INNOPOLIS UNIVERSITY

Introduction to Artificial Intelligence

Assignment 2

Maksim Surkov

November 2017

Contents

Description of task

We will start with a brief description of the task. Given problem was about generation midi file containing short musical composition. Basic rules are the following: chord accompaniment for left hand with 16 quarter notes, single-note melody for right hand with 32 eighth notes. Four consecutive repeats, notes outside range 48-96, >12 note jumps are considered as bad. Limitation of the task was to generate chords and melody using Particle Swarm Optimization (next PSO).

What is PSO?

It is a method that is usually used for finding global function maxima on defined domain. The core idea of the method is using a swarm of candidate solutions generated randomly where each particle acts in respect of best-known position in a swarm and its own best-known position. Using mathematical method of assigning speed to each particle allows swarm to explore domain, find better positions, and as a result converge at the position of global best fitness. Fitness of particular position is calculating using Fitness function. Function can be set up not only for evaluation of position as actual coordinates but for evaluation of certain properties of position. That allows us to use PSO for such task as music generation. Main drawback of PSO, as well as other stochastic methods, is that it may not converge to globally best solution, especially on inconsistent domain.

How can it be applied to music?

We can generate sequence of notes that follow certain properties: chords(in our case triads) are built on tonic, subdominant and dominant notes of major or minor scale. Distances between notes of triads are called intervals. According to mode, we use (major third + minor third) for major triad, (minor third + major third) for minor triad and etcetera. All the distances in chords are measured with semitones. Each note at a lower level of abstraction is a unique frequency. That is all about numbers and math, which makes it possible to evaluate music with fitness function.

What is a beautiful music?

One of the requirements for generator outcomes is beautiful music. So, let's define what beautiful music is. Simply speaking throughout the history humanity created music theory. It generalizes music to set of rules. And following this rules results actual music. Question is the following – anyone can learn music theory and start creating beautiful music? Actually yes. That means we can teach a machine music theory and it will produce masterpieces. Not really, "Beautiful" does not imply "memorable". We do not consider every beautiful melody as masterpiece. Composers do not just write random notes, listen to them and conclude if they are beautiful, they spend lots of time to measure music from mathematical and aesthetical sides, it is necessary for every part of their creation to be perfect. Memorable music is about complex patterns that are variated through whole composition. In addition, numerologists believe that creators of immortal classical masterpieces used golden ratio in many aspects of their compositions.

Applied solution

There are many ways to approach the problem. **One of which is to use single PSO at a time to generate chord by chord.** However, that involves making bridge to connect PSO's by passing generated chord to successor PSO. The task is not about generating chords but generating music. Building chord is deterministic and such approach can result lower involvement of patterns. Nevertheless, chord by chord solution was tested as well and results were compared with opposite approach. The conclusion was the following – using separate PSO is good for targeted result but not an intermediate, music appears to have more sense and stochasticity when generated as a whole piece.

Opposite approach is to use notes of chord bases or melody as one particle, which can be 16-dimensional for only chord bases, 48-dimensional for all chords, 32-dimensional for melody. As tested using particles with more than 20 dimensions decrease performance and what is more important success of PSO drastically.

However, doing PSO to generate only chord bases and building chords automatically is kind of cheat, because in such manner we could generate whole piece. Final approach was achieved by mixing both of previously described methods: at first we generate chord bases using PSO (called PSO in code) with 16-dimensional particles, then for each chord base we run separate PSO (called PSO1 in code) with 3-dimensional particles, after that we have final chords. Then we use PSO (same code as first PSO, just another fitness function passed by pointer) to generate melody (using already generated chords for fitness) with 32-dimensional particles.

As a result generating first two stages showed good result on quite a low particle count/iterations (5000 particles, 400 iterations for chord bases and 1000 particles, 1000 iterations for each chord), however melody required more resources (15k particles, 1.2k iterations). And even then result had a bit of error (some notes are off tonality by fracture N of midi note, 0.5≤N≤1).

Top-view of algorithm is presented below.

