Self-Adjusting Legends for Proportional Symbol Maps

Bernhard Jenny, Ernst Hutzler, and Lorenz Hurni

Institute of Cartography / ETH Zurich / Switzerland

Introduction

Proportional symbol maps are frequently used to represent numerical data. Commonly used symbols are squares or circles that are area-proportional to the numerical values. Proportional-symbol mapping is included in most geographic information systems (GIS) and is frequently used in digital atlases or Web mapping engines. Graduated symbols are simple for the map author to generate and easy for the reader to understand. However, if the reader is to accurately associate the symbol size on the map with the underlying numerical value, a carefully designed legend is essential. To link a symbol with a numerical value, the map reader mentally positions the symbol in the legend and estimates the value from the symbols shown in the legend. To reduce the associated estimation error, a legend should have the following qualities:

1. The legend should show symbols for both the maximum and the minimum values represented in the map, allowing the map reader to quickly grasp the range of values mapped.
2. The legend should contain three or more symbols, in order to ensure accurate value estimates (Cox 1976; Chang 1980; Dent 1999, 181).
3. The values represented should be round numbers, ending with one or more zeros, because round numbers are easier to remember and to calculate (Slocum and others 2009, 315). Non-round numbers are acceptable if they are one-half or one-quarter of 10 raised to an integer power (e.g., 5, 25, or 50).
4. Intermediate symbols should be visually evenly distributed. Placing intermediate values at exactly equal intervals results in a visually uneven distribution. Figure 1 shows an equal-intervals legend (left) compared to a visually balanced legend (right); the equal-intervals legend concentrates symbols in the upper half of the legend, whereas the visually balanced legend distributes symbols more evenly over the legend space.

It should be noted that these qualities are not absolute requirements. For example, the minimum value may be too small to be shown, the range of values mapped may not contain round numbers, or intermediate values with an application-specific meaning may have to be added. One could also argue that the legend should show symbols for the minimum and maximum values even if they are not round numbers. For example, if the maximum value is 105, the largest symbol in the legend could be 100; however, we believe that the minimum and maximum values should be shown if possible, because (1) readers can interpolate between circle sizes, but have difficulties extrapolating (Dobson 1974); and (2) the legend is the appropriate place to indicate the range of values mapped.

Self-Adjusting Legends

The presented method is applicable both to mathematical scaling, where the size of point symbols is in direct
proportion to the data, and to perceptual scaling that compensates for the common underestimation of the size of larger symbols (for an overview of perceptual scaling see Slocum and others 2009, 309). However, the method is not suitable for range-graded scaling that groups data into classes and represents each class with a different-sized symbol. Self-adjusting legends can be constructed for circles, squares, spheres, and other geometric symbols, as well as for pictographic symbols.

Self-adjusting legends determine the symbols to be drawn using two steps. In the first step, a set of candidate values is generated between the minimum and the maximum extent of the data range. The second step removes values from the set of candidate values if they would result in symbols too close to one another. The following section gives details on these two processing steps.

The first step computes candidate values for drawing intermediate legend symbols between the minimum and the maximum of all the values represented on the map, using the concept of base values to compute candidate values. A base value must be between 1 and 10 and is used to derive multiple candidate values by multiplying by increasing decimal powers, as expressed in the following equation:

\[ v_i = b \times 10^i \]  

where

- \( v_i \) is a candidate value,
- \( b \) is a base value,
- \( i \) is an increasing integer number.

For example, the base value \( b = 5 \), and \( i \) varying between –2 and 3, produce the following candidate values \( v_i \): 0.05, 0.5, 5, 50, 500. By adjusting the range of \( i \), we can generate symbols for very small or very large minimum and maximum values.

Self-adjusting legends use more than one base value. The proposed default base values are 5, 2.5, and 1, which generate visually equally distributed values at round intervals and at halves or quarters of 10 raised to an integer power (see Figure 1, right). To generate the complete set of candidate values, first the range of \( i \) is computed from the maximum and minimum values mapped, then equation (1) above is applied to the three base values (5, 2.5, 1) to compute the candidate values \( (v_i) \). Appendix 1 provides pseudocode illustrating the details of this algorithm.

Figure 2 (left) shows the legend for a data range between 0.4 and 105. The default base values 5, 2.5, 1 are applied, resulting in symbols at 0.4, 0.5, 1, 2.5, 5, 10, 25, 50, 100, and 105 (including the minimum and the maximum). The legend in Figure 2 (left) draws circles for all candidate values computed by the first step, and therefore it shows obvious graphical problems at the upper and lower boundaries of the data range. These graphical conflicts are resolved in the second step.

The second processing step filters the candidate values generated in the first step and removes values that would result in symbols lying too close to one another. It computes the geometrical distance between each pair of neighbouring symbols and compares the distance to a minimum vertical difference \( d_{\text{min}} \). If the distance between two symbols is smaller than \( d_{\text{min}} \), the smaller of the two symbols is removed from the set.

The difference \( d_{\text{min}} \) is linked to the type size \( h \) used to label the legend, computed as

\[ d_{\text{min}} = f \times h \]  

The distance factor \( f \) has a default value of 1.25. This value was found to generate acceptable distributions for the majority of value ranges and type sizes.

