Beam Subdivisions in/for LilyPond

Notes for a Reimplementation

Urs Liska

2018-01-02

In January 2016 I worked on the code which is responsible for beaming patterns in GNU LilyPond. This is the code that decides how many beams any given stem gets within a beamed group - the decision *which* notes to beam has already been taken when the beaming *pattern* is calculated.

Before that I had repeatedly complained about errors in beam *subdivisons*. Several fixes had been made upon my request, and it turned out that there were always cases that produced yet another wrong result, triggering new bug reports and new patches. So I intended to go into depth and fix it properly. Unfortunately I couldn't finish that back then because it was somewhat over my head.

Re-reading code both from the `master` branch and my working branch (the relevant code is mainly in `lily/beaming-pattern.cc`) made me realize that - while intending to dig deeper - I essentially made the same mistake that is present throughout the released code: I tried to fix one issue and then looked for further problems. Essentially the beaming pattern code places patches on top of patches, while the basic outline of the algorithm gets very much blurred.

The conclusion is to start from scratch by assessing the topic from a *conceptual* perspective. It is essential to know how beaming patterns behave musically and mathematically, and only then it makes sense to design a strategy/algorithm and think about the implementation. It turns out that often fixing/patching corner and special cases can be avoided by changing the reference point in the first place. This conceptual assessment is the purpose of this document.

The most fundamental aspect is rethinking in what order information will be processed. My impression is that the existing code sometimes calculates values (e.g. flag directions) and later has to touch that information again.

Maybe this convolution of the code is related to two processing stages not being sufficiently isolated. At first *nominal* beam counts have to be calculated for each stem by identifying subdivision points and determine their metric relevance. Only then the actual number of beamlets printed to the left and right side of the stem should be considered, taking all further aspects (count of neighboring stems, single flags, trailing rests etc.) into account.

The old work can be found in the `dev/urs/beaming-pattern` branch, and specifically up to commit `6610d2faa4782d3414686b63ba622c354bbce9bb` (Jan 26, 2016), later commits are merely merge commits to keep the branch in sync with later development. However, I suggest *not* to continue working on that branch but to restart from scratch on a new branch. With regard to the actual *implementation* I suggest not to modify the existing code but to start that from scratch too while trying to reuse existing code where appropriate.

General Observations

I can see three musical problems with beaming pattern in the current implementation:

- Beam subdivisions should be *on* by default.
  Actually this isn't an error, but a feature request, and maybe a controversial one (given the implications on existing scores). But note that Elaine Gould confirmed that beams should be subdivided by default ("YES absolutely, (c) as default. Look how easy the rhythms are to read in comparison with the other examples!").
- Shortened beams cause subdivisions in the last beat of a beam to have too many beams. This is wrong but actually done on purpose. Dorico currently does the same, and, by the way, the text in Gould (p. 156/157) is ambiguous because it lacks an example for this case. But she confirmed that beam shortening should not affect the beam count at subdivisions.
- The handling of tuplets is totally messed up.

I am not sure to what extent the handling of the finer details (treatment of different note values (flags), single beams at a subdivision, shortened beams and rests) are currently treated correctly, but I think we should think this from the ground up and see *then* how these details work and have to be implemented.

Analysis

Determine Subdivisions

In a first step we should determine the stems that represent subdivisions and the nominal beam count for that subdivision. Both values should be stored as properties of the stem to the right of the subdivision, as that is the one whose metric situation is responsible for subdivisions.

Special concern is necessary when encountering tuplets, but this will be discussed in a later section.

General Organization of a Beam

A beam is part of a measure, and it is especially part of the `beatStructure`. *(NOTE: I'm not clear what happens when a beam crosses a barline).* The beat structure is based on `baseMoment` and consists of a sequence of beats/groups. Usually it is defined in the time signature but it can manually be overridden.[1] For example in 6/8 time `baseMoment` is 1/8, and the beat structure `#'(3 3)`, i.e. two groups/beats of three quavers each.

baseMoment vs. subdivisionInterval

Currently the `baseMoment` property is used to determine beam subdivisions, but this is not sufficient as there are cases where the subdivision interval must be different from `baseMoment`. This is most obvious in compound meters with differing denominators. For example in the compound time signature 2/4+5/32 `baseMoment` is 1/32 and `beatStructure` can be `#'(8 8 5)` (or also `#'(8 8 2`

[1]For further details see "Setting automatic beam behaviour" in LilyPond's notation reference.

