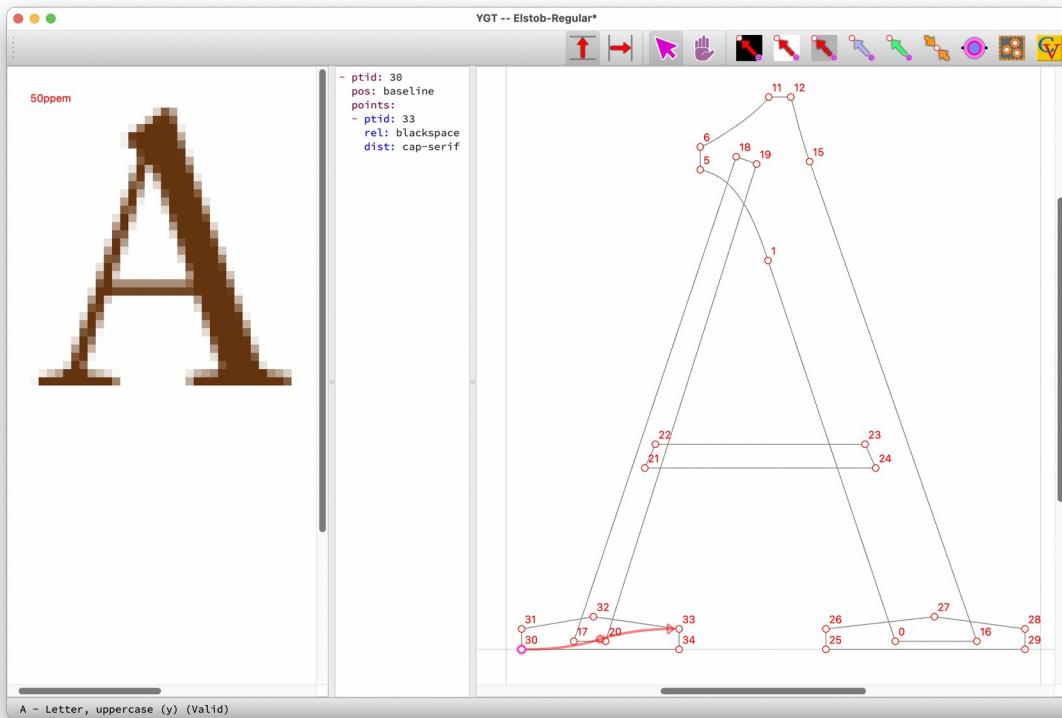
To add another hint dependent on the first, select the point you just hinted (clicking will select the hint rather than the point: you need to drag over it) and another point, then click the tool with the red arrow on a black background or type “B”:



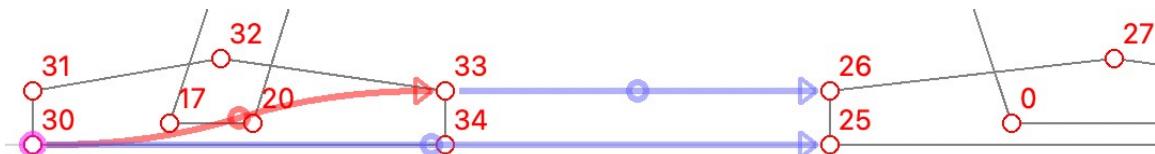
You have just added a “blackspace” hint—that is, one that crosses a space that will be filled (usually by black pixels or black ink). It is displayed as a red arrow running from a **reference point** (in this case, point 30, which you just anchored) to a **target point** (33). The button in the middle of the arrow’s stem is there to make it easier to click on with a mouse pointer.

You can choose a control value for the blackspace hint just as you could for the anchor hint. While the control value for an anchor point always specifies a position, one for a blackspace hint specifies a distance.

