An improved horizontal spacing algorithm for Musescore (v 1.0)

Musicians rely heavily on good spacing to read rhythm. Poor horizontal spacing is often the main problem of badly presented music and hinders the reading of it.

E. Gould, *Behind bars.*

Introduction

One of the golden rules of music engraving, which is dictated by Gould's "bible" but also by common sense, is that notes of the same duration should be equally spaced within the same system. Musescore's current spacing algorithm does not achieve that. Most of the time it *approximates* it with varying degrees of success, but it is easy to build real-world examples where the algorithm fails completely.

I am presenting a new algorithm that guarantees equal spacing of same-value notes *exactly* if no accidentals are present, and *almost exactly* in case of accidentals. Additionally, the spacing rule can be customized by the change of a single formula. I offer three options: one is Musescore's current formula, one is a linear spacing option, and the default one is a formula that almost perfectly replicates the spacing proposed by Gould (see last section for the details). I would propose that these three options should be selectable by Style setting – for now they can be commented in and out of the code. As a biproduct, the } and { stretch commands now give a much more refined and natural response, especially in the narrower range.

I have implemented the new algorithm by making very surgical changes to the current code (I think less than 50 lines in total) and thus I believe that these changes should not interfere with any other behaviour of the program. You can find my code here mike-spa/MuseScore at spacing_improvements (github.com) and I would be happy to receive comments and suggestions before making a pull request, especially testing the algorithm with real-world examples and see if it breaks under any case.

Why now?

With the release of Musescore 3.6, featuring the new Leland and Edwin fonts and a new (absolutely masterful) vertical spacing algorithm, Musescore's engraving quality is getting very close to industry standard programs. However, *horizontal* spacing is still a weak point, and it has been pointed out several times in the forum. I believe this new algorithm could bring Musescore up to the game. Most people would consider these changes to be quite subtle, but when summed on an entire score they will *massively* improve the aesthetics and readability of it.

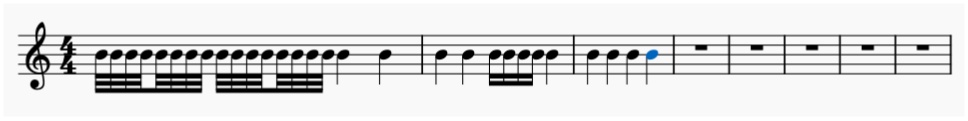
Additionally, the release of Musescore 4.0 has now been delayed, which creates the perfect opportunity to incorporate engraving improvements such as this. Obviously, this would alter the layout of older documents, so a backwards compatibility option should be provided in the starting pop up menu.

Thanks

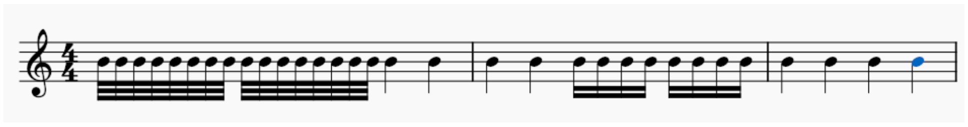
I would like to thank Marc Sabatella and Barnie Snyman for this discussion in the forum Inconsistent spacing of same-duration notes | MuseScore , without which I could not have done this.

The new algorithm at a glance

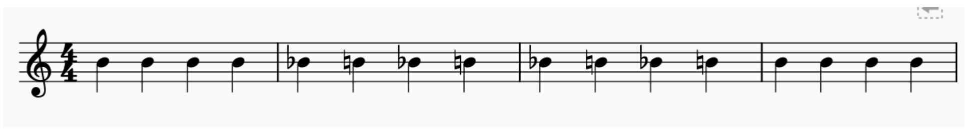
1) Before (extremely inconsistent spacing of same-value notes)

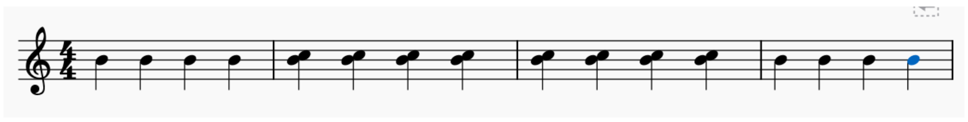


After (mathematically guaranteed consistency)