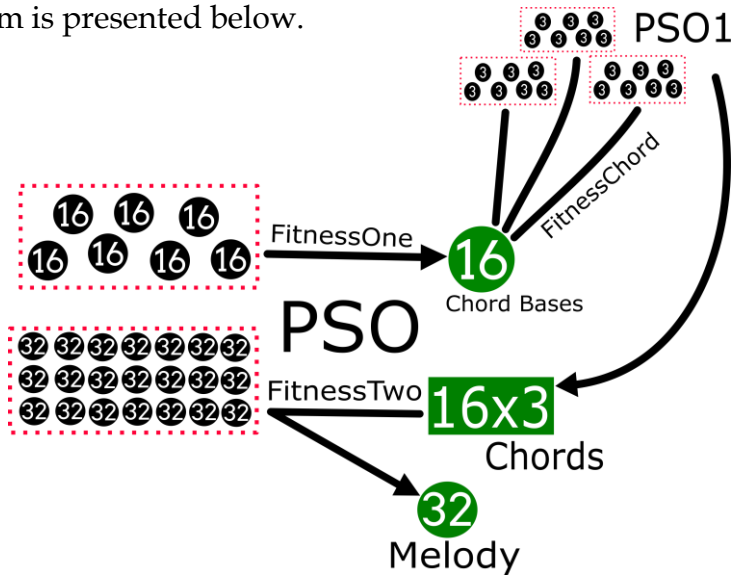


Figure 1. Basic scheme of algorithm

Code comments

The program is written in C++ using some C constructions for optimization. Creation of midi file happens in external Python3 script with a help of "midiutil" library. Actually, final code is the fifth try to approach the problem.

Code consists of 7 files:

Application.cpp – logic of initializing PSOs and constructing output

PSO.h, PSO1.h – particle swarm optimization realizations

Operations.h – small class for vector operations used in PSO

Fitness*.h – three files with fitness functions

Tonality is represented as a number in range 0-11, all the other notes are the same notes with addition of 12*N so all notes can be compared to that lowest sequence using modulo operator. Tonality is generated at the start of application: bool isMajor and Tonic[0-11]. Because of small errors that arise from time to time a special checker was added to program. After each PSO it checks result and if there is a note that drops from tonality it shifts it. As tested that is needed only for a few notes of melody (right hand). All operations of checker are shown in console. (e.g. "75 fixed to 74")

To generate midi file you need to launch "generate.exe". Make sure that it is in the same directory with midi_gen.py script. After launch program will create data.txt for passing data to python script. It is written in following format: 16 chords, notes separated by space, chords separated by newline. 32 melody notes each separated by newline. Script just reads data from file and creates midi. It is necessary to have midiutil lib installed to order to launch the script. You can easily install it using "pip install MIDIUtil" if you have python with pip package.

Stopping condition was chosen to be iterations count but not certain fitness achieved. Fitness function gives a certain value, however the mapping (outcome) is completely different for each of the three. What is common fitness is always evaluated continuously (no fixed penalties for condition break), thus we can evaluate more precisely how "bad" is fitness. Approximately fitness is mapped in the following way:

FitnessOne, FitnessTwo: -N,+N, where +N is the best

FitnessChord: 0,+N, where 0 is the best. Fitness represented as a penalty.

Anyway don't pay much attention to best fitness achieved printed in console, it has no absolute limits and only printed to observe dynamics of fitness exploration.

You can find source code in "source" folder and assembled application in "build" folder. All musical examples are in "midi" with music generated by latest version of program and "DevExamples" with music generated by earlier versions of program including completely pythonic versions.

To run the program launch generate.exe, it will display progress in percents of completion as well as generated outcome after each PSO. After execution it will create txt file with notes. Program will also call python script by itself. However it won't work if script is not in the current directory. Anyway you can place data.txt near midi_gen.py and launch it individually. After generation done you can open midi file with according name, date-time of generation finish will be in filename.

If there are problems with system(…); call during compilation, you can delete it. In that case you will have to run midi_gen.py manually.

Description of fitness functions

Each fitness consists of sub-fitnesses that are weighted and simply summed up. During research it was noted that sub-fitnesses can be divided into several groups, and thus be weighted accordingly. There are easy-achievable fitness and hard-achievable fitness, hard achievable fitness should be set with much more influence. Also rarely-bad fitness such as "repetition fitness" should be given much influence as well, so PSO can drop it quickly in execution.

There are three general fitness functions:

- FitnessOne
  - Repeated: Sequence should not have more than 4 repeats in a row. Implemented by counting number of 5 repeats in sequence.
  - Ending: Sequence should end with tonic in any octave. Implemented by finding absolute difference between closest tonic by modulo 12 and evaluated note by modulo 12.
  - Fractured: Sequence should not have two neighbor elements with difference>12. More the difference the worse. Implemented by summing all exceeding differences for all pairs of notes. (e.g 50 77, exceeding = 15)
  - In Tonality: Every note should be either of three notes of tonality 1 (tonic), 4 (subdominant), 5 (dominant) (TSD sequence). That results three kinds of chord: tonic triad, subdominant triad and dominant triad. Last note is not evaluated, avoiding race with tonic-ending part. Implemented by finding closest TSD note for each note of sequence and summing their distances. Large distance is worse fitness.
  - In Range: Notes of sequence should not go out of range {48 to 77 (96-7[for chord] – 12[for melody])}. Implemented by counting notes inside range, more notes = better fitness.
- FitnessChord
  - Chord Shape: Chord should be in particular shape - lowest note should be the same as according chord base, second note should be greater by 4 (major) or 3 (minor) than bottom note, top note should be greater by 7 than bottom note.
- FitnessTwo
  - In Range: Notes of sequence should not go out of range {top note of according chord to 96}. Two notes of melody (e.g. i=0, 1) are compared to one chord (i = 0), so both notes of melody are higher than chord. That comes from reality - when pianist plays chord in left hand of quarter-duration, he cannot lift his left hand, so two eighth notes played in right hand cannot cross the left hand.
  - Repeated: Sequence should not have more than 4 repeats in a row. Implemented by counting number of 5 repeats in sequence.
  - Fractured: Sequence should not have two neighbor elements with difference>12. More the difference the worse. Implemented by summing all exceeding differences for all pairs of notes. (e.g 50 77, exceeding = 15)
  - Ending: Sequence should end with tonic in any octave. Implemented by finding absolute difference between closest tonic by modulo 12 and evaluated note by modulo 12.
  - In Chord: Odd notes of sequence should be either of three notes A+12*N where A is any note of according chord. Implemented by finding distance between closest chord note by modulo 12 and evaluated note by modulo 12.
  - In Scale: Even note of sequence should be in scale. Scale comes from tonic and mode. E.g. major scale from C 60-62-64-65-67-69-71(-72) or tone-tone-semitone-tone-tone-tone-semitone, minor scale is tone-semitone-tone-tone-semitone-tone-tone. Implemented by finding distance between closest scale note by modulo 12 and evaluated note by modulo 12.

Convergence complexity of PSO highly depends on fitness function, it is noted that more complex fitness functions are more difficult to optimize because PSO tends to walk around local bests in case of "hard paths". As a result, we see that FitnessChord with actually one fitness component is much easier to converge (number of dimensions also play role here). As for FitnessOne, given proper weighting, it has convergence around 99%, and most complicated fitness function, FitnessTwo has convergence ~90%.

Application performance analysis and possible improvements

Use of C++ allowed flexible optimization and memory management. With basic setup:

1) 5000 16-dim particles and 400 iterations take ~2.8 seconds, 4 MB

2) 1000 3-dim particles and 1000 iterations take ~0.4 seconds, 1 MB

3) 15k 32-dim particles and 1200 iterations take ~6 min, 15 MB

Max: Run time: ~6 min, Memory peak: 15 MB. There are several ways to improve performance:

First try was using parallel computation on the graphical card with NVIDIA CUDA, which can be quite useful. There was a try to apply it to vector operations in particle velocity/movement, while velocity calculation operations resulted almost constant operations, overall program was slowed down by 3 times. Why? Because each operation required moving data from RAM to Graphical Memory with cudaMalloc, which is costly procedure. So in order to observe CUDA effect task has to have more calculations and few memory movements. Another approach is to assign part of graphical memory to each particle and construct several blocks of threads to selected particle sections, it is important to have shared global best cell so that new global best could be chosen during iteration. Otherwise particles will seek around their personal best only. Such program is not portable as it requires CUDA SDK (several GB of disk space and tricky setup) and older VC compiler like one that comes with VS2013.

Second try was more successful, but it's quite simpler. AVX/SSE2 processor instructions allow us to use effective vector arithmetics – for example AVX instruction can sum 4 pairs of double (64-bit) numbers (or 8 pairs of floats) at a single processor cycle (with correction to instruction pipelining) with a help of YMM0 — YMM15 registers. SSE2 does the same but it's half the AVX capacity. Use of these instructions allowed to speed up program by few seconds.

So following function:

```
void lsl(double* dest, double* array1, double* array2, int size) {
	for (int i = 0; i < size; i++) {
		dest[i] = array1[i] - array2[i];
	}
}
```

Can be rewritten as:

```
#include "immintrin.h"
void lsl(double* dest, double* array1, double* array2, int size) {
	for (int i = 0; i < size; i += 4) {
		_mm256_storeu_pd(&dest[i], _mm256_sub_pd(_mm256_loadu_pd(&array1[i]),
_mm256_loadu_pd(&array2[i])));
	}
}
```

It is necessary to be cautious when using such instructions because they operate with blocks of size multiple 4 (2 for SSE2). Application version with AVX instructions is not included with given submission because of compatibility reasons. These instructions are not supported by all processors, so they can not be reliably included in release build. Also they are not compatible with PSO1 because 3-dim particles are not multiple of 4, and workaround will result overhead. Writing separate version for each PSO will be overhead in code size.

Some statistics and visualization

Few words about algorithm work visualization. As the first prototypes were done completely in python, it was convenient to develop special visualization methods. Usually PSO is visualized just assuming its dimensions as coordinate axes. However if we will draw particles in 16 dimensions, there would be no way to see them in a reasonable way. So they are translated into 2d, where x - note index, y- note, line of one color – chord. Below you can see such examples.

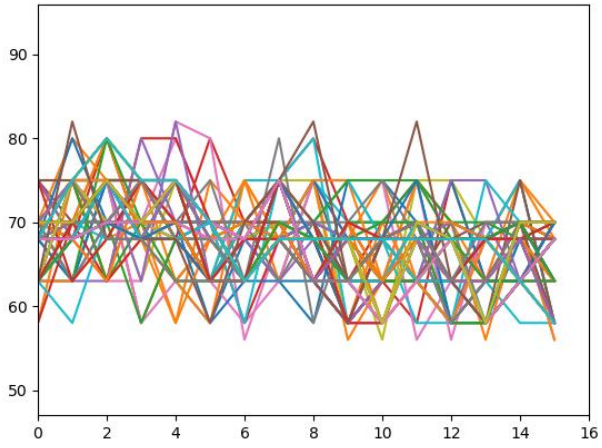


Figure 2. First prototypes of PSO, middle of execution

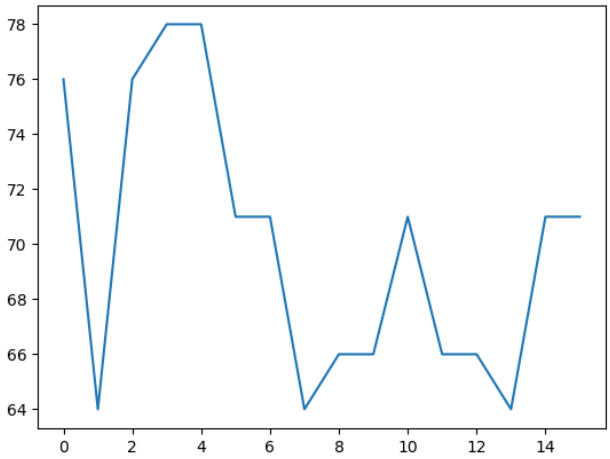


Figure 3. First prototypes of PSO, global best

In the final version visualization was done in another way, using PSO for chord (notes as coordinates xyz):

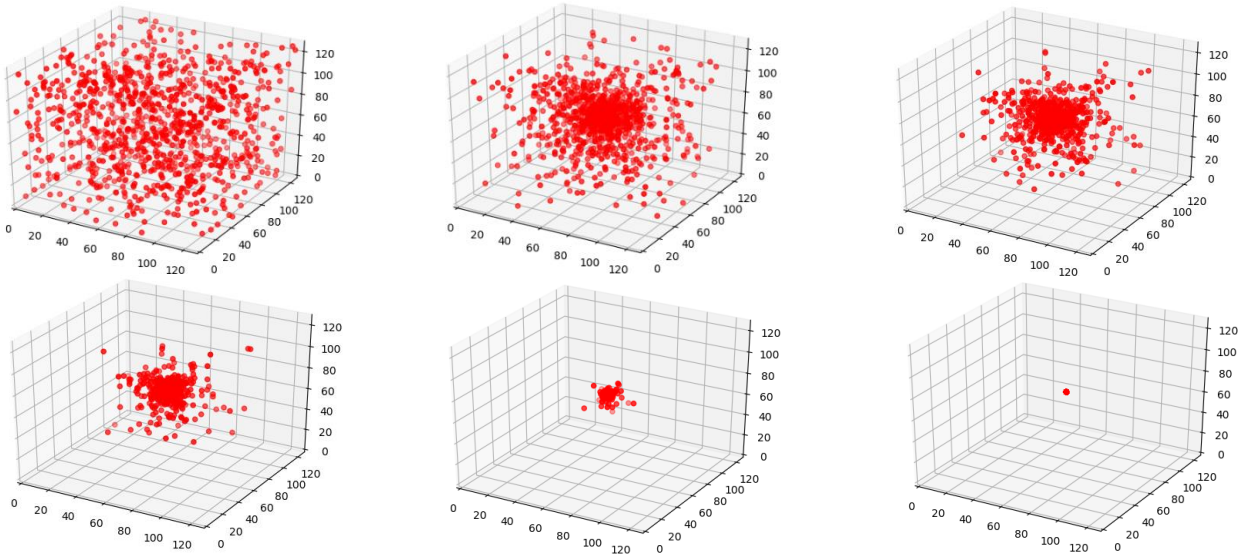
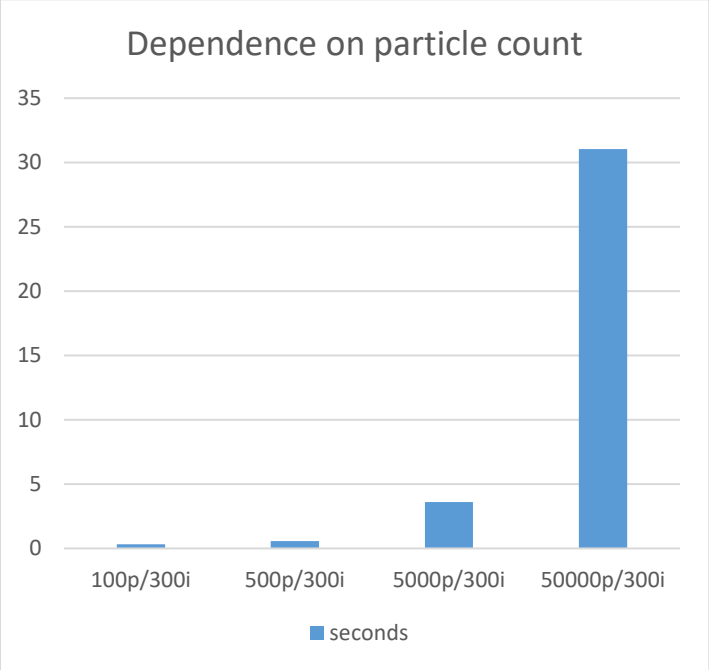
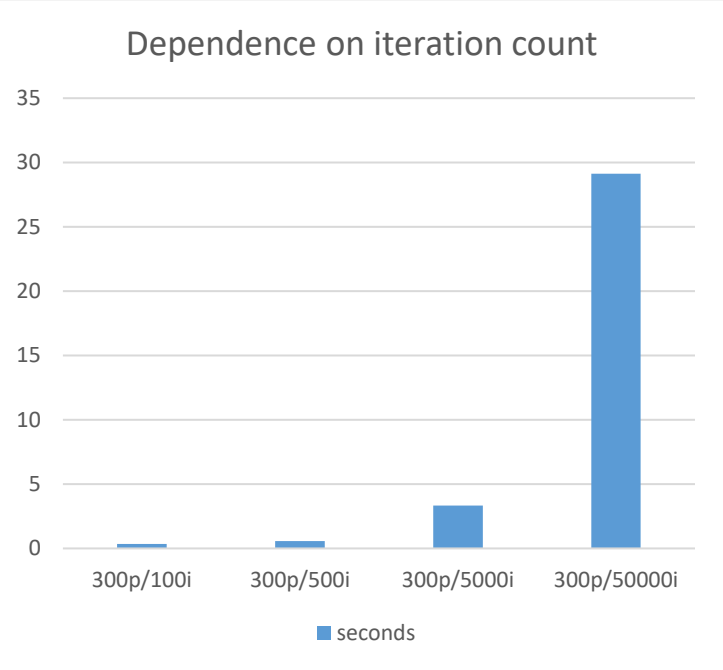


Figure 4. Periodic stages from ChordGen PSO

This visualization was done by exporting all particles data from ChordGen (Application.cpp, line:77) to special python script with "matplotlib module" real-time 3d projection. (not included in this submission)

Relation between particle count/iterations and time (p:particles, i:iterations):





Both graphs are results of testing on chord sequence generation. As we can see their relations P/I and I/P are almost equal. Relation on increasing particles/iterations is close to linear.

Conclusion

Given work has shown that it is possible to generate decent music using random samples and particle swarm optimization aimed at a set of music theory rules. However, given that melody is generated effectively with particles of low dimension count, it will not be useful when attempting to increase length of melody. There is possible workaround by generating fixed-size blocks and appending them with additional fitness.

Overall result quality is impressive. Resulting midi can be easily mistaken for real music composed by human. It does not guarantee masterpiece after every execution, but after a few runs there were some surprising outcomes with unique patterns. In this submission you can find several examples of midi files from latest version as well as older versions.

References

PSO mathematical explanation and realization examples: [1] (in Russian)

PSO wiki page: [2]

Intel AVX: [3] (in Russian)

Nvidia CUDA overview: [4]