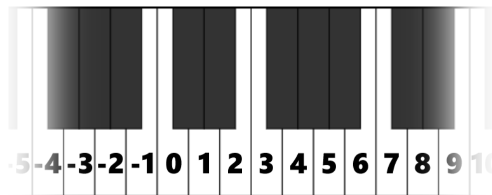
A Few Algorithms for Musical Harmonization

Neil Bickford

You can try out two of these algorithms online at https://neilbickford.com/G4G14/index.htm!

Here's a neat way to generate a harmony from a melody.

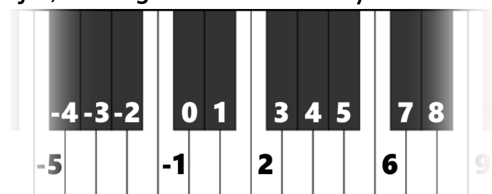
Let's say we have a musical major scale — this could be C major, D-flat major, D major, or any other major scale. Number the notes in the scale consecutively, using the number 0 for the first scale degree in some octave. For instance, for C major, we might number the keyboard like this:



This makes it so notes with the same degree always have the same number taken mod 7.

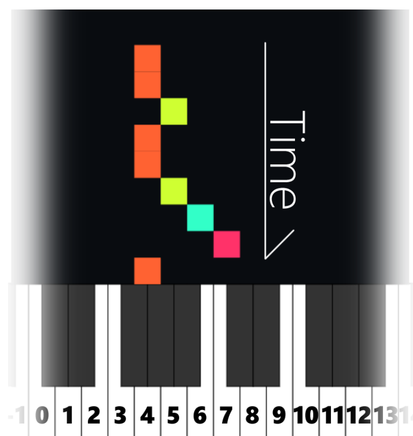
This is a bit like the Nashville Number System, but here we've subtracted 1 from all numbers, and we have an unbounded range so we don't have to denote the octave separately.

And for D-flat major, we might number the keyboard like this:



Now, let's consider a melody using this numbering system. We'd like to generate three additional musical parts, using pitches below the melody, that harmonize well with the melody.

Traditionally, the four total parts are named the bass, tenor, alto, and soprano voices (as if we were writing for a SATB choir), counting from the lowest part to the highest part. However, this algorithm doesn't ensure that each part is within the typical singing part range, and it also doesn't always follow voice leading rules.



This shows the series of notes 4, 4, 5, 4, 4, 5, 6, 7, 4 in C major.

We'll start by creating three-note chords (triads) in three different positions, which we'll call position 0, position 1, and position 2.

This article's positions 0, 1, and 2 are also known as triads' root position, first position, and second position.

To create a position 0 triad, take the melody note *n* and add notes *n-2* and *n-4*. For instance, if our melody note was 9, we'd create the position 0 triad {5, 7, 9}. The *root note, r,* is the lowest note of the chord in position 0 (here, it's *r=n-4*).

To create a position 1 triad, start with the chord {n, n+2, n+4} (which is a position 0 triad with the melody note as the root). We want the melody note to be the highest part, so we'll *invert* the chord by taking all the notes above the melody and subtracting 7 to move them down an octave, then sort them from lowest to highest. This gives the chord {n-5, n-3, n}, with root note *r=n.*

Finally, to create a position 2 triad, follow the same procedure, but start by building a position 0 triad where *n* is the middle note. This gives the chord {n-5, n-2, n}, with root note *r=n-2.*

To add the bass part, add the root note shifted down an octave (*r-7*) —unless the root note is congruent to 6 mod 7! In that case, add *r-9.*

What's going on here is that when r=6 (mod 7), we get a tritone between the root and one of the other notes in the chord! Moving the root down two more notes turns this into an inversion of a dominant seventh chord, which usually sounds less dissonant to most listeners.

Another way of summarizing the above is that we'll generate one of three chords:

- `Position 0: {n-11, n-4, n-2, n}`
- `Position 1: {n-7,  n-5, n-3, n}`
- `Position 2: {n-9,  n-5, n-2, n}`

then if the lowest note is congruent to 6 mod 7, we subtract 2 from it.

To harmonize a melody, take each melody note in turn, generate the chord above for it, and then randomly choose one of the other two positions. Importantly, we never repeat the same position twice in a row!

This algorithm's pretty compact! The core JavaScript code from this article's website for this fits in this sidebar:

```
If (nextPosition == 0)
  chord = [n-11,n-4, n-2, n];
else if (nextPosition == 1)
  chord = [n-7, n-5, n-3, n];
else
  chord = [n-9, n-5, n-2, n];

if((chord[0] % 7) == 6)
  chord[0] -= 2;

nextPosition =
  Math.floor(
    nextPosition + 1
    + Math.random() * 2
  ) % 3;
```

Here's an example of a harmonization generated using this algorithm on the melody above.

Here we've chosen each new position randomly, but we could choose it deterministically instead. We have a choice of 3 positions for the first note and 2 positions for each subsequent note, for a total of $3\ 2^{m-1}$ possible harmonizations of an m-note melody. Knuth also shows how to steganographically encode information this way, as a stream of a base-3 integer followed by m-1 bits.