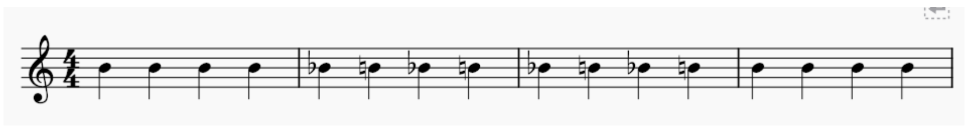


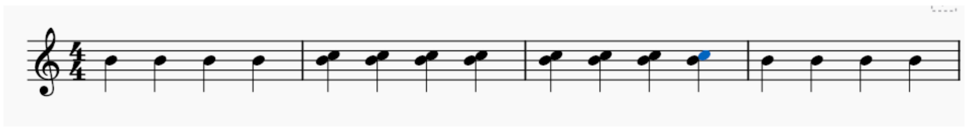
2) Before (measures with many accidentals or displaced notes are unnecessarily larger)



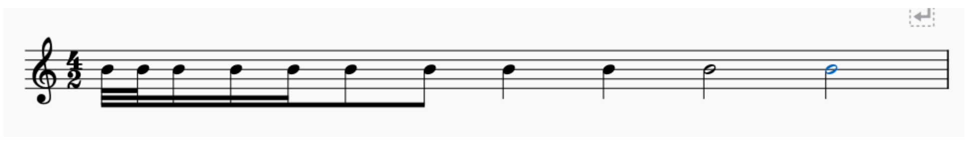


After (measures with many accidentals are equal to a good approximation)

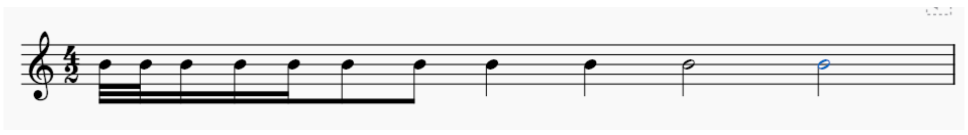




3) Before (there is a very large difference between 32th and 16th note spacing, but barely any difference between quarter and half note)



After (following Gould's spacing rules, with smoother, more natural transitions):



Musescore 3.6 algorithm

If one wants to cut it down to the essence, the reason for the inconsistent spacing in Musescore 3.6 is that the algorithm lays out the music *measure by measure.* This is understandable: it makes the algorithm conceptually simpler, the code simpler, and the process computationally lighter. However, there is absolutely *no way* to obtain consistent spacing across a system if every measure is processed separately. That's, in essence, what we need to change.

Here is a step-by-step summary of how the algorithm works. It all happens within `LayouSystem::collectSystem()`.

1. Store the current width of the system in the `minWidth` variable (zero at the beginning).
2. Append a new measure.
3. Compute the width of the new measure by calling `Measure::computeMinWidth()`. The width is computed by:
   3a. Compute the minimum non-collision distances between the elements.
   3b. Multiply by the `basicStretch()` parameter, which is the product of the Spacing setting in Style -> Measure (defaults to 1.2) and the user stretch (defaults to 1 and does + or - 0.1 for every } or { command).
4. Verify if `minWidth` plus the width of the new measure exceeds the maximum available width to the system. If no, go back to point 1 to append another measure. If yes, remove the last measure, break the loop, and go to the following point.
5. Compute the remaining white space at the end of the system, which is stored in the variable `rest`.
6. Distribute the rest to each measure according to its `layoutWeight()`, which is proportional to the measure's duration. In practice this means that, unless there is a time signature change, the rest is distributed equally to each measure and summed to their current width. Let's call this the `finalWidth` of each measure.
7. Stretch each measure to its new size by calling `Measure::stretchMeasure(finalWidth)`. The function sets the measure's width to `finalWidth` and, *within the constrain of such width*, applies value-based spacing (i.e., ensures that longer notes get larger space) by a logarithmic formula (see later for details).