Since we’ve added a couple of hints, let’s type Ctrl-U to get a preview and see what we’ve done:

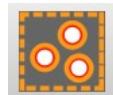


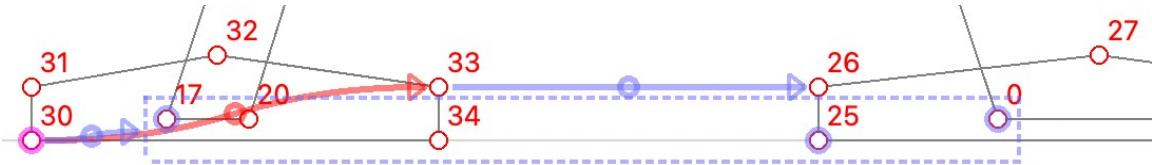
Notice that the serif on the left, the hinted one, looks different in the preview from the one on the right, which is too heavy. To make them match, we'll add some "shift" hints (the tool with the blue arrow). "Shift" causes points to be moved by exactly as much as a reference point has been moved. Select points 30 and 25 and click the blue arrow or type "S." Then do the same with 33 and 26:



We also need to touch the bottoms of the two diagonal stems, since we are going to hint the tops. (Any hint that targets a point leaves that point "touched" so that it can't be moved inadvertently.) The best way to do that is to add points on the bottom of the stems to one of the nearby "shift" hints.

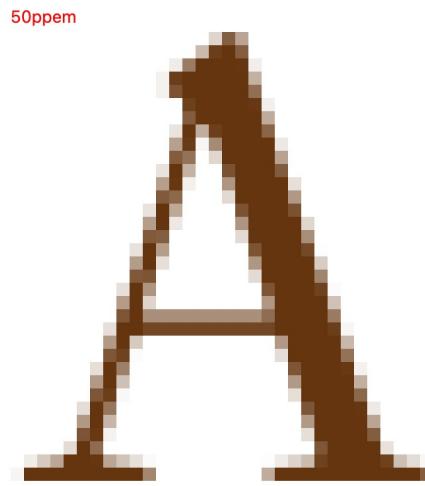
Here you need to know that the shift (blue arrow), align (green arrow) and interpolate (gold arrow) hints can target more than one point. All you need to do is select as many points as you want the hint to target (including the one already targeted) and either click the button for the "Make Set" tool or type K. The result looks like this:





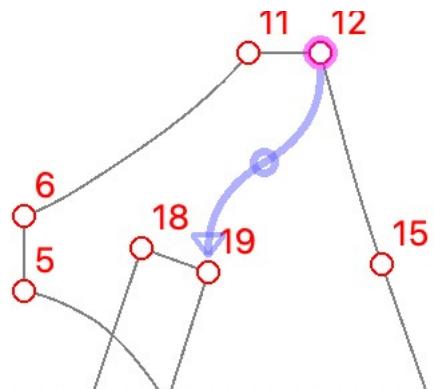
A **set** is a group of highlighted points enclosed in a box of dotted lines. It is always the color of the hint it belongs to and is a part of: if you click on one of the highlighted points or on the box, you will select the whole hint.

The result of our efforts so far looks like this:



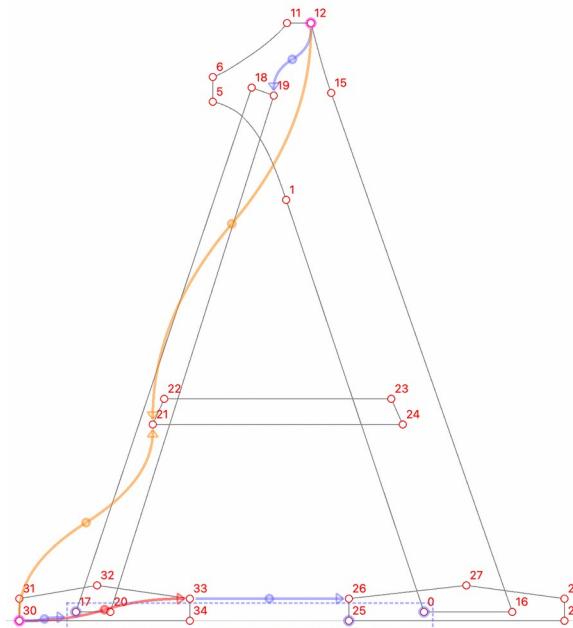
Pretty good, but we've got more to do. The top of the letter looks okay at 50 pixels per em, but we still need to hint it (to make sure it harmonizes with the tops of other capitals), and the cross bar looks fuzzy.

All we need to do at the top of the letter is anchor it with control-value “cap-height-overshoot” and link the anchored point to the top of the left-hand stem with another shift hint:



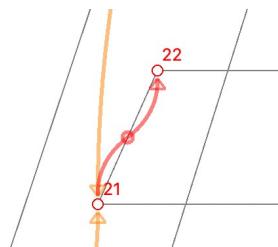
These hints ensure that point 12 always appears at the same y position on the grid, and point 19 is always positioned correctly relative to point 12.

