Creating a Plot Graph with Braille Characters

January 2019 By An7ar35

Contents

List of Algorithms 1

1 Introduction 2

2 Graph Plotting 3
- 2.1 Scale and Aspect Ratio . . . 3
- 2.2 Axes and Ticks . . . 4
  - 2.2.1 Axes Centre . . . 4
  - 2.2.2 Axes Ticks . . . 4
- 2.3 Scaling Data Points . . . 5
  - 2.3.1 Generic Point scaling . . . 5
  - 2.3.2 Amortised Point scaling . . . 6

3 Labelling 7
- 3.1 Pre-labelling concerns . . . 7
- 3.2 X-Axis tick labelling . . . 8
- 3.3 Y-Axis tick labelling . . . 10

List of Algorithms

1 Fixed aspect-ratio scale . . . 3
2 Tick pixel spacing . . . 5
3 Generic point scaling . . . 5
4 Tick-amortised point scaling . . . 6
5 Generic label size calculation . . . 7
6 X-Axis character separation count . . . 8
7 X-Axis tick skipping . . . 8
8 X-Axis tick culling . . . 9
9 X-Axis label writing . . . 10

1 Introduction

Prerequisite: Creating bitmaps using braille characters

Since pixelated graphics can be created with the help of Unicode braille characters, why not make graphs?

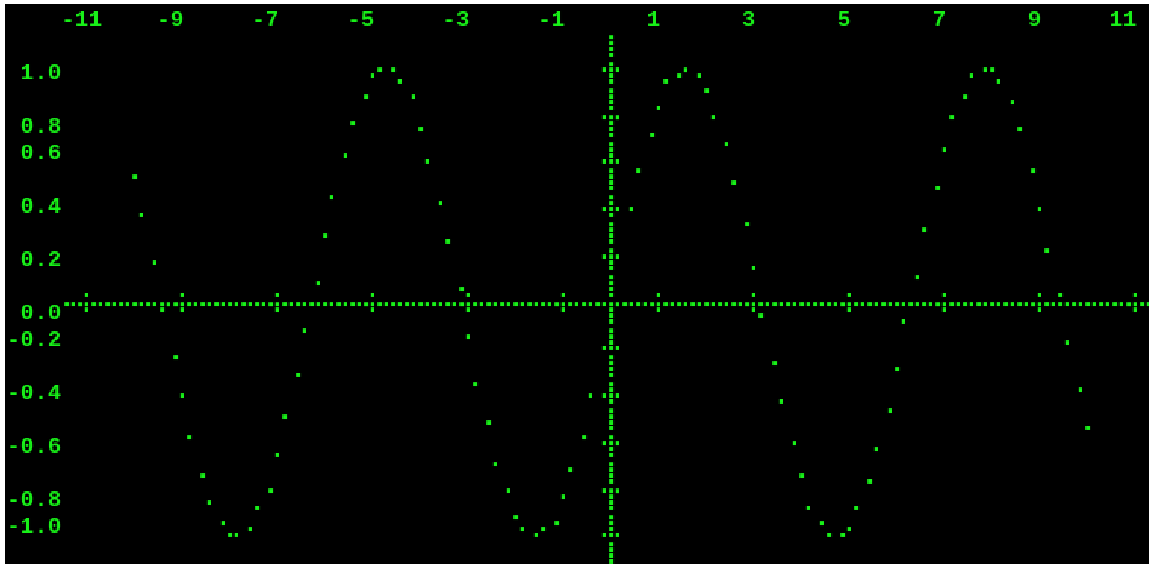


Figure 1: Tasty terminal graph...

This document details the more interesting aspects and problems encountered during development of the EADlib `eadlib::cli::BraillePlot` C++ library component.

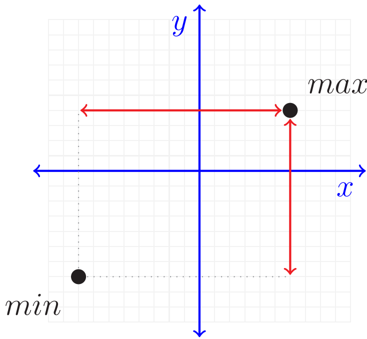
2 Graph Plotting

2.1 Scale and Aspect Ratio

In order to calculate scaling, the drawing area (or canvas) size in pixels and the maximum (x, y) lengths used in the original data point set are required.