The "errors" of the current algorithm

1. `stretchMeasure()` applies value-based spacing with respect to the shortest note *of the measure.*

To put it simply: if the shortest note (say, a 16th note) has duration `t` and occupies a space `d`, then a longer note (say, a quarter note) which has longer duration `T` should also occupy a larger space `D`. The formula you use to calculate `D`, i.e. the function `D = func(t, T, d)` is what we call *spacing function* or *spacing formula.* The current formula is a logarithmic function, which is a perfectly legitimate choice, but causes some non-ideal behaviour (see last section for details). Anyhow: *regardless* of which formula you choose, if in one measure the shortest note is, say, a 16th, and in the next measure the shortest note is, say, an 8th, the space assigned a quarter note in the first measure will be different to the space assigned to a quarter note in the second.

2. `StretchMeasure()` applies value-based spacing *within the constrain* of a given measure width.

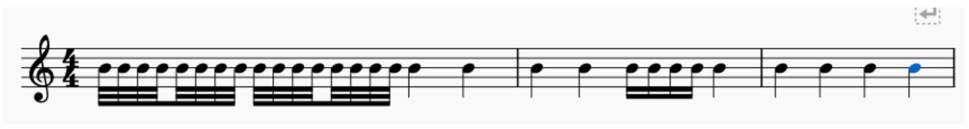
And the width of such measure was calculated by `computeMinWidth()` without considering note duration. This means that, many times, there is simply not enough space in the measure to obtain the desired width for a note. This actually distorts the proportions assigned by the spacing function. And it distorts differently on each measure.

3. The white space at the end of a system is distributed according to `layoutWeight()` which is proportional to duration, and therefore is almost always *the same* for each measure.

This is an error because, if you add the same width to each measure, you are proportionally stretching shorter measures more than longer measures, so you are again distorting the relative proportions. In principle, to maintain proportions, you should distribute the white space proportionally to the current width of the measure.

Now the fun fact: this error causes a distortion that distorts in the opposite direction to error 1, so it is actually beneficial. Without this "error", Musescore's layout would be completely broken.

The problem is that the amount of correction depends on how much remaining space there is at the end of the system, which is completely unpredictable. One extreme case is the one I've shown earlier in example 1: by having many empty measures to fill the remaining space of the system, there is basically no white space to distribute, which results in horrible spacing. Granted, you would never have so many empty measures in reality, but it could also easily happen with the music. In fact if, in that example, I place a line break after the third measure, suddenly the result is much better, though still far from perfect.



By the way, this is also the reason why the trick proposed by Marc Sabatella works, i.e. setting the layout to the minimum possible width and then fixing it with line breaks. It works because after the line break there is always plenty of white space to distribute.

In fairness to the original developers: even though I call this an error, it was probably a deliberate choice, as it produces in fact a better layout than what you would get with a "correct" distribution of the white space.

My corrections

1. `StretchMeasure()` now applies value-based spacing with respect to the shortest note *in the system*.

This is by far the simplest (almost trivial, I would say) but most consequential idea. It is executed through a very simple, dedicated method called `Measure::minSysTicks()` which, starting from a measure, goes up to its parent system, and returns the shortest note of such system (measured in ticks, which is Musescore's internal unit). This way every measure has a way of knowing the shortest note of the system it belongs to, and every measure can comput its own stretch according to that.

Remember, however, that `StretchMeasure()` works within the width boundary calculated by `computeMinWidth()`, which does not consider note values. In order to get consistent spacing, the width of a measure should be natively calculated *already taking into account* the duration of its notes with respect to the shortest note of the system. This leads me to

2. `ComputeMinWidth()` now, after calculating the minimum non-collision distance, applies a further multiplication factor that depends on the duration of the current note.

Since it does so by using the exact same formula as `stretchMeasure()` (by the way, I've moved the formula to a dedicated method called `Measure::stretchFormula()`), consistent spacing across the system is now *mathematically* guaranteed, as long as:

3. The remaining white space is distributed according to the current measure width.

This ensures that the proportions are not altered. The weight is now calculated by a dedicated method called `stretchWeight()`, which is called instead of `layoutWeight()`.