To make the bar less fuzzy, we need, first, an “interpolate” instruction, which positions one or more points relative to two reference points so that their relationship is the same as in the original glyph. To do that, select two touched points and one that has not yet been touched by a hint, and click the “interpolate” tool or type I:

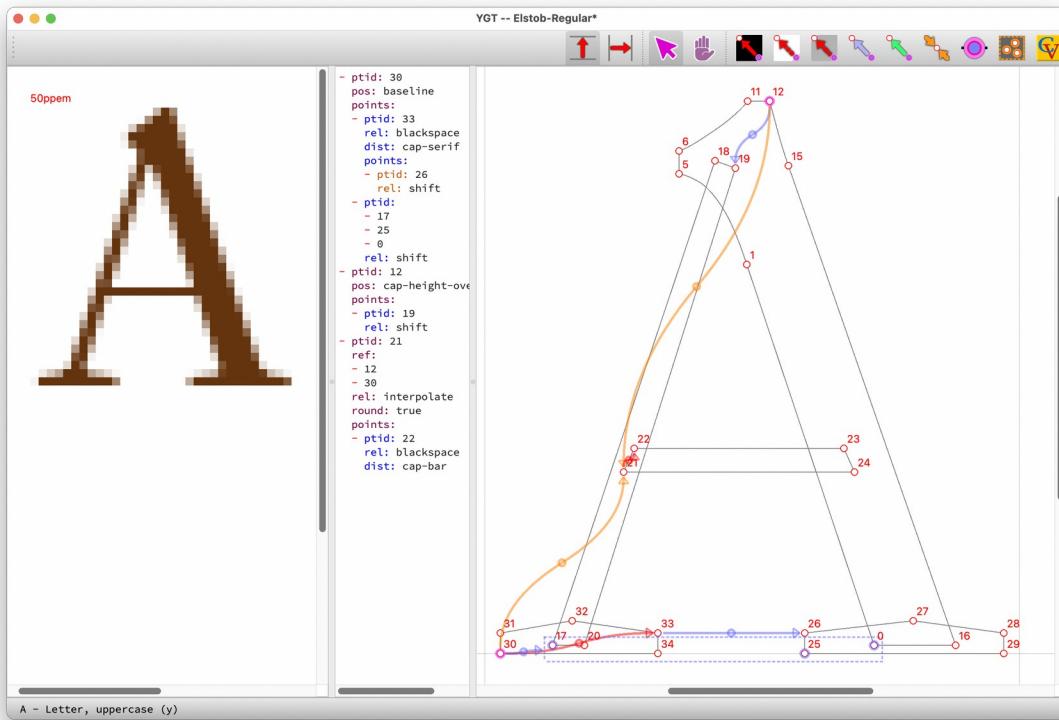


We also have to round point 21 to the grid. To do that, right click on one of the “interpolate” hint’s buttons and choose “Round target point” from the context menu.

Finally, we need to regulate the width of the crossbar. Select one touched and one untouched point (in this case, 21 and 22), click the “Black Space” button or type B. We have added a control value “cap-bar,” and we will assign it to this last hint:



Here is our screen at the end of this process. Notice that the cross bar of the A is no longer fuzzy:



When you type Ctrl-S to save the file, you will save not the font file, but rather one with the same name as the font file, but with the suffix “yaml” instead of “ttf.” When you open the project again, open this YAML file rather than the font file. To save the font file, type Ctrl-E (for “Export font” on the File menu). The font will be saved with the same name as when you opened it, but with “-hinted” added: the original font is not overwritten.

## The YAML source

The source displayed between the preview and editing pane is editable. Once you have learned the simple hinting language used by ygt, you may sometimes find that editing the source is the simplest and surest way to manage a glyph’s hints.