The drawing (x, y) sizes can be simply calculated by multiplying the number of braille characters in a line and the number of lines by their pixel capacities: $canvas_x = W \times 2$ and $canvas_y = H \times 4$

2 by 4 pixel block



The graph size can be obtained by adding the largest point value of the axis with its absolute (`abs()`) smallest: $size_x = max_x + |min_x|$ and $size_y = max_y + |min_y|$. With that we have the max length required on both x and y.

Calculating the individual X and Y ratios is then trivial:

$$ratio_x = \frac{canvas_x}{size_x} \qquad ratio_y = \frac{canvas_y}{size_y}$$

The challenge here is when the aspect ratio must be maintained when scaling. This means that whether the graph is made larger or smaller, it must be done uniformly on both axes.

In the case where the canvas is larger than the graph size on both axes then, in order to maintain the aspect ratio, the up-scaling is bound to the smallest ratio. If we didn't, the graph would end up being bigger on one of the axis than the canvas allows [lines 1-2].

When the graph is larger on both axes than the canvas downscaling is bound to the largest ratio to, again, make sure that the totality of the graph fits inside the given canvas [lines 3-4].

Finally, when only one axis of the graph actually fits within the canvas, we downscale based on the axis that doesn't [lines 5-9].

The final line [10] just applies the scaling to the second axis to maintain the fixed aspect ratio.

Algorithm 1: Fixed aspect-ratio scale

```
1  if ratio_x ≥ 0 AND ratio_y ≥ 0 then
2  |   scale_x ← min( ratio_x, ratio_y );
3  else if ratio_x < 0 AND ratio_y < 0 then
4  |   scale_x ← max( ratio_x, ratio_y );
5  else
6  |   if ratio_x < 0 then
7  |   |   scale_x ← ratio_x;
8  |   else
9  |   |   scale_x ← ratio_y;
10 scale_y ← scale_x;
```

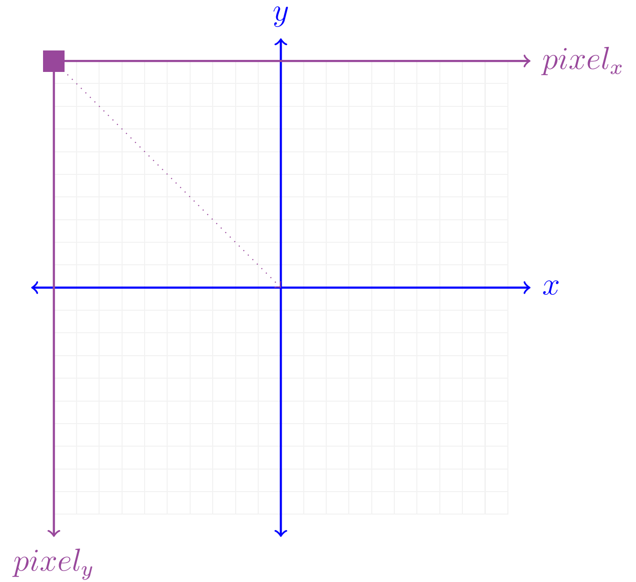
2.2 Axes and Ticks

2.2.1 Axes Centre

Since the canvas' pixel coordinates start from the top left corner at $(0,0)$, the graph axes and data points need to be shifted accordingly. Before that, the axes centre on the pixel grid must be calculated.

$$centre_x = \left(\frac{|min_x|}{size_x} \times (canvas_x - 1\right) + 1 \qquad centre_y = \left(\frac{max_y}{size_y} \times (canvas_y - 1)\right) + 1$$

The equations (above) determine in effect how far the $(0,0)$ graph point is from the $(0,0)$ pixel point. This is why the ratio $(-x,+y)$ area of the graph is calculated and then multiplied by the canvas (x,y) pixel sizes. The -1 is to account for the pixel lines that will be used by the graph's drawn axes.

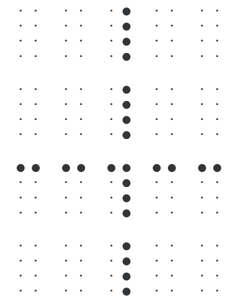


The 1 added at the end is for the pixel wide margin around the canvas which is placed to allow for rounding errors when the data points are scaled. In the event a data point is near the edge of the graph box, a rounding to the nearest pixel location might lead to an out-of-range access to the bitmap data-structure (e.g.: $(19,80)$ on a 40×80 pixels canvas).

2.2.2 Axes Ticks

For tick indicators in the axes some accommodations must be made as the spacing between individual ticks will vary whether the graph has a fixed aspect ratio or not.

The value of intervals needs to be deduced as well as limiting the total number of ticks to be draw on the axes (tick count < pixel length of axis).



Pixelated centre

For some breathing space, the algorithm attempts to have an absolute maximum of 1 tick for every 4 pixels. i.e. every 2 character block on the X-axis and 1 character block on the Y-axis.

The Y-axis restriction is also relevant for labelling as there can only be one label maximum on each lines.

Starting from the given interval values at the start, we check whether or not they would satisfy the above condition. If not they are multiplied by 2 until the condition is met. e.g.: an interval value of 5.0 that has a calculated tick spacing of 3 pixels would end up as 10.0 at 6 pixel intervals.

For fixed-ratio scaling, the smallest of the (x, y) tick spacing would be used on both axes.

Algorithm 2: Tick pixel spacing

Input: $canvas_size$: Canvas XY sizes
Input: $graph_size$: Graph XY sizes
Input: $interval_value$: Tentative interval values
Output: $pixel_spacing$: Tick pixel spacings

1 $tick_count_x \leftarrow \frac{graph_size_x}{interval_value_x}$;
2 **while** $tick_count_x > \left\lfloor \frac{canvas_size_x}{4} \right\rfloor$ **do**
3 $\quad interval_value_x \leftarrow interval_value_x \times 2$;
4 $\quad tick_count_x \leftarrow \frac{graph_size_x}{interval_value_x}$;
5 $tick_count_y \leftarrow \frac{graph_size_y}{interval_value_y}$;
6 **while** $tick_count_x > \left\lfloor \frac{canvas_size_x}{4} \right\rfloor$ **do**
7 $\quad interval_value_y \leftarrow interval_value_y \times 2$;
8 $\quad tick_count_y \leftarrow \frac{graph_size_y}{interval_value_y}$;
9 $pixel_spacing_x \leftarrow \left\lfloor \frac{canvas_size_x}{tick_count_x} \right\rfloor$;
10 $pixel_spacing_y \leftarrow \left\lfloor \frac{canvas_size_y}{tick_count_y} \right\rfloor$;

2.3 Scaling Data Points

2.3.1 Generic Point scaling

For simple scaling each data points is just multiplied by the scale factor and then shifted by the position of the axes pixel centre coordinates on the canvas.

Algorithm 3: Generic point scaling

Input: $point$: Data point (x, y)
Input: $axis_center$: Axis center on canvas
Input: $scale$: Scale
Output: p: Scaled point

1 $p \leftarrow point$;
2 $p_x \leftarrow p_x \times scale_x$;
3 $p_y \leftarrow p_y \times scale_y$;
4 **if** $p_x \geq 0$ **then**
5 $\quad p_x \leftarrow axis_center_x + p_x$;
6 **else**
7 $\quad p_x \leftarrow axis_center_x - |p_x|$;
8 **if** $p_y \geq 0$ **then**
9 $\quad p_y \leftarrow axis_center_y - p_y$;
10 **else**
11 $\quad p_y \leftarrow axis_center_y + |p_y|$;

2.3.2 Amortised Point scaling

With ticks present, simple scaling leads to inaccuracies in the display of the points in respect to the tick values. This is due to:

- rounding errors introduced when casting points to the nearest pixel and
- possible shifts introduced by making the ticks equidistant from each other.

To accurately place the points in respect to the tick values, tick-based amortisation is used.

In short; the nearest tick below (positive) or above (negative) the data point value is first worked out on both x and y (lines 1-5, 7-11) then the extra pixels left to reach the data point (lines 7, 12).

The number of ticks is multiplied by the tick pixel interval then the extra pixels are added to get the relative pixel position of the data point (lines 13-14).

Finally the (x, y) of the amortised point is shifted by the coordinates of the axis pixel centre to get its final pixel position on the drawing canvas (lines 15-22).