Small catch: naively, one would think to obtain the measure's current width by using the standard `*measure->width()`. This doesn't work, because `width()` also includes clefs, time signatures, key signatures, ecc, which should *not* be stretched. So, you need to make sure to compute only the width occupied by the notes.

Small catch to the small catch: notes also contain accidentals. Therefore, measures with many accidentals will receive a larger weight. This was the main responsible for causing the problem I've shown earlier in example 2. Therefore, you want to make sure that you compute the "basic" width of each note, that is the width the note would have *without the accidental*. This makes sure that measures with accidentals receive the same weight as measures without accidentals. And the problem should be fixed.

Small catch to the small catch to the small catch: the problem is actually *not* fixed. Much improved, sure, but not fixed. Measures with accidentals are usually larger than measures without accidentals from the onset. So, if you want to obtain equal spacing *after stretching*, measures with accidentals should receive a *smaller* stretch than measures with accidentals, not an *equal* one. With an equal one, measures with accidentals still end up being larger. I obsessed about this for an afternoon, and I eventually realised that it is impossible to mathematically guarantee equal spacing after stretching (in case of accidentals), because you get into a recursive equation. So, I ended up applying an empirical factor that distributes a *slightly smaller* weight to measures with accidentals. It works quite well, but still not *exactly*. The extent to which this bothers me is immeasurable, but I'm pretty sure there is no solution within this algorithm. If you happen to find one, please do let me know. Anyhow, I am quite sure that only me and some other guy with OCD will ever notice this.

Back to ourselves: the last thing that needs fixing is in the main function, that is `collectSystem()`. Indeed, remember that measures are appended to the system one at a time. It may well happen that, say, the shortest note in the first two measures is an eighth, but the third measure contains a sixteenth. Now we will have that the spacing of the first two measures was computed with respect to the eight note, but the spacing of the third measure and all the subsequent ones will be computed with respect to the sixteenth note. In practice, the first two measures thus will be narrower. This leads me to:

4. Every time a measure is appended to the system, check its shortest note. If it's shorter than the current shortest one, *all* previous measures need to be re-laid out by running computeMinWidth() again. If the system ends up being too long, and therefore the last measure is removed, the change to the previous ones should be undone by running computeMinWidth() once more.

The new algorithm: MS21

I will admit that I initially nicknamed my algorithm msGold, as a reference to "Gould" and to the "golden standards" of engraving… but it felt way too cocky. So, I decided for a more neutral MS21 (MS being not only MuseScore's but also my initials, funnily enough).

Jokes aside, here is the step-by-step summary.

1. Store the current width of the system in the `minWidth` variable (zero at the beginning).
2. **Store the current shortest note of the system in the variable `minSysTicks` (set to a random high value at the beginning).**
3. Append a new measure.
4. **If the shortest note of the new measure is shorter than the previous shortest one, re-lay out the previous measure and update `minWidth` accordingly.**
5. Compute the width of the new measure by calling `Measure::computeMinWidth()`. The width is computed by:
   5a. Compute the minimum non-collision distances between the elements.
   5b. Multiply by the `basicStretch()` parameter, which is the product of the Spacing setting in Style -> Measure (defaults to 1.2) and the user stretch (defaults to 1 and does + or - 0.1 for every } or { command).
   5c. Multiply by the stretchFormula(), where the shortest note corresponds is the shortest of the system.
8. Verify if `minWidth` plus the width of the new measure exceeds the maximum available width to the system. If no, go back to point 1 to append another measure. If yes, remove the last measure, break the loop, and go to the following point.
9. **If the last measured caused a re-layout, undo this re-layout after removing the last measure by calling `Measure::computeMinWidth()` again.**
10. Compute the remaining white space at the end of the system, which is stored in the variable `rest`.
11. Distribute the rest to each measure according **to the new function `stretchWeight()`**, which is proportional to the measure's current **width**. Add the rest to each measure obtaining its `finalWidth`.
12. Stretch each measure to its new size by calling `Measure::stretchMeasure(finalWidth)`.