The source editor employs a couple of methods to help you write valid code. First, it validates continuously as you type, displaying the status of the code at the bottom of the ygt window (the program will display more detailed error messages in due course):



Second, the code panel employs a simple scheme of color coding. When a point block (we’ll explain the concept shortly) is nested inside another point block, it changes

color. The hyphen that signals a list in YAML will be red when it is correctly positioned:

```
- ptid: 28
  pos: xheight-overshoot
  points:
    - ptid: 11
      rel: stem
      points:
        - ptid: 20
          rel: grayspace
    - ptid: 0
      pos: lc-baseline-undershoot
      points:
        - ptid: 48
          rel: shift
          points:
            - ptid: 38
              rel: stem
```

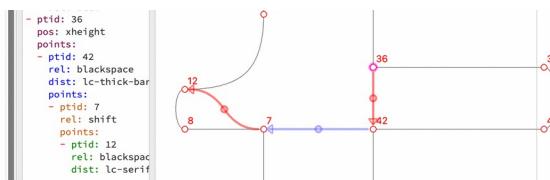
When a keyword in the block is wrong (here “ptic” for “ptid”), it turns black. When a hyphen is wrongly positioned, it turns a different color (and of course, the status bar will signal that the code is “Invalid”):

```
- ptid: 11
  rel: stem
  points:
    - ptic: 20
      rel: grayspace
```

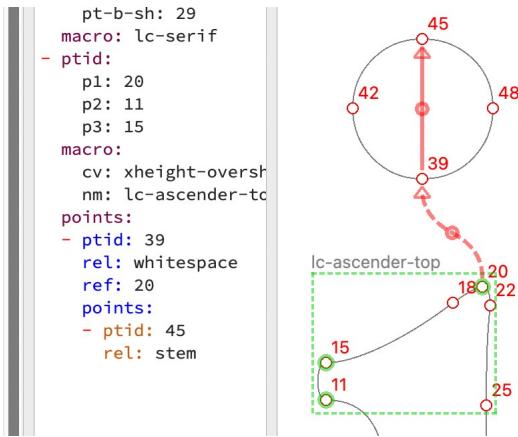
When you have made a change to the source code, type Ctrl-R (Cmd-R on the Mac) or select “Compile” from the “Code” menu. If the code is valid, your changes will be reflected immediately in the editing pane.

Most of the time, though, the program will “compose” the YAML code as you work in the editing pane. YGT will do its best to keep the code in order (for example, making sure that reference points refer only to touched points). If you suspect that the code is out of order (for example, you have added all the hints you need but the preview still doesn’t look right), try selecting “Clean up” from the “Code” menu. Chances are that will fix the problem; if not, you will have to edit the source yourself.

The YAML source consists of a list of point blocks—that is, descriptions of how a point or collection of points should be positioned on the grid. Each point block can contain a list of one or more dependent point blocks. For each dependent point block, one reference point (in the graphical editor, the point at the starting end of a hint arrow) must be the same as one of the parent block’s target (“ptid”) points. For example:



Here is a sequence of four points in the cross bar of the letter f. It begins with an anchored point (36), as sequences always should. Second, point 42 is distance “lc-thick-bar” from 36; third, point 7 is shifted along with 42; fourth, point 12 is distance “lc-serif” from point 7. The reference point (“ref”) for each of the arrow hints is implicit: it is understood to be the same as the parent target (“ptid”), at least when the parent target is unambiguous. When a hint targets more than one point, its child hint must have a “ref” property indicating which of the parent points is its reference.



For example, here the macro “lc-ascender-top” regulates three points (11, 15, 20), so the “whitespace” hint that positions the dot must have a “ref” property indicating that point 39 is to be positioned with reference to point 20 (not 11 or 15). But the hint for point 45 does not need a “ref,” since its reference point is implicitly the “ptid” (39) of its parent.

Here is a list of types of point blocks (excluding functions and macros, which we will discuss later):

## anchor

Most hints have a “rel” property that signals the relationship of the target point to its reference point. An anchor positions a point at a particular place on the grid, not relative to a reference point, and so it has no “rel” property. It can take the following properties:

- |       |  |
|-------|--|
| ptid  | Required. The target point. It must be a single point.   |
| pos   | Optional. Positions the target point at a place indicated by a control value. If this is absent, the point is either moved to the nearest grid line (if “round” is “true”), or simply touched without being moved (if “round” is “false”). |
| round | Optional. The default is “true.” Whether the point should be moved to the nearest grid line after any other positioning.   |

## **blackspace, whitespace, grayspace**

These hints (always represented by red arrows) regulate the distance between a reference point and the target point. With a “blackspace” hint, the distance crosses a space that will be filled with ink. With a “whitespace” hint, the distance crosses a space that will not be filled with ink. A “grayspace” hint crosses a space that is a mix of black and white; or use it when you don’t need to indicate what kind of space the hint is regulating.

ptid	Required. The target point. It must be a single point.
dist	Optional. The name of a “distance” type control value. If it is present, the distance between the reference point and the target point will be regulated by the named control value. If absent, the original distance between the two points will be used.
ref	Required when the parent hint has more than one target point. The point that the target point is positioned relative to.
round	Optional. The default is “true.” Whether to round the distance between “ref” and “ptid.” If “ref” is on a grid line, then “ptid” will be positioned on a grid line too.