3)). Of course it is useful to subdivide the 2/4 part with 1/8 or 1/16, but this is currently not possible: figure 1 shows a reasonable rendering of that measure, subdivided by 1/16. (Probably it would be better to use 1/8 as interval in real scores, but this way is better for demonstration.)

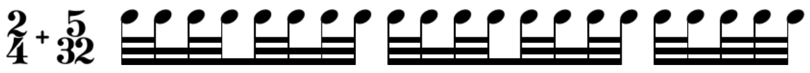
Figure 1: Subdividing in compound meter (as most of the examples in this document this had to be created manually in an inacceptably tedious way)

It is obvious that a new property `subdivisionInterval` has to be created that is independent from `baseMoment`. I suggest to make this an integer that corresponds to LilyPond's notion of durations. *(NOTE: this would limit beam subdivision to 1/x intervals and rule out values like 3/16. I'm not sure but doubt that's acceptable. So this has to be reconsidered, including its implications for the actual implementation.)* A convenience function `\subdivideBeams` should be added that accepts an integer or a boolean as argument and sets both context properties. `##f` would of course disable subdivision while `##t` might set it to the value of `baseMoment` (which would even make the new code more compatible with existing scores). It should be possible to encode the example with

```
\relative a' {
  \compoundMeter #'((2 4) (5 32))
  \set baseMoment = #(ly:make-moment 1/32)
  \set beatStructure = #'(8 8 2 3)
  \subdivideBeams 16
  \repeat unfold 16 a32
  \repeat unfold 5 a32
}
```

and automatically get the displayed result. In more regular time signatures this will make the process even simpler, as currently one more or less always has to switch subdivisions on *and* change `baseMoment`. In a typical file it would simply look like this:

```
\relative a' {
  \subdivideBeams 1/8
  \repeat unfold 32 a32
}
```

Reasonable presets for `subdivisionInterval` should be defined in the time signatures' `beamExceptions`.

Separating `baseMoment` from `subdivisionInterval` has another implication that may be perceived as an additional complexity but actually matches musical reality. The length of a beat is defined as a multiple of `baseMoment`, so subdivisions always fit cleanly into a beat. This is no more the case when the interval is an independent value. In the previous example the subdivision of 1/16 does *not* fit into the last beat of 3/32. This means that *whenever* the beat length is not a multiple of `subdivisionInterval` there must be *no* subdivisions in the beat, even when a stem should fall on a multiple of the interval. This check is necessary to avoid wrong subdivision as in figure 2 on the next page

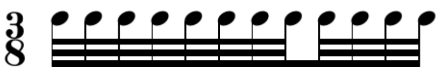

Figure 2: Wrong subdivision in a 3/8 measure with `subdivisionInterval` 1/4

General Identification of Subdivisions - `subdivide()` (Without Tuplets)

This section describes the process of determining the nominal beam count for all stems in a beam when no tuplets are involved. The specifics of dealing with tuplets are discussed in the next section, while all further considerations regarding the actual *printed* left and right beamlets are discussed in a later part. It partly reproduces the existing method `Beaming_pattern::beamify()` and should be implemented as a new function `Beaming_pattern::subdivide()` which is called from `beamify`.

For each stem we determine the position (Moment) relative to the current *beat*. We can't work with the position relative to the *measure*, since that may produce wrong results in some cases. For example in a 5/32+2/4 time (the opposite of the compound time in the previous example) all the positions in the 2/4 part would be "off" the grid by 1/32.

The property from which we can determine whether a stem is at a subdivision is its `rhythmic_importance`. This is simply the denominator of the `Moment` obtained in the previous step. For example the moments of eight 32th notes in a quarter are `0/1` (or 0) – `1/32` – `1/16` – `3/32` – `1/8` – `5/32` – `3/16` – `7/32`, and the corresponding values for `rhythmic_importance` are 1, 32, 16, 32, 8, 32, 16, 32. This works because LilyPond's `Moment` class automatically shortens its fraction *(the denominator of any Moment with numerator 0 is 1).*

A stem represents a subdivision when its rhythmic importance is less than or equal to the subdivision interval. In this case the subdivision occurs to the left of the current stem. *(NOTE: this may not work properly when we allow subdivision intervals other than 1/x.)*

Each subdivision has a nominal beam count that corresponds to the rhythmic importance of the stem and can be calculated by `intlog2(rhythmic_imporance) - 2`. The actual number of beams may be modified in a later stage, but for now we just make a note of this information. *(It has to be seen if we should actually store that in a stem property or if it's sufficient to calculate it later upon request.)*

Note: For moments longer than 1/8 (i.e. `rhythmic_importance < 8`) this calculation will result in a nominal beam count of 0 or less. But at some point in the end of the process we will have to ensure that at least 1 beam is printed. Recall that we don't decide whether the notes are beamed at all (and 0 beams at a subdivision would actually break the beam).