I have tested the algorithm with more or less everything I could think of, and it always seems to hold. Please *try to break it*. See if you can find weird, limit cases that escape the rules. Also, try to apply it to real music and see if it improves the layout and it behaves as it should.

Computational cost

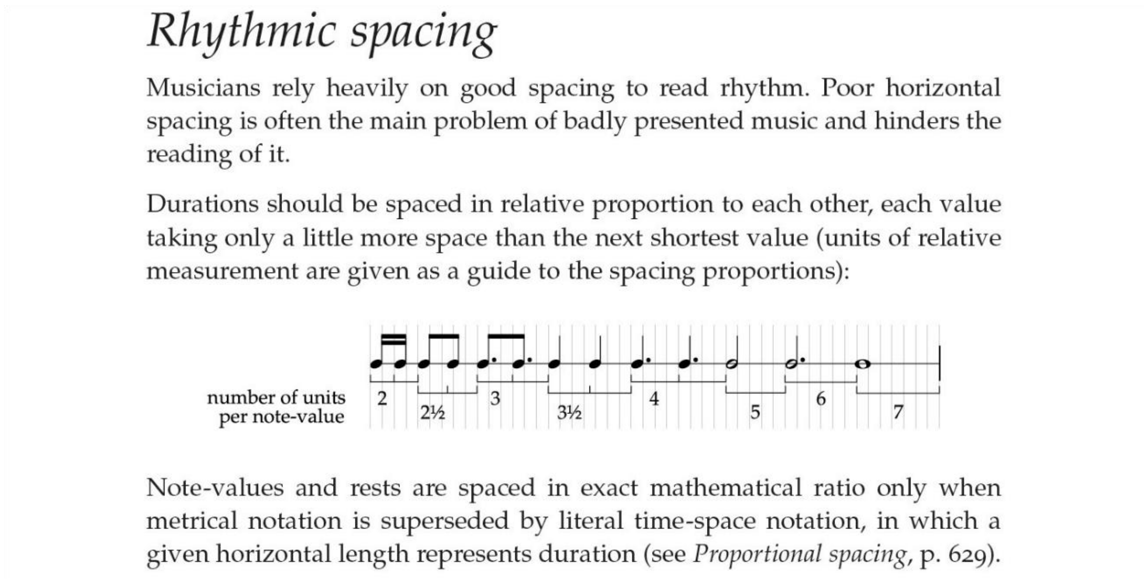
The new is algorithm *is undoubtedly and unavoidably more expensive* because it needs to loop several times across the measures of a system. I have no experience on how to quantify this cost. My gut feeling is that this should still be negligible compared to other processes such as playback or graphics, but I ultimately don't know.

An immediate improvement would be to avoid calling the method `minSysTicks()` from within `computeMinWidth()`, because `computeMinWidth()` is applied to every measure, but the shortest note *of the system* is obviously the same for every measure of a system, so it is unnecessary to compute it every time. The obvious solution would be to *pass* the shortest note of the system to `computeMinWidth()` when calling it. The reason I didn't do it is because `computeMinWidth()`

is called by *many* other methods, so changing its definition was a bit too scary. But this is definitely in the TODO list.

The fascinating math behind music spacing (made approachable… I hope)

Let us start by considering this small excerpt from Gould's *Behind bars*, which is pretty much considered as one of the bibles of music engraving.



