Improving MuseScore's Horizontal Spacing

5 March 2020
Barnie Snyman
For MuseScore version 3.4.2

This document describes problems with MuseScore's horizontal spacing algorithm and proposes a new design.

Contents

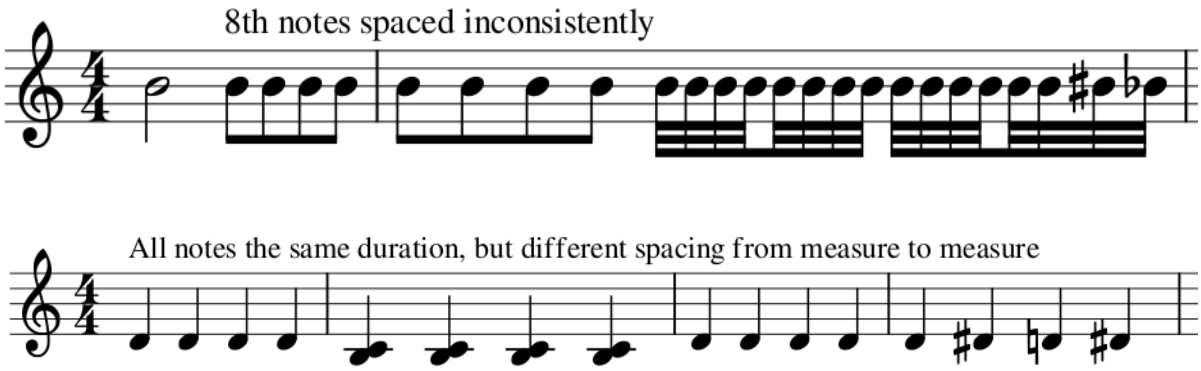
1. Two types of inconsistent spacing 2
   1.1 Unequal spacing of equal-duration notes on the same system 2
   1.2 Inconsistent spacing-ratios between notes from one measure to the next 2
2. What should the spacing look like? 3
3. Examples from actual scores 3
4. The current algorithm 5
5. Pinpointing the source of the problem 6
   5.1 computeMinWidth() doesn't take note-durations into account 6
   5.2 Pass 2 in collectSystem() distributes remaining space incorrectly to measures 7
   5.3 stretchMeasure() uses a flawed calculation 7
6. Proposed new design 9
   6.1 The mockup 9
   6.2 Default values and limits 10
   6.3 Added flexibility 10
7. Coding it 11
   7.1 All the core spacing-math in one function 11
      7.1.1 "Original Spacing Formula" selected 11
      7.1.2 "Fixed Ratio" selected 11
      7.1.3 "Spacing Lookup Table" selected 12
   7.2 A single line changed in StretchMeasure() 13
   7.3 Getting the right measure-width - 3 possible approaches 13
      7.3.1 Option 1: Calculate the right segment width in computeMinWidth() 13
      7.3.2 Option 2: Calculate the right measure-width in collectSystem() 14
      7.3.3 Option 3: Stretch a complete system in collectSystem() 14
   7.4 Equal measure widths per system 16

1. Two types of inconsistent spacing

MuseScore's spacing-issues causes 1) unequal spacing of equal-duration notes on the same system, and 2) inconsistent spacing-ratios between notes from one measure to the next.

1.1 Unequal spacing of equal-duration notes on the same system

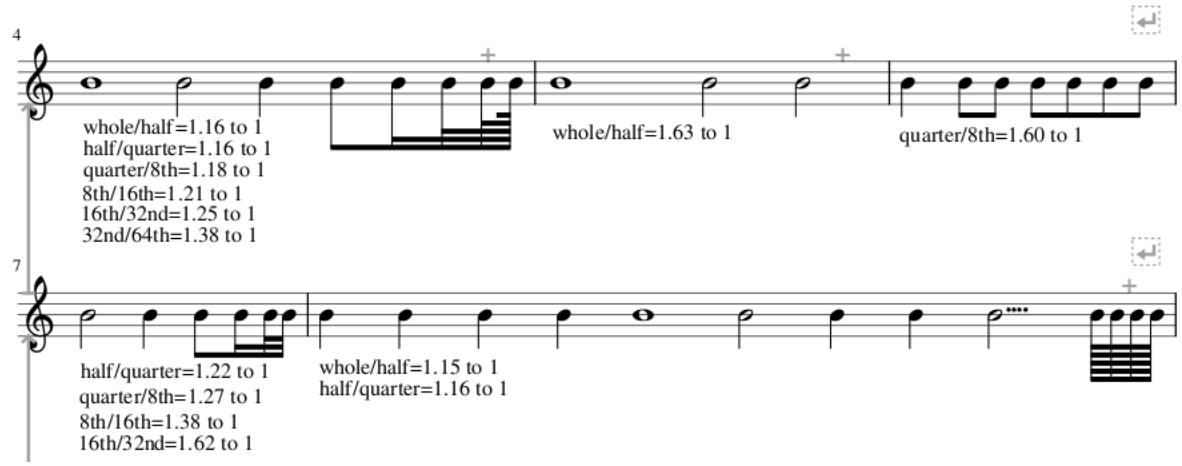
Notice how equal-duration notes get different amounts of space from one measure to the next.