## **shift, align**

“Shift” moves the target point by as much as the reference point has been moved. “Align” aligns the target with the reference point on the current axis, so that aligned points will end up in a horizontal line when the axis is “y” and in a vertical line when the axis is “x.” The “align” hint can be tempting, but “shift” is usually the better choice.

ptid	Required. The target can be either a point or a list of points (a “set”).
ref	Required when the parent hint has more than one target point. The point that the target point is positioned relative to.
round	Whether the distance between the reference and target points should be rounded to the grid.

## **interpolate**

Positions one or more points relative to two other points, in such a way that all points have the same relationship to each other as in the original outline. This hint is never the child of another hint.

ptid	Required. The target can be either a point or a list of points (a “set”).
ref	Required. A list of two points.
round	Whether the position of the target point(s) should be rounded to the grid after interpolation.

## Macros and functions

Functions and macros are written in the Xgridfit hinting language and embedded in the YAML source. In the TrueType instruction language, functions are somewhat like functions in other programming languages, but they are numbered rather than named, and they have no formal mechanism for passing parameters. Xgridfit supplies these wants. It also makes available a macro feature. Macros insert code instead of function calls in the program stream, enable conditional compilation, and allow the passing of sets and even bits of code as parameters.

Because functions and macros have to be written in Xgridfit (though you can use Xgridfit’s “command” element to insert raw TrueType instructions if you prefer), writing them involves a learning curve; but a function or macro, once written, may be used (possibly with minor changes) in a number of fonts.

Functions and macros are very much alike from a programmer’s point of view. They consist of a YAML list of parameters and the code itself. Here’s an example of a macro:

```
lc-ascender-top:
  p1:
    type: point
    subtype: target
  p2:
    type: point
    subtype: target
  p3:
    type: point
    subtype: target
  cv:
    type: pos
    val: lc-ascender
  code: |
    <code xmlns="http://xgridfit.sourceforge.net/Xgridfit2">
      <move distance="cv">
        <point num="p1"/>
      <move>
        <point num="p2"/>
        <move distance="lc-serif">
          <point num="p3"/>
        </move>
      </move>
    </move>
  </code>
```

The macro definition lives in a “macros” block in the source file: you can edit it by selecting “Edit macros” from the “Code” menu. The macro begins with its name, followed by a list of parameters and the code itself.

Each parameter declaration needs, at minimum, a name and a type. The name should be something that helps you remember what the parameter is for; the type can be **point**, **pos**, **dist**, **int**, or **float**. A point parameter can also have a subtype, **target** or **ref**. Those who are familiar with the TrueType instruction set will recognize that these types have no use in the compiled code; but they enable functions and macros to behave like other hints in the YAML source.

A parameter declaration can also have the property **val**. This is the default value of the parameter—the value that is used when the parameter is omitted from the function or macro call.

The code itself must be enclosed in `<code></code>` tags, and the opening `<code>` tag must have the Xgridfit namespace declaration, as in the example.

A macro call looks like this:

```
- ptid:
  p1: 10
  p2: 12
  p3: 14
macro:
  nm: lc-ascender-top
  cv: xheight-overshoot
```

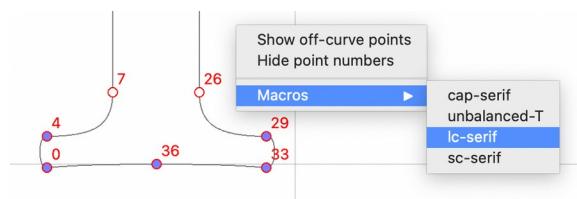
If you find it more convenient, you can also use a JSON-like syntax:

```
{ptid: {p1: 10, p2: 12, p3: 14},
macro: {nm: lc-ascender-top, cv: xheight-overshoot}}
```

YGT will convert this to its default YAML syntax.