Stems *on* a beat are a corner case of this analysis and have to be treated specially. Their `Moment` relative to the beat is 0/1 (or 0), and their nominal beam count would be -2. Instead the nominal beam count of the subdivision on a beat is determined by the beat's *length*. For durations different from 1/n we have to apply a special shortening operation that actually produces the *floor* of the matching duration. For example, in a 5/16 time with `baseMoment` 1/16 and `beatStructure #'(2 3)` the beats are 2/16 (= 1/8) and 3/16. But both beats have a `rhythmic_importance` of 8 because they are `>= 1/8 < 1/4`. To achieve that we shorten the fraction by 2 and floor the numerator in each iteration until the numerator becomes 1. The most efficient way to do that in a loop where both numerator and denominator are shifted right until the numerator is 1. The resulting denominator will be the `rhythmic_importance` of the beat's stem. *(NOTE: This functionality is already needed, so it is very likely that we can use the* *existing code.)*

Note: Currently LilyPond actively checks for shortened beams and increases the count for subdivisions in the last beat of the beam. This is due to a misunderstanding of an ambiguous remark in "Behind Bars" (p. 156/57) which speaks of the "duration of the groups" and doesn't state what should be done when a group is shortened by a rest. However, Elaine Gould actually speaks of the duration of the *beat*, whether it is fully beamed or shortened by rests.

Tuplets

The subdivision of tuplets is completely messed up in LilyPond and has to be rebuilt from scratch. Currently tuplets are recalculated to their actual `Moment`, and subdivisions are determined from there. This can easily be seen from figure 3, where a subdivision interval of 1/8 causes a subdivision at the *actual* 1/8 position. The correct rendering would divide the triplet in three equal segments instead, as is shown in figure 4

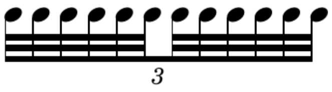

Figure 3: Automatic rendering, the subdivision placed at the *absolute* quaver

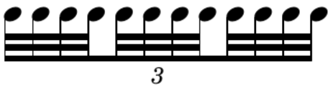

Figure 4: Correctly subdivided triplets

In fact tuplets resemble beamed grace notes in so far as their actual timing is ignored and they are handled according to the *visual representation* instead. The basic idea of tuplet handling is to *skip* the tuplet and have it handled separately. A good way to demonstrate how this works is to look at an example of a *shifted* tuplet (figure 5):

```
c32 c
\tuplet 3/2  {
  \repeat unfold 12 c64
}
c32 c
```

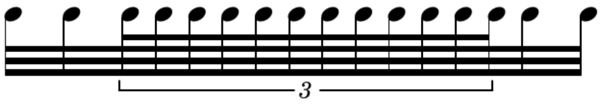

Figure 5: Shifted tuplet, without subdivisions

The beam spans 1/4 in total, a triplet starting on the second semiquaver and spanning a quaver (i.e. going over the middle of the beat). By the way, this is a good example for the importance of subdivisions, and we would definitely want this beam subdivided by 1/16 intervals. But if we request that LilyPond again produces wrong results, as shown in figure 6 on the next page.

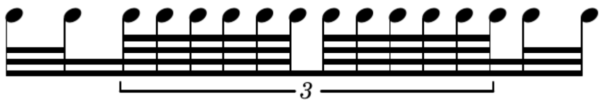

Figure 6: Shifted tuplet within a beam. Default rendering by LilyPond with partially wrong subdivisions

While correctly identifying the subdivisions after the first and third semiquaver, LilyPond finds a subdivision at the *absolute* 1/8. However, the correct rendering would treat the triplet independently, as shown in figure 7. (Actually this is a diminshed form of the triplet given in figure 4 on the previous page as the first example in this section.)