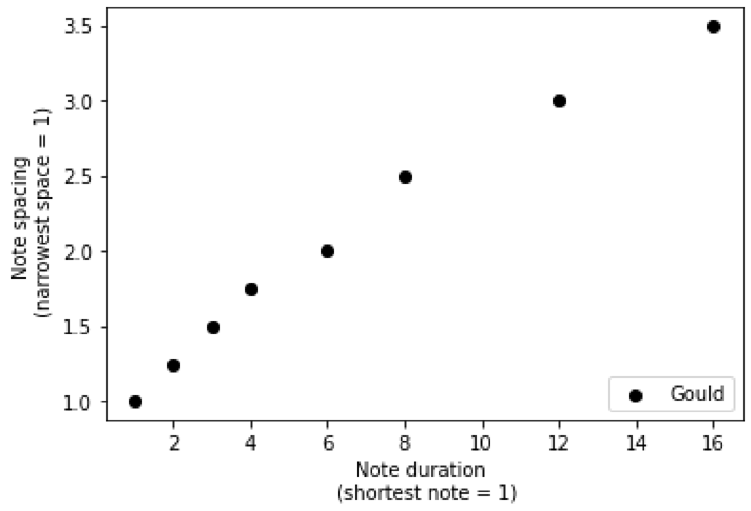
Rhythmic spacing

Musicians rely heavily on good spacing to read rhythm. Poor horizontal spacing is often the main problem of badly presented music and hinders the reading of it.

Durations should be spaced in relative proportion to each other, each value taking only a little more space than the next shortest value (units of relative measurement are given as a guide to the spacing proportions):

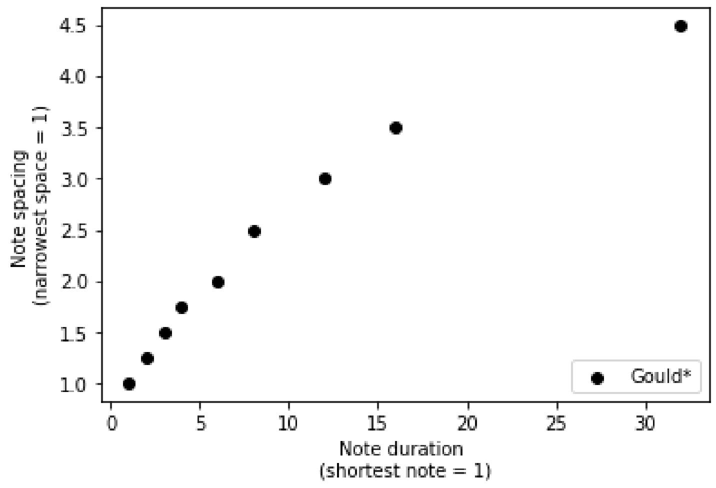
Note-values and rests are spaced in exact mathematical ratio only when metrical notation is superseded by literal time-space notation, in which a given horizontal length represents duration (see *Proportional spacing*, p. 629).

This spacing table is to me very fascinating, as it's clearly the result of decades of development in handmade engraving, and it is basically a compound of rules that surely have developed out of painstaking trial and error. To get a clear view, let us plot them on a graph that has the note duration on the x-axis (1 representing the shortest note), and the corresponding note space on the y-axis (1 representing the narrowest space). We get this.



Now, while a human engraver works best with a lookup table such as the one in the Gould, for a computer engraver it is not very elegant nor efficient to work with a lookup table, especially because in order the get the values in-between (imagine notes in complex tuplets) you would anyway need to interpolate in-between the points. At that point, it is thus much better to find a general function that interpolates *all* of the points as best as possible.

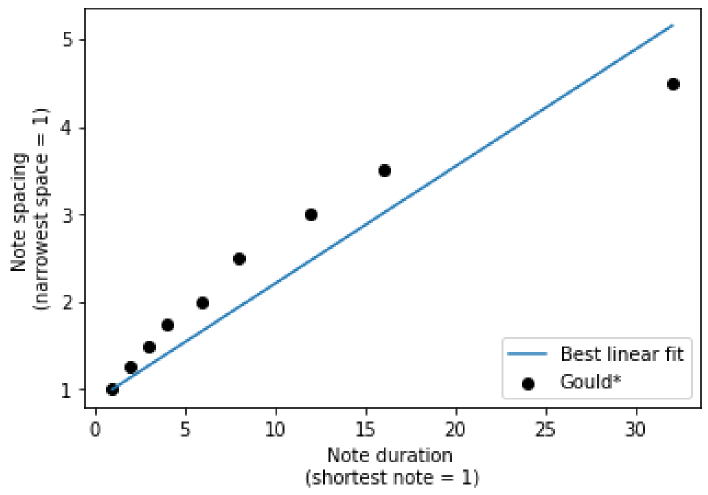
By an immediate look, we can already tell that the function is nonlinear, meaning (in the simplest possible terms) that it cannot be interpolated by a line. This would be even more evident if Gould added the next note value, and it is a real pity that she didn't. However, it's easy to bet that the next note (I.e., the breve) would have a space of 8 or 9 units (I would say 9, because they increased three times by ½ and three times by 1, so it makes sense that the next step increases by 2). This is what we get (I'll refer to it as Gould*).



The fact that the function is nonlinear is now very evident. Now, we certainly don't need to take Gould's rule as if it's set in stone, and in fact linear spacing has many advocates. If we try to find a linear function that "kind of" fits to Gould's points (and I won't enter in the details of how mathematically you say it "fits") we get the following:

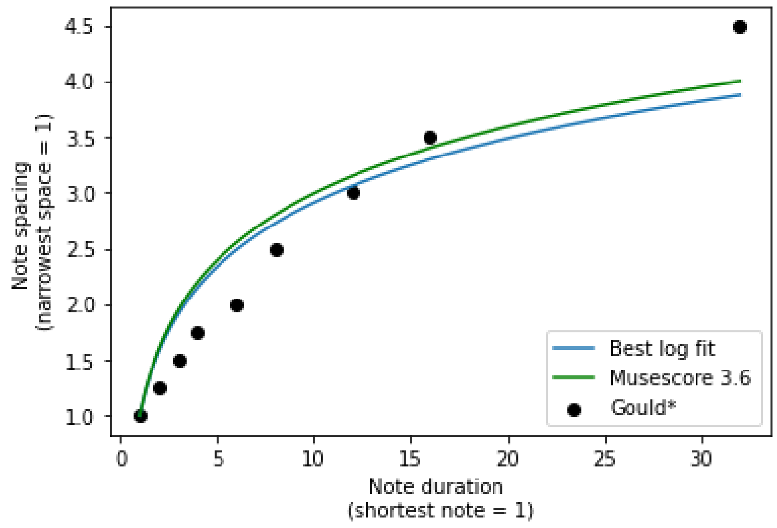
$$s = 1 - 0.134 + 0.134 * t$$

where *t* is the note duration (again, the shortest note is assumed to be t = 1) and *s* is the note spacing. This is what it looks like.



You can see by yourself the trade-off of linear spacing: if you try have larger spacing in the lower range, the longest note would end up having a ridiculously large space. Conversely, if you want to keep the space of the long note within reason, you need to accept that the shorter notes will have a smaller space.

Now I finally come to what Musescore currently does, namely a logarithmic function. The benefit of a log function is precisely the opposite of the linear one: it allows you to get a wider spacing of the shorter notes without making the longer notes explode. Now, this is what you get if you try to "kind of" fit a log function to Gould's points, and you can see that it is very similar to the current Musescore spacing formula.

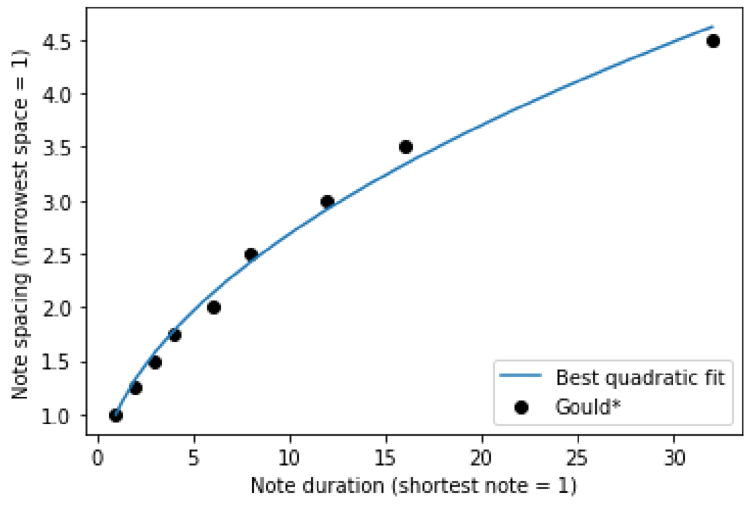


Musescore's formula is:

$$s = 1 + 0.865617 * \log(t)$$

As I mentioned earlier, this is a perfectly legitimate choice, and it was probably chosen because keeping a smaller space for the longest notes was causing less trouble and spacing inconsistency with the previous algorithm. However, it causes the problem I described at the very beginning in example 3, namely the fact that there is a large difference between the shortest notes, but almost no difference between the longest ones. The reason you can see from the plot above: the log function is very steep for short values, but it basically flattens for longer one.

We are still left with a question: isn't it possible to find a function that fits better to these points? Yes, and it's actually very simple. In fact, just by looking at the points one could tell (and I find this very fascinating) they were unconsciously shooting for a quadratic relation (or rather its inverse, the square root). Indeed, a square root function almost perfectly fits the points.

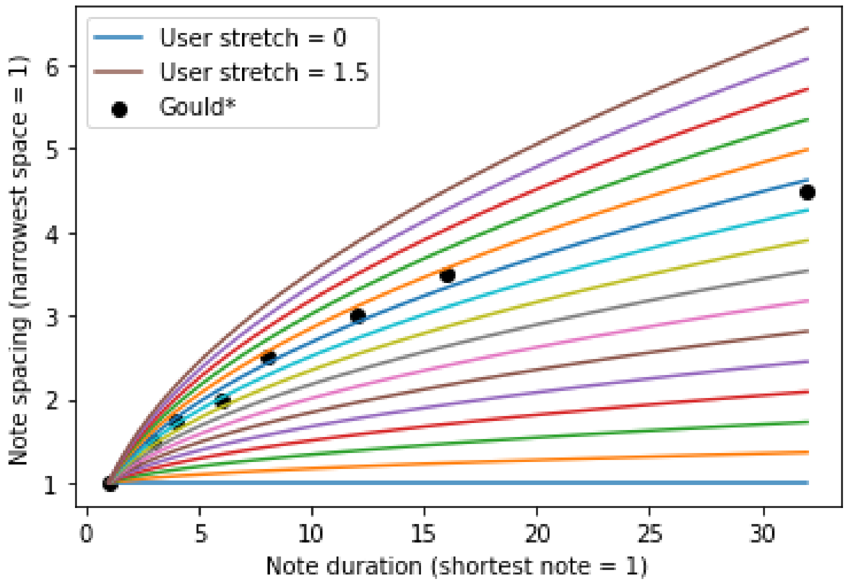


And it reads as:

$$s = 1 - 0.777 * 0.777 * \sqrt{t}$$

Finding out the square-root relation was interesting also for another reason. Dorico mentions that its default spacing formula applies a multiplication factor of 1.41 between the spacing of one note (for instance an eighth) and the spacing of the next longer one (for instance a quarter). Do you wonder why that is? Quite simply, because if the eighth has duration *t*, the quarter has duration *2t*, and the square root of 2 happens to be 1.41. In other words: yes, I believe Dorico's default is a square root formula.

A final remark: I have decided to incorporate into the coefficients of each of these formulas also the spacing stretch defined by the user. That is both the general setting in Style->Measure, which defaults to 1.2, and the user stretch commands, which default to 1 and change by +- 0.1 for each } or { command. Here is what it does to the spacing (exemplified with the square root formula but also valid for the others).



When the user stretch is equal to 1, you get the default spacing, which matches the Gould. By pressing } or { you increase/decrease the spacing coefficient, causing the notes to be spaced narrower or larger. At the lowest point (user stretch = 0) you get flat curves, meaning that no spacing is applies, meaning that notes are kept at the minimum non-collision distance. Feel free to experiment with this, and you should find that this gives a much finer response to the stretch commands.

Michele Spagnolo
Vienna, November 30th, 2021