In filtering the values determined in the first processing step, the maximum value is never removed. The minimum value is removed only if its symbol is too small (i.e., if its height is smaller than \( d_{\text{min}} \)). Intermediate values are filtered using the \( d_{\text{min}} \) criterion, from larger to smaller values. The smaller a value is, the more likely its symbol or label is to conflict with neighbouring ones, and therefore the more likely it is to be removed. Appendix 2 provides pseudocode with more details for this second algorithm. The result of this second step is a final set of values that results in a legend as shown in Figure 2 (right).

Both the first value-generating step and the second filtering step are required for automatic generation of a legend that meets the requirements listed in the previous section. The two steps could be merged into a single, more compact algorithm; however, computing the legend values in two steps not only improves code readability but also provides additional flexibility by allowing extra values to be inserted into the legend. For example, a self-adjusting legend for an air-pollution map generates candidate values using the first step; legal pollution thresholds are then added to the candidate values; and, finally, the filtering algorithm of the second step solves graphical conflicts, retaining the threshold values if space permits.

The range of the data is the only required parameter for computing intermediate legend symbols. Internally, the two processing steps use two additional parameters: the base values and the minimum difference \( d_{\text{min}} \), which is

![Figure 2](image-url)

**Figure 2.** Intermediate legend drawn with all candidate values (left) and final legend with filtered values (right).
derived from the type size used to label the legend. Most mapping applications can use the default values for these parameters (5, 2.5, 1 for the base values and 1.25 for the scale factor $f$). Specialized mapping software, however, may offer the user the possibility of customizing these values if required.

**Conclusion**

Self-adjusting legends, as presented in this article, have a series of advantages over other construction methods. Indeed, many alternatives exist, such as equal intervals, percentile intervals, and more advanced methods for data classification – for example, the Jenks-Caspall algorithm (Jenks and Caspall 1971, proposed by Dobson 1974) for graduated-symbol legends. Compared to these and other possible methods, self-adjusting legends have the following advantages: no user interaction is required for defining intermediate values or for rounding numbers; intermediate values are easy to remember and to calculate; any data range can be handled, including very large or very small minimum and maximum values (see Figure 3); symbols are visually equally distributed; the number of intermediate symbols is automatically adjusted for the available space; the method applies to any graphical shape; and the method is applicable to both nested and linearly distributed legends.

A Java applet is available for interactive testing of self-adjusting legends for graduated circle and square maps at http://jenny.cartography.ch/legend/. This page also links to the source code for the applet.

The automatic method presented here could be added to software applications that generate proportional symbol maps from user-defined data, such as desktop GIS or online mapping systems. It would also be useful for digital atlases that generate a number of maps from statistical data, such that the manual definition of legends would be too time consuming for the author of the atlas.

**Acknowledgements**

The authors wish to thank the two anonymous reviewers for their valuable comments.

**References**


**Appendix 1: Pseudocode Computing All Candidate Values**

The function `computeCandidateValues()` takes the minimum and the maximum value as input. It holds an array of base values that determine the generated values, and returns an array of values, in decreasing order, which are candidates for inclusion in a proportional symbol map.

Note: this pseudocode is not an executable program but is simplified for human reading. Comments are preceded by `//`.

```plaintext
function computeCandidateValues(min, max)
    // array of base values in decreasing order in the range [1,10)
    bases = [5.0, 2.5, 1.0]
    // an index into the bases array
    baseID = 0
    // array that will hold the candidate values
    values = []
    // scale the minimum and maximum such that the minimum
    // is >1
    scale = 1.0 // scale factor
    min_s = min // scaled minimum
```
max_s = max // scaled maximum
while (min_s < 1)
    min_s *= 10
    max_s *= 10
    scale /= 10
// compute the number of digits of the integral part of the
// scaled maximum
ndigits = floor(log10(max_s))
// find the index into the bases array for the first limit smaller
// than max_s
for (i from 0 to bases.length-1)
    if ((max_s / 10^ndigits) >= bases[i])
        baseID = i
        leave for loop
loop
    v = bases[baseID] * 10^ndigits
    // stop the loop if the value is smaller than min_s
    if (v <= min_s)
        leave loop
// otherwise store v in the values array
    values.add(v / scale)
// switch to the next base
    baseID++
    if (baseID == bases.length)
        baseID = 0
        ndigits = ndigits - 1
return values

function filterValues(values, dmin)
    // array that will hold the filtered values
    filteredValues = []
    // add the maximum value
    filteredValues.add(values[0])
    // remember the height of the previously added value
    previousHeight = valueToSymbolHeight(values[0])
    // find the height and value of the smallest acceptable
    // symbol
    lastHeight = 0
    lastValueID = values.length-1
    while (lastValueID >= 0)
        lastHeight = valueToSymbolHeight(values[lastValueID])
        if (lastHeight > dmin)
            leave for loop
        -lastValueID
    // loop over all values that are large enough
    for (limitID from 1 to lastValueID)
        v = limits[limitID]
        // compute the height of the symbol
        h = valueToSymbolHeight(v)
        // do not draw the symbol if it is too close to the
        // smallest symbol (but is not the smallest limit itself)
        if (((h - lastHeight) < dmin) and (limitID != lastValueID))
            continue with next loop
        // do not draw the symbol if it is too close to the previously
        // drawn symbol
        if ((previousHeight - h) < dmin)
            continue with next loop
        filteredValues.add(v)
        // remember the height of the last drawn symbol
        previousHeight = h
return filteredValues

function valueToSymbolHeight(value)
    // compute the diameter of a circle (must be adjusted for
    // other symbol shapes or for perceptual scaling)
    // f is a scale factor to adjust the size of the circle symbols
    return 2 * sqrt(f * value/π)