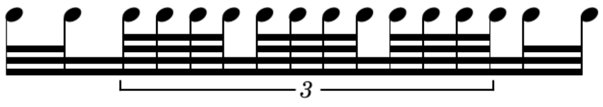

Figure 7: Correct rendering, tuplet spanning over middle of the beam

The tuplet is an independent entity of actual length 1/8, generic length 3/16, and a `baseMoment` of 1/16. This has to be implemented as a form of "local beat structure" as is discussed below.

This local beat structure is totally isolated from the surrounding beat structure. Figure 8 shows the same triplet starting at the *second* 32th note of the beam, and although I must admit I'm not sure there's an authoritative way to subdivide I find this rendering correct.

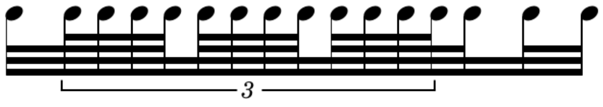

Figure 8: Shifted tuplet starting at an undivided stem

The same is true for *nested* tuplets. These too should be treated conceptually just like regular tuplets: any subdivision consideration is done relative to the visual representation, regardless of the absolute position in the measure (figure 9 on the next page).

As a general rule the *first* element of a tuplet is treated by its regular `Moment`, so the tuplet itself is actually processed only from its second element onwards. The first element *after* the tuplet is again a regular element and can ignore the fact that the tuplet has happened at all.

Encountering Tuplets

As seen in the previous section tuplets have to be handled separately and processing picked up regularly after the last stem in the tuplet. When encountering the start of a tuplet we will

- calculate the *first* stem of the tuplet as part of the outer beam: when the tuplet starts at the third 32th the first stem will be treated like one at the third 32th,
- pass along the whole tuplet to some separate processing (see below),
- determine which stem comes after the tuplet, and
- continue processing with that stem.

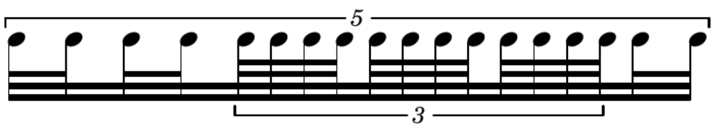

Figure 9: Nested tuplets. The subdivisions apply to the *visual* representation and don't care about position in the measure

> **NOTE:** Right now it is not clear to me how to determine the number of stems belonging to the tuplet. As far as I can see all we know is the `fraction` of a stem's duration, and if it's the *start* of a tuplet. What we need is a way to tell how many stems belong to that tuplet, either by inferring from the `fraction` property (sounds unsafe) or by passing that information into the Beaming_pattern at an earlier stage.

Handling `tupletSpannerDuration`

Using `\tupletSpan`, `tupletSpannerDuration` or the shorthand `\tuplet 3/2 8` it is possible to enter multiple tuplets within one `\tuplet` expression. With regard to beaming and beam subdivision these multiple tuplets have to be handled independently. As explained in the previous section the first element *in* the tuplet and the first element *after* the tuplet are treated like regular elements, and this applies to consecutive tuplets as well.

> **NOTE:** I don't know where we can get this information from in C++. From the wording in the NR "Entering several tuplets using only one `\tuplet` command"[2] I assume that this has already been resolved in the parsing stage. So I *assume* we will have each tuplet's first stem marked by the `start_tuplet` property.

Figure 10 throughout figure 14 on the next page show how tuplet grouping and consequently beam subdivisions should follow the `tupletSpannerDuration`.

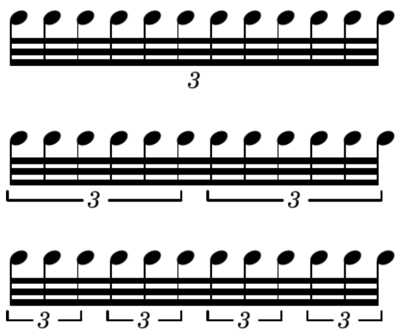

Figure 10: Different groupings depending on `tupletSpannerDuration`

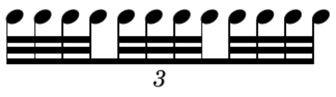

Figure 11: Triplet of length 1/4, subdivided by 1/8

[2]http://lilypond.org/doc/v2.19/Documentation/notation/writing-rhythms#index-tuplet-formatting

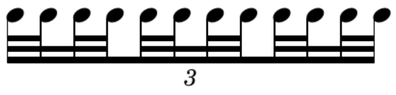

Figure 12: Triplet of length 1/4, subdivided with 1/16

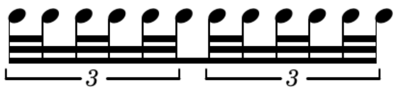

Figure 13: Triplet of length 1/8, subdivided by 1/16

Tuplets That Can Be Shortened

Sometimes tuplet fractions can be shortened, for example 6/4 or 12/8. By default (i.e. in most styles of Common Western Notation) these *should* be shortened while adjusting `tupletSpannerDuration` accordingly.

```
while tuplet can be shortened:
- shorten tuplet;
- expand denominator(tupletSpannerDuration) accordingly

6/4 4 => 3/2 8
12/8 4 => 6/4 8 => 3/2 16
```

Figure 15 on the following page and figure 16 on the next page show how the beaming matches that of the shortened tuplets.

NOTE: This is only the default case, and other options are discussed in the next section. It is likely that also the implementation should rather make use of default "local beat structures" rather than shortening tuplet fractions.

The Tuplet's Local `beatStructure`

As discussed above a tuplet has to get its own isolated structure, and discussion on the mailing lists[3] came up with the idea of giving tuplets a "local beat structure".

> **NOTE:** This is an issue that has to be dealt with on another level. At least the automatic-beam-engraver is affected by the same shortcomings as the beaming-pattern code, and probably the necessary behaviour is something that should be handled directly in the representation of tuplets themselves. The discussion below refers to that general level, and I assume it is directly applicable to the subdivision issue, i.e. when tuplets are fixed generally we'll have to see if anything remains to be done for subdivisions or not.

[3]**TODO:** Link

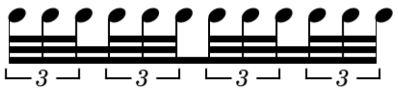

Figure 14: Triplet of length 1/16, subdivided by 1/16

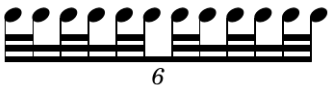

Figure 15: 6/4 tuplets, beamed like in figure 13 on the previous page, but with different tuplet number

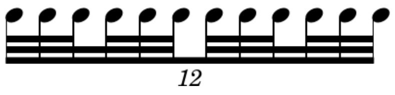

Figure 16: 12/8 tuplets, beamed like in figure 14 on the previous page, but with different tuplet number

A tuplet contains its own beat structure that always begins with the start of the tuplet, independently of how it is anchored in the surrounding beat structure of the measure (or an enclosing tuplet). This beat structure has a fixed `baseMoment` that can directly inferred from the tuplet's denominator and the `tupletSpannerDuration`. The base moment is the length of the tuplet divided by the denominator, or: one over the product of numerator(tuplet) and `tupletSpannerDuration`. In a `\tuplet 3/2 8` expression the base moment is `1 / (2*8) = 1/16`. The number of base moments is the tuplet's numerator (here: 3), and the tuplet can be divided in different beat structures like any time signature. There are default beat groupings (that should be provided as defaults), but a user should be able to override them like with time signatures. For example sextuplets are usually grouped `(3 3)`, but they can also have `(2 2 2)`, quintuplets or septuplets are usually not dividable at all, but could also have groups.

> **NOTE:** This requires new context properties whose design and terminology have to be discussed. One additional question that has to be discussed is whether beats in a tuplet's structure will split beams, will not split beams or if that should be made configurable.

I *think* it would be best to have the tuplet handled through a new `Beaming_pattern` object that is created from the stems contained in the tuplet. If we know which (or how many) stems that should be we can probably write a function `Beaming_pattern::extract_pattern` modeled after `Beaming_pattern::split_pattern`. Using such a new object would also be best for handling nested tuplets, as these would be automatically handled recursively by the code. The local `beatStructure` that has to be passed into such a temporary `Beaming_pattern` can be derived from the tuplet properties, as shown above.

Calculating Actual Beamlet Counts

What we have by now is each stem's actual duration and regular beam count. Additionally we have identified stems to the right of subdivisions together with their nominal beam count. However, the actual number of beams printed to the right and left of a stem may vary depending on several factors such as the beam count of the stems to the right and left, shortened stems and the property `strictBeatBeaming`.

I have not looked into this topic yet, but maybe it's possible to simply reuse the current logic and (partially) code for that anyway. In any case the topics so far should be fixed before proceeding with the finalization stage.