This issue has been reported here, here, here and here. If we try to correct a score's spacing by adjusting each measure's stretch (be pressing "{" and "}"), we run into another more subtle form of inconsistent spacing:

1.2 Inconsistent spacing-ratios between notes from one measure to the next

In this case, the ratio of space between two given note-values within the same measure, differs from measure to measure. Careful measurements of the notehead-to-notehead widths in the following example reveals wildly inconsistent width-ratios from one measure to the next:



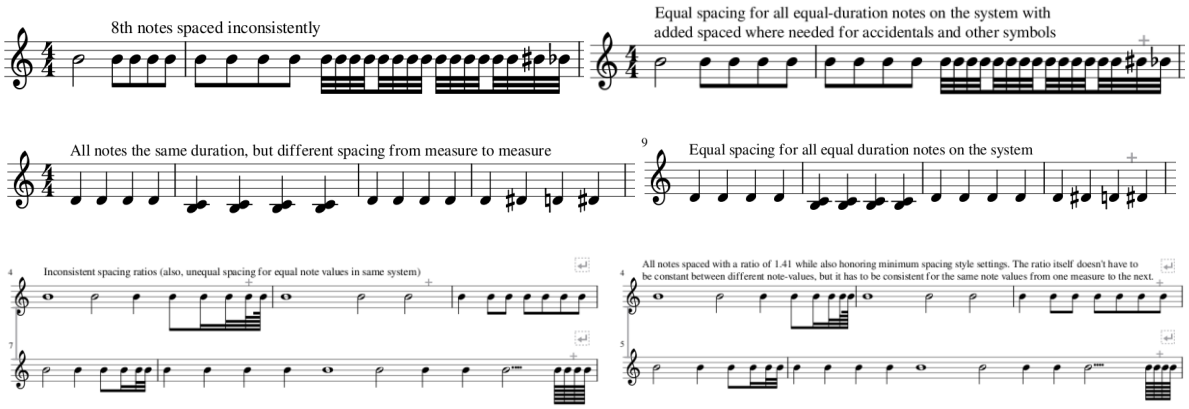
Note the dramatic difference between the whole/half ratio the first two measures of the example , as well as the different quarter/8th ratios in the last two measures. This issue has been reported here.

This means that, if you try to correct inconsistent spacing within a system, you might end up with consistently spaced quarter notes, but inconsistently spaced half notes. If you then try to get the half notes' spacing consistent, quarter notes' spacing would be inconstant again. Thus, this problem makes it impossible to achieve consistent spacing by adjusting each measure's layout stretch.

2. What should the spacing look like?

Engraving best-practice requires that notes of equal duration are spaced equally within the same system (Behind Bars, page 40). Only when symbols (like accidentals, dots, arpeggios, etc.) between one note and the next cannot fit into the given space, should more space be added between the notes.

Here are the same examples again next to their corrected versions:

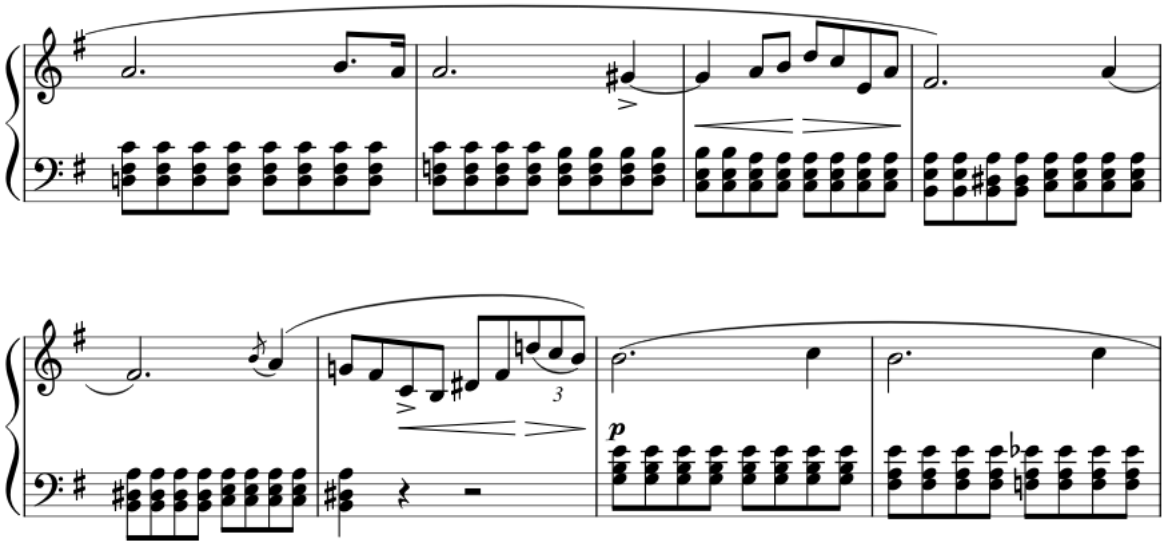


3. Examples from actual scores

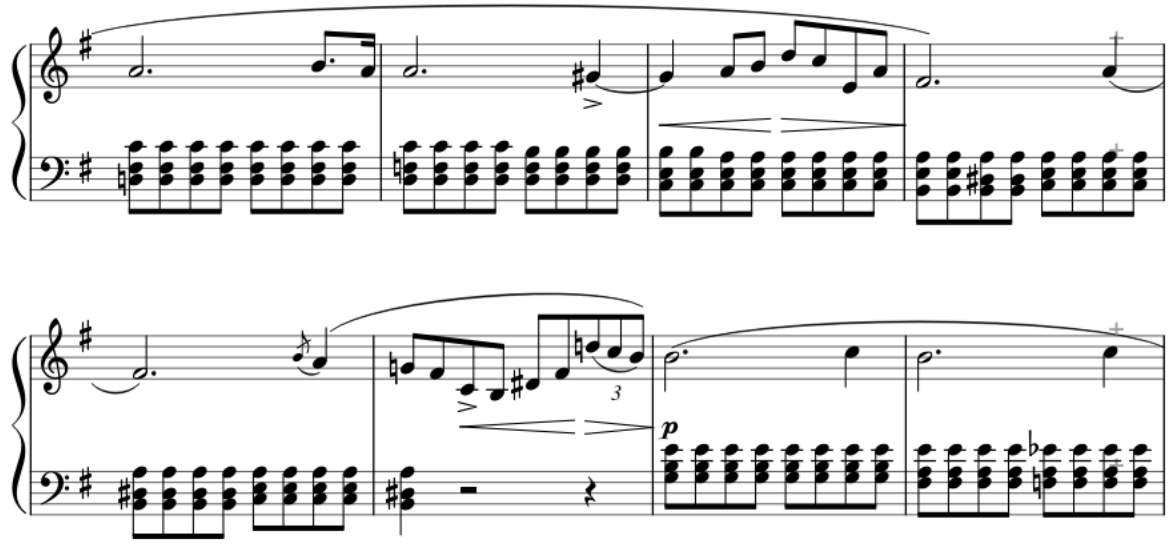
How prominently these issues manifest depends heavily on the contents of the score. It tends to be most prominent in scores with lots of same-duration notes with width-altering properties (like accidentals, ledger lines, dots, 2nd-interval chords, etc.) as well as scores with a single line of music, like orchestral parts for example.

Here are some cherry-picked examples of where it is very apparent, compared to their corrected versions.

Chopin's Prelude in E Minor… note the wildly inconsistent spacing of the left-hand chords.

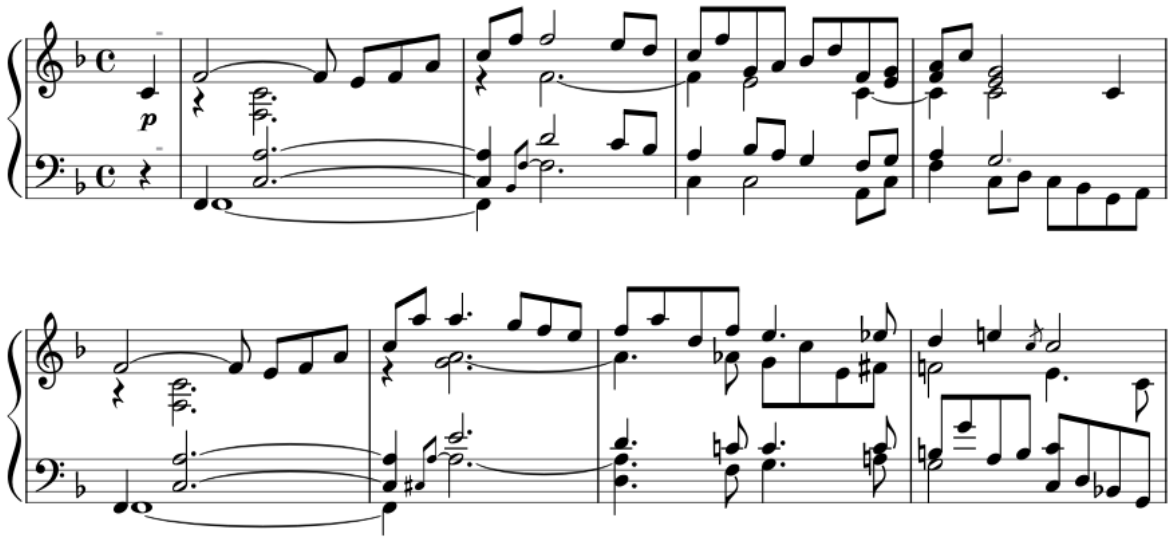


… and here it is with consistent spacing:

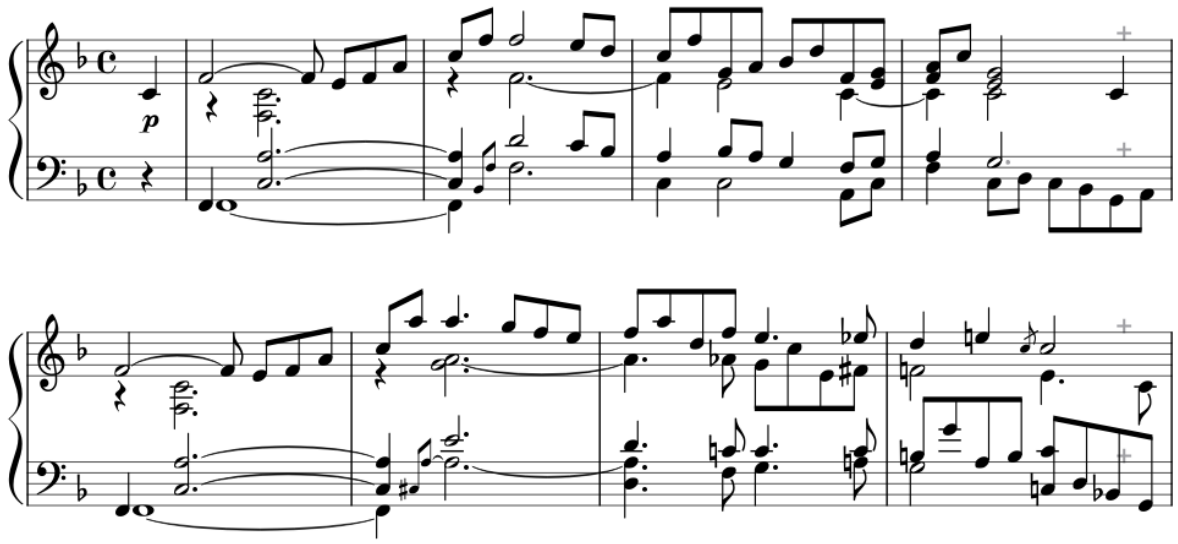


Much better, isn't it?

Schumann's Traumerei… the issue is less visible here. However, note how crammed the 8ths in measure 3-4 are compared to the rest of the system. Also note how widely-spaced measure 7 are compared to the surrounding measures in the same system. The overall result looks cluttered and jumbled, even if you don't directly notice the spacing problems.



Here it is again, but with consistent spacing and a spacing ratio of 1.41:



The difference is subtle, but notice how much more easy on the eye this second version is compared to the first.

4. The current algorithm

To understand why these spacing-issues happen, an overview of the current algorithm is needed. Many thanks to Marc Sabatella for helping me understand the code.

Disclaimer: My understanding of the code is limited. Therefore I'm likely to miss the mark with my descriptions in some way or the other. Please double-check everything I say.

When laying out music on the page, MuseScore needs to decide how many measures would fit on each system. This happens in Score::collectSystem(). Each measure's width is calculated and added to a system. Once the combined measure-widths is wider than the system's desired width (usually the distance between the left and right page-margins), the last-added measure is removed again and we are left with the measures that would make up the system. Let's call this part "pass 1".

During pass 1, the each measure's width is calculated by calling Measure::computeMinWidth(). This function figures out a few parameters and passes them to Measure::computeMinWidth(Segment* s, qreal x, bool isSystemHeader), which then loops through the given measure's segments. For each segment, it calculates the bare-minimum width (to avoid collisions with neighbouring segments) and then applies stretch to it (the "Spacing" style-setting as well as user-applied stretch). The sum of these segments' calculated widths then make up the measures' width.

Back in collectSystem(), after calling computeMinWidth() in several places in the code, the measure's width is passed as a "target-width" to Measure::stretchMeasure(). This function then loops through the measure's segments. For each segment, it calculates a stretch-value based on the segment's duration (in ticks) and the duration (in ticks) of the shortest note within the measure. The shortest note will always get a stretch-value of 1.0 and longer notes more than 1.0. These calculations are then used to arrange segments within the measure's target-width according to note-durations This is where a quarter note is given more space than an 8th-note, etc.

Back in collectSystem(), after we have decided the number of measures for a system and arranged the contents of each measure according to note-durations, the total width of the measures are less

than the desired system-width. The system thus needs to be “stretched” to the right-page margin. This remaining space is then divided up equally between the measures and the new widths of each measure is passed to stretchMeasure() in order to rearrange the measures’ contents based on the new widths.

Now we have a complete system.

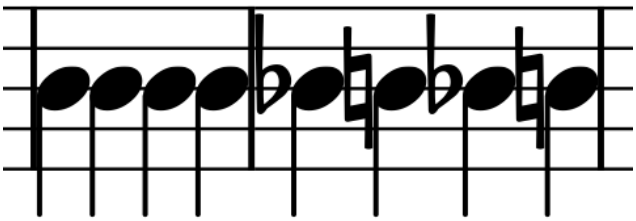
5. Pinpointing the source of the problem

So how does the two types of inconsistent spacing happen?

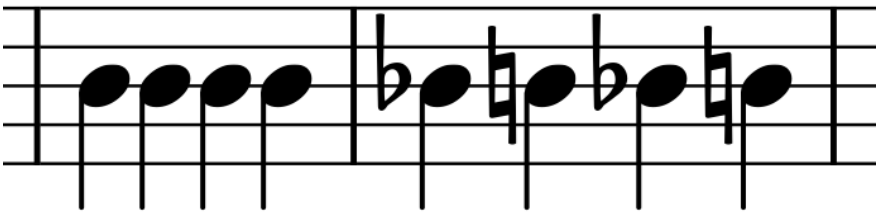
5.1 computeMinWidth() doesn’t take note-durations into account

Unequal spacing of equal-duration notes on the same system happens because, during pass 1, a measure’s width is calculated based only on its segments’ bare minimum widths (the minimum widths that would avoid collisions with neighbouring segments) and not also on its notes’ durations. To illustrate with an example…

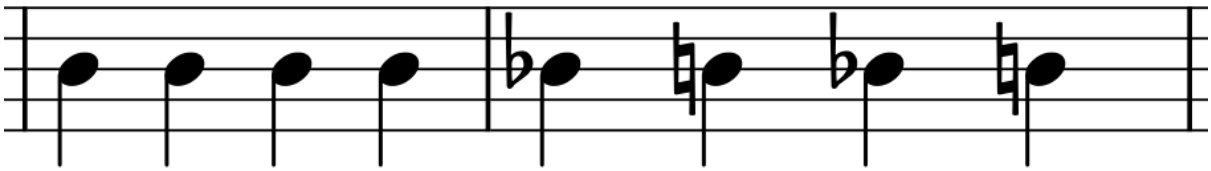
Two measures with their segments at bare minimum collision-avoiding widths would look like this:



The same two measures, now also with the minimum-widths defined in various style settings (note left margin, minimum note distance, etc.) incorporated, would look like this:



After applying the stretch defined in the Spacing style setting (in this case, a value of 2.0), we get this:



A measure’s calculated width (during pass 1) is thus a multiplication of the sum of its segments’ bare-minimum-collision-avoiding-widths (also incorporating style-setting-minimum-widths) with the Spacing style-setting and user-applied stretch.

The differences in spacing corresponds to the differences in the initial bare-minimum-collision-avoiding-widths.

computeMinWidth() should base its calculations on both collision-avoidance and note-durations.

5.2 Pass 2 in collectSystem() distributes remaining space incorrectly to measures

After the number of measures for a system has been decided, the difference between the desired system-width and the sum of the measures' widths is distributed equally between the measures. This is to make the system span its full intended width. However, distributing this remaining space equally between measures changes their widths relative to each other, again resulting in inconsistent spacing. Let's assume an example of 3 measures with consistent spacing after pass 1. A few calculations in Excel demonstrates nicely why pass 2 causes inconsistent spacing:

	these are the measure widths before pass 2	these are each measure's width's percentage of the combined width of all the measures before pass 2	now, during pass 2, equal amounts of space are added to each measure	these are the measure widths after pass 2	these are each measure's width's percentage of the combined width of all the measures
measure 1	100	16.66666667	40	140	19.44444444
measure 2	200	33.33333333	40	240	33.33333333
measure 3	300	50	40	340	47.22222222
Column totals	600	100	120	720	100

See how adding equal amounts of space during pass 2 causes the measures' widths' percentages of the total width to change. Their proportions relative to each other change, thus resulting in spacing-inconsistencies.

During pass 2, space should be distributed to measures so that they retain their proportions relative to each other.

5.3 stretchMeasure() uses a flawed calculation

Even if we try to correct measures' widths to get equal spacing for same-duration notes, spacing ratios between different note-values differ from one measure to the next, thereby preventing us from achieving consistent spacing. The reason for this is the formula with which stretchMeasure() calculates a notes duration-based spacing.

As said previously, stretchMeasure() loops through a measures segments, and calculates a stretch-value for each segment based on the its duration (in ticks) and the duration (in ticks) of the shortest note within the measure. We see the formula in this line within stretchMeasure():

```
qreal str = 1.0 + 0.865617 * log(qreal(t.ticks()) / qreal(minTick.ticks()));
```

And in a more human-readable format:

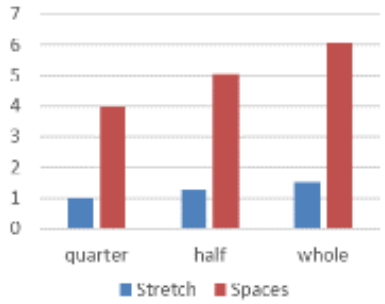
```
Stretch = 1.0 + 0.865617 * log(ticks/minTicks)
```

In the formula, “minTicks” is the measure’s shortest note’s duration and “ticks” is the current note’s duration.

The shortest note will always get a stretch-value of 1.0 and longer notes a value greater than 1.0. Every longer notes’ space is thus a multiplication of the measure’s shortest note’s space.

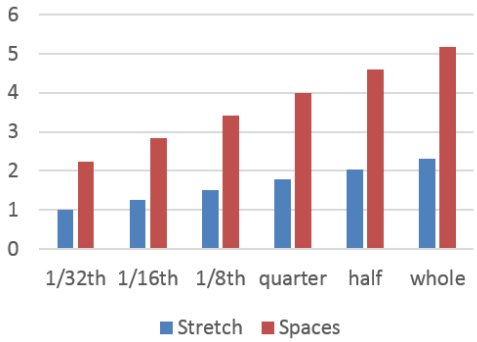
Let’s demonstrate how the problem happens. Assuming a measure has a shortest note of a quarter note spaced at 4sp. Using the formula in an Excel spreadsheet, we get the following stretches, spacings and ratios for the different note-values:

Note	Stretch	Spaces		ratio
quarter	1	4		
half	1.260576682	5.042306727	half/quarter ratio	1.260576682
whole	1.521153364	6.084613454	whole/half ratio	1.206712281



Now, assuming in the next measure a quarter is also spaced at 4sp, but this measure has a shortest note of a 32nd. Now we get the following stretches, spacings and ratios:

Note	Stretch	Spaces		ratio
1/32th	1	2.245009007	32nd/64th ratio	
1/16th	1.260576682	2.830006005	16th/32nd ratio	1.260576682
1/8th	1.521153364	3.415003003	8th/16th ratio	1.206712281
quarter	1.781730045	4	quarter/8th ratio	1.171302045
half	2.042306727	4.584996998	half/quarter ratio	1.146249249
whole	2.302883409	5.169993996	whole/half ratio	1.127589396



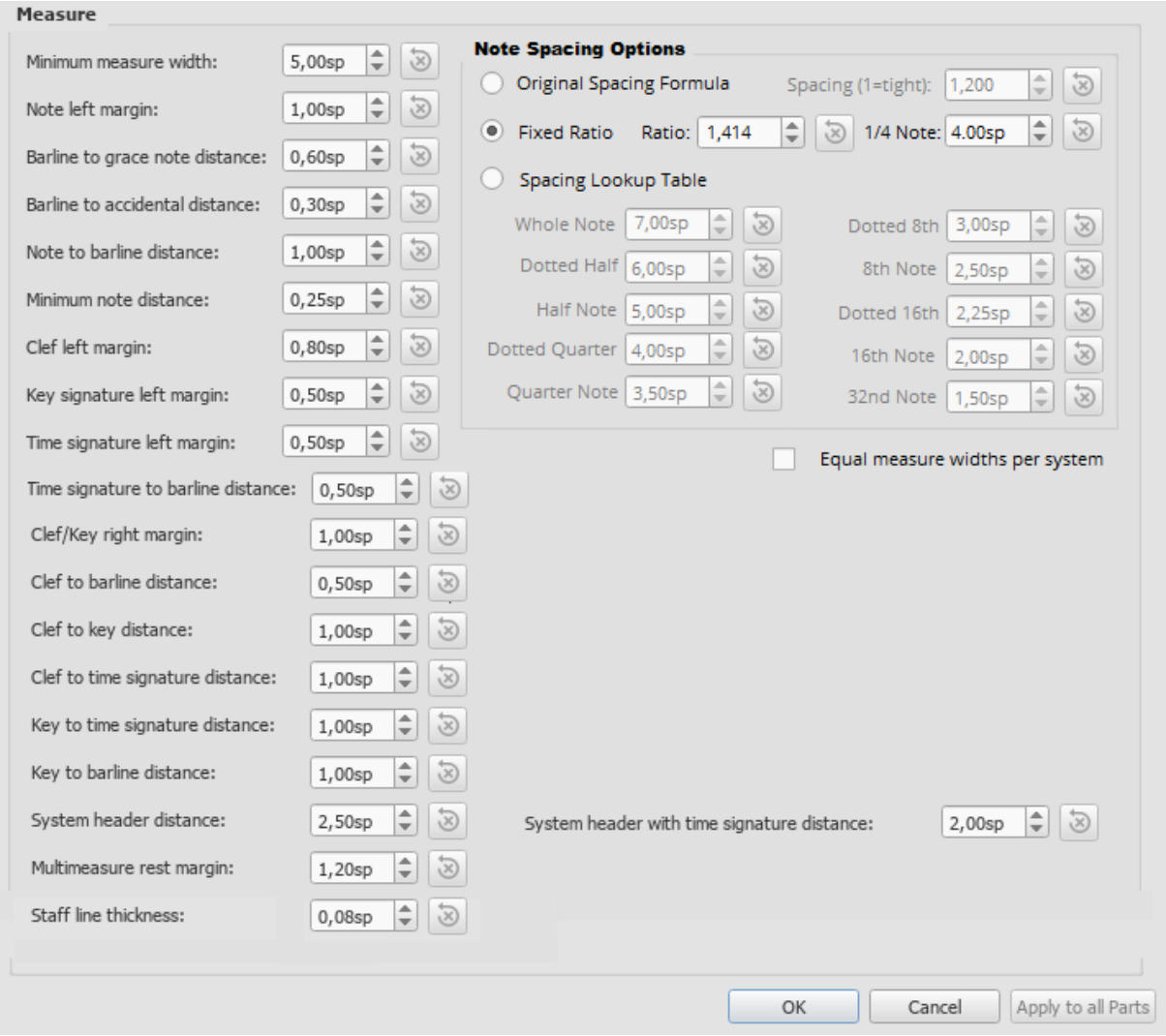
Even though both measures are spaced at quarter=4sp, the half- and whole notes have different space from one measure to the next.

A better formula to calculate duration-based stretch is needed.

6. Proposed new design

6.1 The mockup

The following new spacing style settings are proposed (the top-right quadrant in this mockup of Measure style-settings):



Selecting the *Original Spacing Formula* radio-button would make the *Spacing* setting available. The rest of the settings within the *Note Spacing Options* box will be greyed out. This would enable the current spacing algorithm.

If you select the *Fixed Ratio* radio-button, the *Ratio* setting and the *¼ Note* setting becomes available. The rest within the *Note Spacing Options* box gets greyed out. This setting will space quarter notes at least as wide as the selected sp-value. Longer/shorter notes will get correspondingly more/less space according to the set ratio. To clarify: assuming the settings in the mockup, a half note's space will be 4sp times 1.414 while an 8th note's space will be 4sp divided by 1.414. A 16th note's space will be 4sp divided by 1.414 and divided again by 1.414. More details later. (Note that Dorico uses these exact same parameters and calculation in its note spacing settings)

Selecting *Spacing Lookup Table* will enable setting individual note spacings while greying out all the other options. (I didn't include a dotted 32nd note because I couldn't figure out where to neatly place the setting in the mockup.)

If the tickbox for "Equal measure widths per system" is ticked, then measures on each system will be forced to have the same width as the widest measure-width on the system (more later). The selected spacing algorithm would still be used to position notes within measures. The user would still be able to change individual measure-widths by applying stretch.

6.2 Default values and limits

The mockup shows the suggested default values for new scores. For backwards-compatibility, scores made in previous versions of MuseScore would default to using *Original Spacing Formula* with whatever *Spacing* value is saved in the score. Upon opening the score, a dialog would give users the option to apply the new default spacing algorithm and also recommend resetting the entire score's layout stretch for best results.

The *Spacing* setting in the mockup shows the current default value and has a minimum limit of 1.0.

The shown *Ratio*- and *¼ Note*-settings values are what Dorico defaults to. I chose it like this simply because Dorico's default spacing looks good. The *Ratio*-setting would have a minimum value of 1.0. The *¼ Note* setting's minimum would be zero.

The *Spacing Lookup Table* values are taken/inferred straight from Behind Bars page 39. Each value would have a minimum of zero. With this, you could for example give shorter notes more space than longer notes if you want to for whatever reason.

The lyrics' minimum distance style setting currently defaults to zero. This default needs to be increased, otherwise lyrics' words would have no space between them in most cases with the new spacing-algorithms.

6.3 Added flexibility

Besides fixing the inconsistent-spacing problem, this design also adds flexibility. Some use-case examples:

By using *Fixed Ratio* spacing and setting the *Ratio*-value to 2.0, one will get true proportional spacing where a whole note gets double the space of a half note which in turn gets double the space of a quarter note. This feature has been requested before.

The current spacing algorithm's problematic design coincidentally spaces lyrics in quite a visually pleasing way, perhaps more so than would be the case with the new algorithms. So, in scores where the lyrics are the main focus instead of the notes (church-hymnals for example), this algorithm may be the better choice.

When writing scores by hand with pencil and paper, barlines are often drawn (with equal measure-widths) before notes are written. If you want to create a score with a handwritten look, the option to set equal measure widths in combination with the MuseJazz font (and some other style-tweaks) would be very effective.

In a more general sense, this design would allow any note-spacing ranging from beyond wide true proportional spacing to all notes crammed up against each other.

7. Coding it

Disclaimer (again): My understanding of the code is limited, therefore I'll probably miss the mark in one way or the other with the following suggested changes. Hopefully the logic in my descriptions will still be correct and useful.

Making it all happen in the code would entail the following:

1) A new function which would contain the core-math for all the spacing-options
2) Changing a single line in stretchMeasure()
3) Changes to computeMinWidth() (ideally…)
4) Changes to both pass 1 and pass 2 in collectSystem()

7.1 All the core spacing-math in one function

StretchMeasure() uses a "reference" note duration (the shortest note in the measure) to calculate a duration-based stretch-value for longer notes. To get correct spacing, I can imagine this calculation being used in more than one place in the code. So let's put it in a function of its own… computeRhythmicStretch(), for example.

Parameters passed to the function would be 1) a "reference" note-duration (let's call it "refTicks"), and 2) a note-duration for which a stretch-value would be calculated (we call this "ticks"). The function would thus return this stretch-value.

This stretch-value would then be used by stretchMeasure(), or used in other place(s) in the code to multiply the reference note's width with to get a correct width for a given note.

If *ticks* is equal to *refTicks*, the returned stretch would be 1.0. If *ticks* is smaller, the stretch would be < 1.0 and for *ticks* larger than *refTicks* the stretch would be > 1.0.

7.1.1 "Original Spacing Formula" selected

If the "Original Spacing Formula" style-setting is selected, the stretch would be calculated by the formula currently found in stretchMeasure():

$$\mathtt{Stretch = 1.0 + 0.865617 * log(ticks/refTicks)}$$

7.1.2 "Fixed Ratio" selected

If the "Fixed Ratio" style-setting is selected, the stretch would be calculated with this formula (we call the proposed ratio-style setting "ratio"):

$$\mathtt{Stretch = ratio}^{\mathtt{log_2(ticks/refTicks)}}$$

Or to put it another way to avoid confusion:

Stretch equals ***ratio*** to the power of (log base 2 (***ticks***/***refTicks***))

7.1.3 "Spacing Lookup Table" selected

If the "Spacing Lookup Table" style-setting is selected, things get more complicated and some explaining is needed. **You need to determine the note-widths of both *ticks*' and *refTicks*' and then divide *ticks'* width by *refTicks'* width to get the stretch-value.** This is essentially the calculation that needs to be done. If both *ticks* and *refTicks* correspond to an exact note-value in the spacing lookup table, then this is easy.

For example… *ticks*=480 and *refTicks*=120. This corresponds to a quarter and a 16th note respectively. Assuming the values in the mockup above, the stretch-value would be calculated as: 3.5/2.0=1.75.

But what if either/both *ticks* and/or *refTicks* don't correspond to a note-value in the lookup table (such as in the case of any double-dotted note or a note within a triplet)? For such a note, we first have to find out "between" which two notes in the lookup table they fit, and then calculate the note's space based on the two lookup-table-notes' duration and space.

Let's put this into a formula. Let's assume *ticks* doesn't correspond to note in the lookup table. We'll name the tick-value for the lookup-table-note just longer than *ticks* "longerTicks." That note's space would be named "longerSpace". Similarly, the lookup-table-note just shorter than *ticks* would be "shorterTicks" and its space will be "shorterSpace". We'll use these values to calculate *ticks'* space AKA "ticksSpace". The math would thus be (zoom in!):

```
ticksSpace=shorterSpace+((ticks-shorterTicks)*(longerSpace-shorterSpace)/(longerTicks-shorterTicks))
```

The same formula can also be used to calculate *refTicks*'s space.

Let's do a practical example. Assuming *ticks*=420 (double-dotted 8th) and *refTicks*=210 (double-dotted 16th). We need to find out both of these notes' space-values.

Let's calculate *ticks'* space first. We see that a double-dotted-8th sits "between" a dotted 8th (space=3.0 tick-value=360) and a quarter (space=3.5 tick-value=480) on the lookup table. The values for all our formula's variables would thus be:

shorterTicks=360
shorterSpace=3.0
longerTicks=480
longerSpace=3.5

Thus… ticksSpace=3.0+((420-360)*(3.5-3.0)/(480-360))=3.25

Similarly for *refTicks*… We see that a double-dotted 16th sits "between" a dotted 16th (space=2.25 tick-value=180) and an 8th (space=2.5 tick-value=240) on the lookup table. The values for all our formula's variables would thus be:

shorterTicks=180
shorterSpace=2.25
longerTicks=240
longerSpace=2.5

Thus… refTicksSpace=2.25+((210-180)*(2.5-2.25)/(240-180))= 2.375

Now that we have the space for both *ticks* and *refTicks*, we can calculate stretch: 3.25/2.375=1.368

7.2 A single line changed in StretchMeasure()

We replace this line in stretchMeasure()…

```
qreal str = 1.0 + 0.865617 * log(qreal(t.ticks()) / qreal(minTick.ticks()));
```

…with something like this (I hope I got the syntax right)…

```
qreal str = computeRhythmicStretch(qreal(t.ticks()), qreal(minTick.ticks() );
```

…where "minTick.ticks()" is the reference-ticks-value and "t.ticks()" is the note-ticks-value for which a stretch-value is being calculated. The new function "computeRhythmicStretch" will then calculate the correct stretch-value depending on which spacing algorithm is selected in style settings. The two new algorithms will ensure consistent spacing-ratios from one measure to the next.

7.3 Getting the right measure-width - 3 possible approaches

We can now arrange notes within a measures' width according to the algorithm selected in style settings, but we still need to calculate a measure's correct width in order to get consistent spacing. There are several different angles to approach this from…

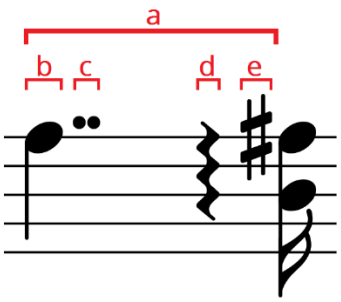
7.3.1 Option 1: Calculate the right segment width in computeMinWidth()

If the "Original Spacing Formula" style-setting is selected, then segment-widths will be calculated according to the current algorithm. If the *Equal measure widths per system* style-setting is enabled, then user-applied stretch is not incorporated into the calculation (more later).

If the "Fixed Ratio" or "Spacing Lookup Table" style-setting is selected, a note's space (distance from the notehead's starting point to the next notehead's starting point) is calculated as follows:

1) Multiply the "1/4 Note" or "Quarter Note" style setting (depending on which spacing algorithm is selected) by the stretch-value returned by computeRhythmicStretch(). The parameters passed to computeRhythmicStretch() would be the given note's tick-value and a reference-tick-value of 480 (because the style settings defines a quarter-note's space as reference width)
2) If the sum of all the symbols widths (noteheads, accidentals, etc., also including their relevant width-style-settings) between the start of the given note and the start of the next note is greater than the note's calculated space, then the note's calculated space is changed to the sum of all the said symbols' widths.
3) If the *Equal measure widths per system* style-setting is **not** enabled, then multiply the note's calculated space by the user-applied stretch-value. (The "Spacing" style setting is ignored, because it is specific only to the "Original Spacing Formula" style-setting.)

To sum this up in a picture: 1) "a" is calculated. 2) if (b+c+d+e+f) > a then a=(b+c+d+e+f). 3) "a" is multiplied by user-applied stretch if the *Equal measure widths per system* style-setting is disabled.



f = width style settings relating to b,c,d and e

Once the correct width-calculation is implemented, pass 2 within collectSystem() would also have to be modified to divide up remaining space (up to the right-margin) between measures proportional to their widths instead of equally.

This method of calculating the correct segment widths within computeMinWidth() (and thereby ensuring the correct measure-widths) seems to be the ideal solution, since it addresses the spacing issues at the very source of the problem, and doesn't seem on the face of it to have any significant drawbacks.

7.3.2 Option 2: Calculate the right measure-width in collectSystem()

Instead of modifying computeMinWidth(), one could calculate a measure's correct width within pass 1 of collectSystem() by looping through its segments and calculating the segment-widths as described above while also adding up the widths to get a correct measure-width. Pass 2 will also have to be modified as described above.

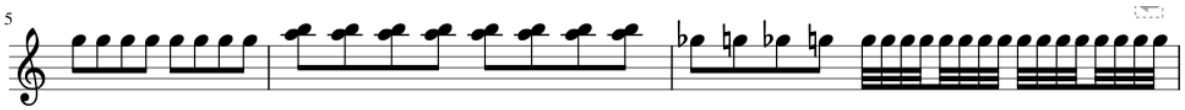
This approach, however, has some drawbacks. Firstly, we probably won't always decide on the best number of measures for a system, because that decision would still be based on incorrectly calculated measure widths. In some situations we would probably end up choosing a number of measures for a system, while an extra measure would've actually also fitted. Secondly, changes to collectSystem() doesn't seem to affect how a score is rendered in continuous view. Thus, in continuous view, we would still get inconsistent spacing.

7.3.3 Option 3: Stretch a complete system in collectSystem()

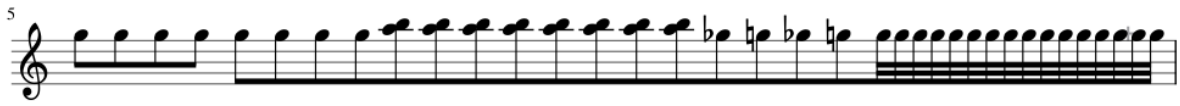
Notice that stretchMeasure() always spaces notes consistently *within* the measure. So instead of applying stretchMeasure() to one measure at a time., what if we could somehow apply stretchMeasure() to a whole system? Would it space notes consistently across the whole system? Yes it would (possibly…).

We can fake this effect by joining a whole system's measures into one (Tools > Measure > Join Selected Measures). To demonstrate:

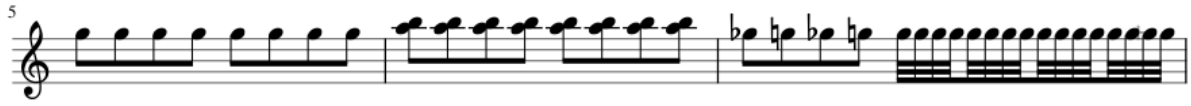
The 1/8th notes in this system is spaced quite inconsistently…



After selecting all three measures and joining them, we get this…



…and after manually adding barlines and fixing the beaming, we get consistent spacing…

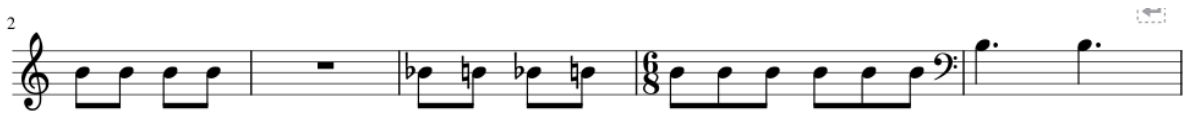


This demonstrates that, if stretchMeasure() could arrange an entire system's contents instead of just one measure, the problem of inconsistent spacing could be solved. (This is also how I created my consistently spaced examples elsewhere in this document).

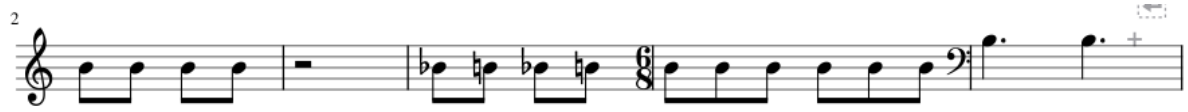
So, instead of calculating correct segment- or measure-widths, we can find a way to use stretchMeasure() to stretch a whole system to the page's margin-to-margin width right after deciding the number of measures for the system in pass 1 of collectSystem(). This would also make pass 2 of collectSystem() unnecessary, because the system would already be at its desired width. We would, however, need to restretch each measure one by one again in order to apply user-applied stretch.

Now for the drawbacks… This approach seems attractive, but it merely "covers up" inconsistent spacing-calculations instead of calculating it correctly from the start. As described above, we also probably won't always decide on the best number of measures for a system, because that decision would still be based on incorrectly calculated measure widths. And also, because changes in collectSystem() doesn't affect how scores render in continuous view, continuous view would still render with inconsistent spacing.

Stretching an entire system might also create new problems. Take this example:



If we join all the measures, manually add barlines and fix the beaming we get this…



Note the wrong positioning of the fullmeasure-rest, the time-signature, the clef-change and the change in the barline-to-accidental distance.

This approach avoids having to figure out how to calculate the correct segment/measure widths, doesn't solve *all* the spacing-problems and potentially creates a bunch of new problems.

7.4 Equal measure widths per system

Normally, when collectSystem() decides the number of measures for a system, it calculates the measure's width and adds the measure to the system. Once the combined widths of the measures exceed the desired width of the system, the last added measure is removed again from the system and we have the measures that would make up the system.

If *Equal measure widths per system* is enabled, then, when adding a new measure to the system and having had calculated its width, the widest measure's width ("widestWidth" for short) would be selected and each measure's width would be recalculated (but not yet changed) to *widestWidth* multiplied by its user-applied stretch value. These recalculated widths are then added up to see if their sum exceeds the desired system width. If it does, the last-added measure is removed again, *widestWidth* is selected again and the remaining measures' widths are calculated again. Only this time, the measure's widths would actually be changed to the calculated widths.

To clarify with an example:

The system already contains these two measures:

Measure 1 is 20sp wide with a stretch value ("stretch1" for short) of 1.0.
Measure 2 is 15sp wide with a stretch value ("stretch2" for short) of 1.3

Now we have just added Measure 3 (width=25sp and "stretch3"=1.1) to the system and need to decide if it would fit. Measure 3 is the widest, so we set *widestWidth*=25 accordingly. The combined measure-widths would thus be:

```
widestWidth * (stretch1 + stretch2 + stretch3)
```

…or in this case, 25 * (1.0 + 1.3 + 1.1) = 85sp.

Let's assume 85sp is too wide for the system and we have removed Measure 3 again, then we set *widestWidth*=20 and actually change the remaining measure-widths as follows…

Measure 1 width = *widestWidth* * *stretch1* = 20*1.0=20sp
Measure 2 width = *widestWidth* * *stretch2* = 20*1.3=26sp

Now we have a system with equal measure-widths with user-stretch applied and we can proceed to pass 2.

By always using the widest measure-width, we are guaranteed to never run into a situation where the measure isn't wide enough for its contents.