This algorithm comes from Chapter 22, *Randomness in Music*, of Donald Knuth's book *Selected Papers on Fun & Games*, where he attributes it to a 1969 class from David Kraehenbuehl (1923-1997) at Westminster Choir College. I've rephrased it a bit in the presentation above.

Procedures for creating harmonies have existed for a while, although they're usually phrased as a set of constraints. The extra step of choosing a random harmony that satisfies the constraints sometimes turns it into an algorithm in the usual sense.

A set of constraints can also be a program! One can write a Sudoku solver in Prolog by specifying the rules of Sudoku and the initial clues as CLP(FD) constraints; Prolog's constraint solver will find a solution.

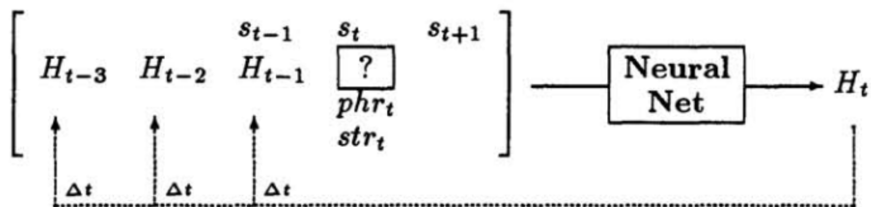
For instance, Johann Joseph Fux's 1725 *Gradus ad Parnassum* describes a set of rules for constructing each of four kinds of counterpoint, such as beginning and ending on consonance, avoiding tritones, and avoiding parallel fifths and octaves.

It turns out that the first few chapters of Peter Ilyitch Tchaikovsky's *Guide to the Practical Study of Harmony* have guidelines that come relatively close to describing Kraehenbuehl's algorithm above! Section 9 of (from the 1900 English translation) describes the rules above for the three triad positions above and the bass part (although it doesn't include the rule for moving the bass note to create a dominant 7th chord). §14 describes how Kraehenbuehl's algorithm never uses the same position twice in a row in terms of avoiding parallel fifths and octaves, with the constraint that

Two triads, however closely related internally and externally, must never directly follow each other in the same position, as parallel fifths and octaves must necessarily occur.

In addition to algorithms like the ones described above, some musical harmonization algorithms take a corpus of existing harmonized pieces, then try to construct a statistical model of what harmonies in the corpus tend to look like. They can then generate harmonies for new melodies by choosing from the distribution of harmonies they think are likely.

Hild, Feulner and Menzel (1991) trained a system named HARMONET on a collection of 400 Bach chorales. The neural network (or neural networks – it can use one, or choose between the outputs of three) tries to guess the next symbolic harmony given the previous harmonies, the previous, current, and next notes, and the rhythmic location of the note in each bar.



From HARMONET: A Neural Net for Harmonizing Chorales in the Style of J. S. Bach

More recently, David Li's *Choir* and *Blob Opera* are web applications that also generate four-part SATB harmonies using a neural network. There doesn't appear to be much information about the neural network they use (or, worryingly, about the dataset it was trained on!), but its input and output format looks similar to Liang, Gotham, Johnson, and Shotton's BachBot (from *Automatic Stylistic Composition of Bach Chorales from Deep LSTM, 2017)*.

Both Choir and Blob Opera use the same neural network – or, at least, the same weights. Blob Opera has an additional network to synthesize sound.

BachBot uses a Long Short-Term Memory (LSTM) architecture, which essentially is a deep neural network that transforms vectors to other vectors while keeping some internal memory state.

BachBot's source code is available online at https://github.com/feynmanliang/bachbot.

BachBot takes as input a series of 16th-note *frames*, each of which contains a melody note (continued or not from the last frame) or a fermata (denoting the end of a musical phrase). These are embedded into vectors and passed one by one to the LSTM, which outputs a probability distribution. Liang et al.

then optimized the parameters of the model so that it tended to assign high probabilities to the full harmony of the Bach chorale for that frame. When running on new melodies, BachBot inputs frame and melody symbols to the LSTM, retrieves the note probability distribution output by the LSTM, and chooses the harmony consisting of the highest-probability notes. If we wanted BachBot to make more unexpected decisions, we could instead have it randomly sample from the probability distribution.

However, It's possible to miss in the above discussion that constructing harmonies is an art. There are many harmonies Kraehenbuehl's algorithm above can't construct. The neural network-based methods would be unlikely to guess surprising but sublime melodies, and also usually have no way to artistically collaborate with a user. The algorithms here also generally have no concept of the emotions of the piece — Kraehenbuehl's algorithm will choose a random harmonization at each step!

Breaking melodic patterns can be a powerful tool. Additionally, books on harmony sometimes have conflicting guidelines, and while that defies computer implementation, that's okay: Tchaikovsky, for instance, writes of weighing which conflicting rules to follow, or of breaking earlier rules artistically.