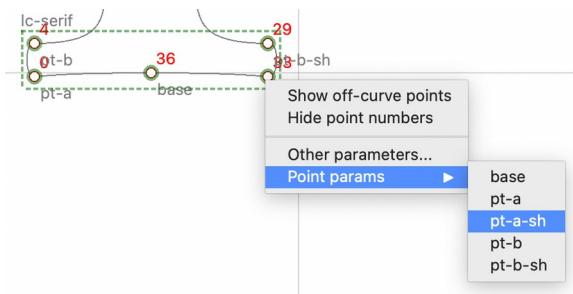
Those of the macro’s parameters that refer to points are passed in the property “`ptid`”; others in the “`macro`” (or “`function`”) property. The latter property must also declare the macro’s name (“`nm`”).

It will usually be more convenient to insert the macro or function call via the graphical hint editor. To do this, select the points that will be affected by the macro or function, then right click to bring up a context menu and select the macro or function by name:

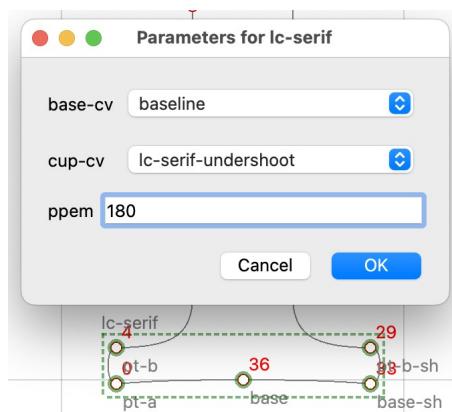


The menu presents only a selection of the macros and functions you have declared, depending on the number of points you selected.

YGT has no way of knowing how to assign your selected points to parameters: they will be applied more or less at random. To sort them out, first select the macro by clicking on its border or one of its points. When the macro is selected, you will see which parameter is associated with which point. Right click on any point that is incorrectly assigned and assign it to the correct parameter by selecting from the “Point params” submenu:



Repeat this operation until all points are correctly assigned. To set other (non-point) parameters, select “Other parameters” from the same menu and adjust them using a dialog box like this one:



## Point designators

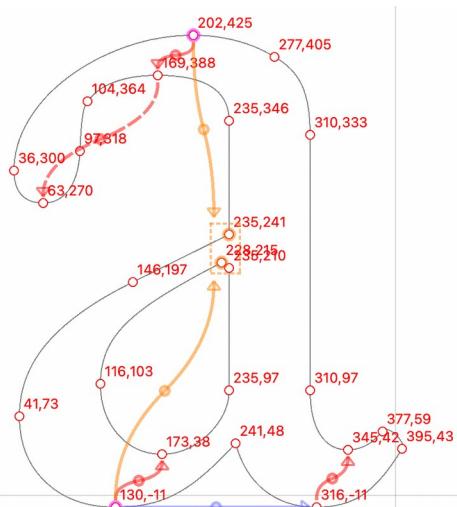
In a TrueType font, the points that define a glyph’s outlines are numbered from zero, and TrueType hints refer to points by these indices. You’ll have little need to think about point numbers unless you edit the YAML code yourself, but there is one thing worth knowing—namely, that YGT can refer to points not only by number, but also by their coordinates. In the YAML code, a sequence like {548;425} means “the point at coordinates 548,425.”

The reason to use coordinates instead of indices in your code is that index numbers are unstable, especially if you use more than one program, for different programs will number points differently when generating fonts. Further, if you make even a minor change in a glyph's outlines after you have hinted it, the point numbers will change next time you generate the font, even if you always use the same font editor. The result will be invalid hints and messed up glyphs.

But software that generates fonts ordinarily does not alter the coordinates of on-curve points—so if you hint only on-curve points and designate those points by their coordinates it won't matter if you generate the font with, say, Glyphs or fontmake.

Display coordinates rather than indices on screen by selecting “Point coordinates” from the “View menu.” When “Point coordinates” is selected (even if point numbers are hidden), hints added to the YAML code will refer to points by their coordinates. You can also change all indices in the current block of code to coordinates (and vice versa) by selecting “Indices to coords” or “Coords to indices” from the “Code” menu. It is a good idea, if you have been working with point indices (which are admittedly more straightforward than coordinates) to select “Indices to coords” after you are finished hinting a glyph.