Algorithm 4: Tick-amortised point scaling

Input: $interval_value$: Tentative interval values
Input: $axis_center$: Axis center on canvas
Input: $pixel_spacing$: Tick pixel spacings
Input: $point$: Data point (x, y)
Output: p: Scaled point

1 $tick_count_x \leftarrow \frac{point_x}{interval_value_x}$;
2 $whole_ticks_x \leftarrow tick_count_x > 0$
3 ? $\lfloor tick_count_x \rfloor$
4 : $\lceil tick_count_x \rceil$;
5 $tick_pixels_x \leftarrow$
$(tick_count_x - whole_ticks_x) \times pixel_spacing_x$;
6 $tick_count_y \leftarrow \frac{point_y}{interval_value_y}$;
7 $whole_ticks_y \leftarrow tick_count_y > 0$
8 ? $\lfloor tick_count_y \rfloor$
9 : $\lceil tick_count_y \rceil$;
10 $tick_pixels_y \leftarrow$
$(tick_count_y - whole_ticks_y) \times pixel_spacing_y$;
11 $pixel_count_x \leftarrow$
$(whole_ticks_x \times pixel_spacing_x) + tick_pixels_x$;
12 $pixel_count_y \leftarrow$
$(whole_ticks_y \times pixel_spacing_y) + tick_pixels_y$;
13 **if** $pixel_count_x \geq 0$ **then**
14 $\quad p_x \leftarrow axis_center_x + pixel_count_x$;
15 **else**
16 $\quad p_x \leftarrow axis_center_x - |pixel_count_x|$;
17 **if** $pixel_count_y \geq 0$ **then**
18 $\quad p_y \leftarrow axis_center_y - pixel_count_y$;
19 **else**
20 $\quad p_y \leftarrow axis_center_y + |pixel_count_y|$;

3 Labelling

3.1 Pre-labelling concerns

The list of ticks and their corresponding axis coordinate need to be sorted by ascending order on the canvas axes $(0..N)$: left-to-right for the X-axis and top-to-bottom for the Y-axis. This helps on placing ticks consecutively and avoid jumping back and forth whilst iterating the labelled canvas.

Globally, there are 3 pieces of information needed to uniformly place labels on any given axis:

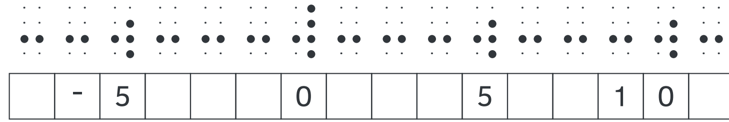
1. maximum characteristic size in all tick values (line 5),
2. mantissa size of the given tick interval value (line 4),
3. total character length of the longest tick value (lines 6-9).

Algorithm 5: Generic label size calculation

Input: $interval_value$: Tick interval value
Input: max_neg_value: Maximum negative tick value
Input: max_pos_value: Maximum positive tick value
Output: $characteristic$: Reserved character length for characteristic
Output: $mantissa$: Reserved character length for mantissa
Output: $size$: Total reserved length for label

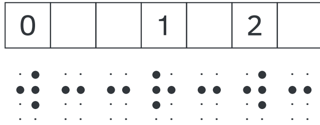
```
1 max_neg_c ← characteristicLength( max_neg_value );
2 max_pos_c ← characteristicLength( max_pos_value );
3 interval_c ← characteristicLength( interval_value );
4 mantissa ← mantissaLength( interval_value );
5 characteristic ← max( max( max_neg_c, max_pos_c ), interval_c ) );
6 size ← (mantissa > 0 ? characteristic + mantissa + 1 : characteristic );
```

3.2 X-Axis tick labelling



The labelling on this axis is the least trivial. First, any ticks whose label may overlap of touch must be removed from consideration in a distributed manner. e.g.: if tick '2.0' overlaps with '2.5' then every other ticks must be dropped to maintain symmetry.

In the event of odd tick intervals the space between ticks is not uniform.



e.g.: (above) The interval is 5 pixels but the number of characters before the next tick differ at 0 and 1 with 3 and 2 blocks respectively.

Algorithm 6: X-Axis character separation count

Input: $tick_interval$: Pixel interval for X-axis ticks

Input: $axis_center$: Axes pixel centre

Output: $char_count$: Worst number of characters between and inclusive of 2 ticks

1 $char_count \leftarrow \frac{tick_interval}{2}$;

2 **if** $tick_interval \bmod 2 > 0$ *and* $axis_center_x \bmod 2 \equiv 0$ **then**

3 $\quad char_count \leftarrow char_count + 1$;

With this in mind; we must find the worst spacing possible (see #6 above) to base the tick skipping algorithm (#7) on to make sure that no labels touch each over or overlap.

To get the number of ticks to skip between the ones to keep, the number of character block needed is checked against the label size calculated in the "Generic label size calculation" algorithm (#5) and the skip count is incremented on each failed iteration check (see right).

Algorithm 7: X-Axis tick skipping

Input: $size$: generic label size

Output: $skip_count$: Number of ticks to skip between those to keep

1 $skip_count \leftarrow 0$;

2 $char_blocks \leftarrow char_spacing$;

3 **while** $size > char_blocks - 2$ **do**

4 $\quad char_blocks \leftarrow char_blocks \times 2$;

5 $\quad skip_count \leftarrow skip_count + 1$;

Instead of the remove-as-we-go approach a pre-culling is much a easier to implement for maintaining the labelling symmetry.

The first step (lines 1-5) gets us to the '0' tick on the axis using a stack to push any other ticks that are prior to it (i.e.: negative valued ticks since the list is sorted based on the axis coordinates).

From there we iterate to the end of the list whilst removing on-the-fly (lines 6-13) any ticks that overlap based on the $skip_count$ calculated in the "X-Axis tick skipping" algorithm (#7).

The same is done in reverse from the '0' tick by using the stored negative ticks on the $negative_values$ stack.

At the end, the only ticks remaining in the list are those that do not overlap both in their positive and negative versions thus keeping labelling symmetrical.

Algorithm 8: X-Axis tick culling

Input: $tick_list$: List of ticks comprised of their $\{value$ and $coordinate\}$

Output: $skip_count$: Number of ticks to skip between those to keep

```
1  tick_iterator ← tick_list.begin();
2  negative_values ← Stack();
3  while tick_iterator ≠ tick_list.end() and
     tick_iterator.value < 0 do
4      negative_values.push( tick_iterator );
5      tick_iterator++;
6  erase_counter ← 0;
7  while tick_iterator ≠ tick_list.end() do
8      if skip_count > 0 and erase_counter < skip_count
         then
9          tick_iterator ← tick_list.erase( tick_iterator );
10         erase_counter ← erase_counter + 1;
11     else
12         erase_counter ← 0;
13         tick_iterator++;
14 erase_counter ← 1;
15 while !negative_values.empty() do
16     if skip_count > 0 and erase_counter < skip_count
         then
17         tick_list.erase( negative_values.top() );
18         erase_counter ← erase_counter + 1;
19     else
20         erase_counter ← 0;
21     negative_values.pop();
```

Finally, the labels are written to the X-Axis label row. The "X-Axis label writing" algorithm (#9) is used on each consecutive ticks in the list. Each character block is added one-by-one whether a space ($0x20$) or a character in a tick value label.

Algorithm 9: X-Axis label writing

Input: $tick$: Tick with $value$ and $coordinate$

Input: $label_mantissa$: Global mantissa character length for the X-Axis

Input: row: Labelling row

1. $characteristic \leftarrow$ lengthInt($tick.value$);
2. $mantissa \leftarrow$ min(lengthFrac($tick.value$), $label_mantissa$);
3. $str_size \leftarrow (mantissa > 0$? $characteristic + mantissa + 1 : characteristic)$;
4. $str_offset \leftarrow \left(str_size > 1 \text{ ? } \left(str_size \bmod 2 > 0 \text{ ? } \frac{str_size - 1}{2} : \frac{str_size}{2}\right) : 0\right)$;
5. $tick_index \leftarrow \frac{tick.coordinate_x + 1}{2} + (tick.coordinate_x + 1 \bmod 2?1 : 0)$;
6. $start_index \leftarrow (str_offset \geq tick_index$? $0 : tick_index - str_offset - 1)$;
7. $value_str \leftarrow$ toString($tick.value$);
8. **for** $i \leftarrow row.$size(); $i < start_index$; i++ **do**
9. $\quad$ row.add($0x20$);
10. **for** $i \leftarrow 0$; $i < str_size$; i++ **do**
11. $\quad$ row.add($value$.at(i));

3.3 Y-Axis tick labelling

Labelling on the Y-axis is incrementally done based on the order of the tick list (from smallest y position to largest). As all labels have a predefined default size based on the "generic label size calculation" algorithm (#5) which is applied to the axis.

The only interesting aspect is the left-side padding which is calculated on a per-label basis so that all label values are aligned horizontally with each other.

Since all labels will have the same mantissa length (if any), the padding is worked out from the characteristic and its difference in length with the generic label's one.