Points can also be named in ygt, but this feature is not yet complete.



## Control Values

Control values are of two kinds: **position** (pos) and **distance** (dist). Position control values specify the location of a point on the current axis (x or y); distance control values specify the distance of one point from another—again on the current axis.

You can manually edit the font's control values by choosing “Edit cvt” from the “Code” menu (the process is described in greater detail below). But the easiest way to add control values is via the “CV” button on the toolbar. To use it, select one point for a “pos” control value or two points for a “dist” control value, then press the button or type “C.” This dialog box will appear:





The box is already filled in with most essential information: the position or distance in the “val” box; the type of cv (“pos” or “dist”), and the axis (“x” or “y”). The only information you have to supply is the cv’s name (it should contain no spaces, but it can contain numbers, hyphens and underscores: make the name descriptive but not verbose).

Much of the information you supply in this box is used to filter the list of control values when you need to choose one from a menu. For example, when the current axis is “y,” you only see control values for the “y” axis. The more information you supply, the fewer choices you’ll confront.

- |        |   |
|--------|---|
| type   | Whether the cv is for positions or distances.   |
| axis   | Which axis the cv is intended for.  |
| cat    | The unicode category of the cv (e.g. “Letter, uppercase”). If supplied, the cv will only be offered for a character whose category matches the cv (but you can change the category of any glyph, or supply categories where they are missing, by selecting “Set category” from the context menu). |
| suffix | Character names often have suffixes: for example, small caps commonly have names ending in “.sc” (e.g. “a.sc” for “small cap a”). If a suffix is supplied here, the cv will only apply to characters whose names have that suffix.  |
| color  | This property is not for filtering control values, but rather for affecting the behavior of hints. If a hint’s color (black, white, or gray) is not otherwise specified, the color of the cv is used, if given. Most of the time, there is little need to set this property.                      |

## Categories

Every character with a Unicode code point also has a category, assigned by Unicode. To see them, choose “Set category” from the context menu. Ygt uses these categories as filters for control values: when you assign a category to a control value, that control value only appears on the menu for glyphs with matching categories (but any control value can be entered in the editing pane).

You can override these categories whenever it is useful to do so. (The categories you choose are not recorded in the font, and they have no effect at all except on the selection of control values in the menus.) For example, the “yen” sign (U+00A5) is categorized as “Symbol, currency,” but it is a capital Y with bars, and so it is useful to apply to it control values for capital letters. Simply select “Letter, uppercase” from the “Set category” menu, and all capital control values will be available for that glyph. Likewise, if you use the Unicode Private Use Area, you’ll find that Unicode classifies all the glyphs there as “Other, private use”—not very useful! But you can assign any category you like to Private Use glyphs.

Some groupings of characters do not have Unicode categories—small caps, for example. For these, use the “suffix” property of your control values.

## Variable fonts

A variable font with TrueType hinting should have a **cvar** table to allow the rasterizer to vary control values as the font’s axes change. The cvar table normally has several sections for various combinations of regions of the font’s design space. These will often correspond to your font’s masters. To take a simple case, if your font has a weight axis running from “ExtraLight” (200) to “Bold” (700), its “regions” will have entries for one axis each: one for weights 200–400 and one for weights 400–700. The values that define each region are generally normalized so that the “Normal” weight is zero and the extremes on that axis are -1 and 1. The cvar table might look something like this:

```
- regions:
- tag: wght
  bot: -1.0
  peak: -1.0
  top: 0.0
vals:
- nm: cap-serif
  val: 31
- nm: cap-bar
  val: 39
- nm: cap-round-stem
  val: 31
```

```
- nm: lc-serif
  val: 21
  . . .
- regions:
  - tag: wdth
    bot: -1.0
    peak: -1.0
    top: 0.0
  vals:
  - nm: cap-serif
    val: 34
  - nm: num-circle
    val: 51
  . . .
```

For each collection of regions, you need a list of values, each identified by a name (which must match one name in the cvt) and a value. List only control values that differ from the ones in the cvt. List alternative values, not differences from the cvt values.

Test how your hinting behaves at various points on your axes by selecting different instances from the “Preview” menu. For example, here are previews of the Junicode **g** for the Regular and SemiExpanded Bold instances:

