**Learn live coding with Sonic Pi**

**A Tutorial for Young Musicians**

Dr Brendan McCloskey BMus MA LRSL MA PhD

2024-2025

Sonic Pi is a free software program for making music with code. We can type instructions – live and in real time – to control many different elements of musical sounds, including pitch, duration, timbre, shape, and Sonic Pi includes many different sound sources (synthesizers, sound samples and loops). It was designed and created by Dr Sam Aaron as a tool to help young musicians learn to code. And it’s amazing.

Installation instructions for Sonic Pi on Windows, Mac and Linux can be found here:

<https://sonic-pi.net/>

Installing Sonic Pi within Linux is a little more involved, so this tutorial assumes either a Mac or Windows OS. After launching Sonic Pi, the user is presented with a single window, comprising three main sections or panels. Make the application window as large as you can, so you can see each of the three main panels, highlighted in green below.

A screenshot of a computer

Description automatically generated

The main area is the top left panel, where we type, run and save our text/code. The upper right panel contain several smaller panels – they include a visual scope, preferences, and a cue/log panel - more on these later. The third main panel runs along the bottom and contains help files, tutorials, links to sound files etc.

Before we begin writing code, and just to make things simpler, let's temporarily close a few of those interface panels – we'll reopen them later when we need them. At the top menu bar, go to View and ensure the only three options with a tick are `Show Log/Buttons/Tabs/Help'. We can reactivate the other panels later.

Sonic Pi documents are called Buffers (also known as a Workspace), and you can see them numbered 0 – 9 along the bottom of the top-left panel. All code in a single buffer will be run at the same time, so if the code buffer becomes cluttered, simply choose a new empty buffer.

**Using play, sleep and simple loops**

In a code buffer (the larger left panel), delete any text or code that may be there, then type the following code exactly as written:

play 60

Click the Run button near the top left (keyboard shortcut on Mac is cmd + R; Windows alt + R). You will hear a short musical note. The **play** instruction tells Sonic Pi to play a note using the default sound – which we can change later. The number 60 is an additional instruction controlling how low or high the pitch of the sound is. Try changing the number to a higher or lower one. Numbers between 30 and 120 work best, but why do you think this is so? Smaller, lower numbers equal lower pitch, higher numbers equal higher pitch. We can also use musical notation such as :e3, :c2 etc, representing note name and octave.

Now add a second line as follows, but don’t run it yet.

play 60

play 67

What do you predict the sound will be? Writing play twice and running it plays two notes at the same time. Sonic Pi runs, or executes, the code immediately, with no gap between the two instructions. So, we could play a chord, using several play instructions with different note numbers. Run the following code and try other numbers too.

set\_volume! 0.5

play 58

play 61

play 66

play 70

Let’s now try to get the notes to sound one after the other, in sequence. To sequence events in time, we use the **sleep** instruction, which is given a duration in beats (0.5 = half a beat; 2 = two beats, etc). The default tempo of Sonic Pi is 60bpm. BPM stands for beats per minute, and at 60bpm, 1 beat = 1 second but try to think of duration in beats.

puts current\_bpm # this prints out the current bpm, or tempo

play 60

sleep 1

play 72

The hash (or number) symbol # allows us to write comments and helpful tips for ourselves and other users, which Sonic Pi will not see as code to run and thus ignore.

An important concept in Sonic Pi is that many instructions for making musical sound require additional information. A *function* (such as play and sleep) is a set of instructions to perform a particular task, and it needs further details in order to perform that task. If I ask you to make me a sandwich by saying “make me a sandwich please”, that is not enough information. You might ask “what type of bread, do you want butter, what do you want in it?”.

Type the following on its own:

play

When you run this Sonic Pi will show an error: “wrong number of arguments given”. The play function needs to know “ play WHAT?”, so we give it an *argument*, in this case a number or note name. Likewise, sleep will expect a number telling it how long to wait before moving to the next instruction. So, remember: functions (instructions) often require arguments (details).

Delete your code and type the following code into an empty buffer:

plan 72

Notice the log or debug panel – if there is an error in our code, such as this spelling mistake – Sonic Pi will indicate where the error is (View/Show Log). The Log panel is a helpful tool, as we will probably make a few mistakes, and Sonic Pi will let us know.

We will now explore how to tell Sonic Pi to play musical patterns more than once, by repeating a block (or paragraph) of code. If we wanted to write a very long piece of music, with certain patterns repeating, we could write out 100 lines of code with many, many play-sleep pairs for each note. This is time consuming and complicated.

We encapsulate (surround) the code block that we wish to repeat with the word **do** at the start and the word **end** at the, well, at the end. Then all we need to say is how many times to repeat the block.

Read the code below and try to predict what sound will occur before you copy and run it.

4.times do

play 60

sleep 0.5

play 72

sleep 0.5

end

Any code outside the **do/end** block will run only once, whereas the code inside the block will be repeated 4 times. The code inside the block obviously has to be valid and working – if the code works without the .times do-end block, then it should work inside this loop. Now try creating your own repeating patterns, with different note numbers.

Side note: yes, we can write a code block on one line, but it requires semicolons to separate instructions.

4.times do play 60, release: 0.4; sleep 0.5; end

We don’t usually write different instructions on a single line. Writing our code in blocks, or paragraphs, from top to bottom helps us keep track of what events are happening at what time.

Using the functions play and sleep we can write melodies with different pitch values (between 30 and 120), and duration values (commonly between 0.01 and 20, depending on the tempo), and we can repeat sections using .times do-end.

**Changing sounds in Sonic Pi**

Sonic Pi has several ways to create musical sounds. The most common way is to use the built-in sound generators, called *synthesizers* or **synth** for short. If you use the play function on its own, Sonic Pi will assume you want to use the default synth (called ‘beep’). There are currently 66 different types of synth in Sonic Pi. Run the following code:

play 60

sleep 1

use\_synth :beep

play 60

sleep 1

You will hear that the sound does not change, because the default synth is called :beep.

Listen to some other synth sounds with the following code:

use\_synth :bass\_foundation

play 48

sleep 1

use\_synth :dpulse

play 48

sleep 1

use\_synth :tb303

play 36

sleep 1

Here, we tell Sonic Pi to use a different synthesizer with the **use\_synth** function. In the Help files, click Synths and you will find a list of all the different synths available. All of them have different options (we will explore these later), which allow us to create our own sounds.

For now, we will just change a few of the synth’s default settings with the synth options **use\_synth\_defaults**:

use\_synth :dsaw

use\_synth\_defaults detune: 0.4, cutoff: 70, release: 0.25

4.times do

play 48

sleep 0.5

play 46

sleep 1

end

The options (called opts in Sonic Pi) **detune:**, **cutoff:** and **release:** control how out of tune the synth sounds, the colour (or timbre) of the sound, and how quickly the note ends.

When we run the code above, we will hear two alternating pitches, repeated four times, and the synth defaults do not change over time. We have changed them from the default settings, but they are static, unchanging, just like the repeating two pitches. How would I ask Sonic Pi to pick different numbers for the pitches, each time the loop repeated? This would introduce some variation into the simpler and unchanging pattern. There is a handy function called **rrand\_i( )**, which will choose a random whole number between the first number and the second number inside the brackets; for example, rrand\_i(48, 55) will choose random whole numbers in that range. If we need smaller decimal numbers, we can use **rrand( )**.

Change the code slightly, so that the note numbers, detune, cutoff and release opts are *inside* the .times loop (they will update with random numbers each time it repeats).

use\_synth :dsaw

8.times do

use\_synth\_defaults detune: rrand(0, 0.6), cutoff: rrand(40, 120), release: rrand(0.01, 3)

play rrand\_i(48, 55)

sleep 0.5

play rrand\_i(48, 55)

sleep 1

end

The synth defaults of detune, cutoff and release are being controlled by rrand(x, y), which chooses random numbers between the first number (x) and second number (y) inside the brackets, each time the block of code repeats; the note numbers or pitches are also being randomly chosen (between 48 and 55), with each repeat of the loop. We will explore randomisation in more detail later in this tutorial.

Let’s now try to recreate some well-known tunes, using different sounds or synths.

use\_synth :piano

use\_synth\_defaults release: 0.8

play 60

sleep 0.5

play 60

sleep 0.5

play 67

sleep 0.5

play 67

sleep 0.5

play 69

sleep 0.5

play 69

sleep 0.5

play 67

sleep 1

Try to recreate a well-known tune or write one of your own; explore the different synthesizers available in Sonic Pi. Notice that after you type use\_synth a dropdown menu appears, listing all the available synths. Other well-known tunes include Freres Jacques; Happy Birthday; Love Me Tender; 7 Nation Army; My Girl; Hey Jude. You should also be able to change some of the synth options. Use the Play button (top left) or use cmd or alt plus R to run your code; use cmd or alt plus S to stop it running.

**About arrays**

As we have already seen, musical notes (and many other things) can be represented in Sonic Pi by numbers, such as pitch and duration. When we want to write longer musical pieces, that might involve writing play and sleep many, many times, which can get repetitive, boring and confusing. Helpfully, Sonic Pi lets us make use of programming tricks to avoid such repetition.

We can collect all the numbers for our tune into a list, and in programming such a list is called an *array*, which we store inside two square brackets [ ]. I like to think of it as a box, or a container.

Let’s start with a simple tune in C major, such as the opening to Happy Birthday.

use\_synth :piano

use\_synth\_defaults release: 0.8

play 55 # :g3

sleep 0.25

play 55

sleep 0.25

play 57

sleep 0.5

play 55

sleep 0.5

play 60

sleep 0.5

play 59

sleep 0.5

It’s easy to see that, if we wanted to write out and play the entire tune, it would require a lot of repetition of the play/sleep functions. Using an array of notes for the tune would be more convenient. Type the following into Sonic Pi and run it:

puts scale(:c3, :major)

In the right-hand panel (View/Show Log) we see:

(ring <SonicPi::Scale :C :major [48, 50, 52, 53, 55, 57, 59, 60])

The eight numbers in the array (inside the box [ ]) represent the notes in one octave of the C major scale (note name :c3 = note number 48).

**Using scale and other arrays/rings**

The simplest function in Sonic Pi for creating and playing a pattern or sequence of notes is **scale**. Click the Help button (top right); two panels will open along the bottom, on the left is one called Docs, and on the right is the text of the selected document. Select the Docs tab called Lang, click on the list and scroll down to the word scale.

Type the following code and run it:

use\_synth :piano

use\_synth\_defaults release: 0.8

myTune = scale(:c3, :major)

8.times do

play myTune.tick, release: 0.4

sleep 0.25

end

We have created a *variable* (more on this later) called ‘myTune’ and told Sonic Pi that we want myTune to contain the notes from octave of the C major scale. The cool thing about a variable is you can name it however you wish. Change the word myTune (which appears twice in this example) to anything you like, maybe ‘frog’ or ‘onions’, just don’t use spaces.

tomato = scale(:c3, :major)

8.times do

play tomato.tick, release: 0.4

sleep 0.25

end

When you run this code, you will hear an ascending C major scale. Change .tick to .choose. What changed, and why? The *method* **.tick** (an extra instruction, in this case a loop counter) plays through the notes in the myTune array, in order. The method **.choose** does exactly what you expect – it chooses notes randomly from the array.

What happens if you change 8.times to 12.times, using myTune.tick? There are only 8 notes in our array. Look again at the Log panel and you’ll see that our array is a special type of a collection called a **ring**. This simply means that it will return to the start after reaching the end of the list of note numbers.

Each number, or *item*, in an array has a place number. There are 8 items in the myTune array, and we number them 0 to 7. So, the first note number (48) is item zero, and the last note number (60) is item 7.

The key difference between an array and a ring can be demonstrated as follows:

myArray = [1,2,3,4]

myRing = [10,11,12,13].ring

puts myArray[7] #there is no 8th item, hence 'nil', or nothing

sleep 1

puts myRing[5] #this loops back on itself

The array called myArray contains only 4 items (at index 0, 1, 2, and 3), so when I ask Sonic Pi for index 7 (there is no index 7 in that array), Sonic Pi responds with ‘nil’. The ring array called myRing also has just 4 items, but if I ask for the 6th item (index 5), Sonic Pi responds with the number 11. Can you work out why?

We will use arrays in this way later on, but for now, let’s return to the C3 major scale, and the tune to Happy Birthday. Run the following code:

6.times do

tick

play [55,55,57,55,60,59].look

sleep 0.5

end

The notes of the tune go g-g-a-g-c-b. The rhythm or durations are almost right. Inside the 6.times block, we put a tick at the start – an internal counter that keeps track of how many times the code is executed (6 times and counting starts at 0). The **.look** method simply uses that counter to read through the list of note numbers in the array from start to finish; but the durations aren’t right. We need two shorter notes at the start:

6.times do

tick

play [55,55,57,55,60,59].look

sleep [0.25,0.25,0.5,0.5,0.5].look

We can refine this further using the **knit** function, which eliminates the need to type out repeated sleep values. The knit function requires pairs of number – the value we want, and the number of repetitions. If I want the sleep value 0.25 repeated twice, followed by 0.5 repeated three times I would write knit(0.25,2, 0.5,3). Copy and run the following code:

puts knit(0.25,2, 0.5,4)

#this creates a ring array: (ring 0.25, 0.25, 0.5, 0.5, 0.5, 0.5)

6.times do

tick

play [55,55,57,55,60,59].look

sleep knit(0.25,2, 0.5,4).look

end

Another way to select specific notes from a scale array, besides typing them out by hand, is to use the place number of each item. If I want the 3rd item from an array, I write myArray[2]. As an example, run the following code:

use\_synth :piano

use\_synth\_defaults release: 0.6

notes = scale(:c4, :major)

puts notes

4.times do

play notes[0]

#play the 1st item in the array, note number 60 or :c4

sleep 0.5

end

The function play is using the 1st item from the scale array; change the zero to 5, and you’ll hear the 6th note (remember that we start counting items from zero, not 1). The place number of an item in an array is called its *index*.

In the following example, we create another array of numbers (called ‘tune’), each one representing a different index in the array. We now have an array called notes, containing the note numbers of one octave of the :c3 major scale, and we have a second array (called tune), which we use to choose specific items from the scale array:

use\_synth :piano

use\_synth\_defaults release: 0.6

notes = scale(:c4, :major)

puts notes

tune = [0,3,3,5,2]

#1st, 4th, 6th, and 3rd notes

8.times do

play notes[tune.tick]

sleep 0.5

end

Two other versions of the play function should be mentioned here: **play\_chord** and **play\_pattern\_timed**; from what you have learned already, you should be able to work out what they do. Suffice to say, play\_chord expects an array of different note numbers to play all at once, and play\_pattern\_timed expects two arrays (separated by a comma), one containing note numbers and the other filled with sleep values:

use\_synth :piano

play\_chord [60, 65, 69, 72]

sleep 2

play\_pattern\_timed [60, 65, 69, 72], [0.5, 0.25, 0.125, 1]

Lastly, there are several other array methods we can have fun with, to explore the scale array and write our own tunes. We can shuffle the items in an array, we can remove items, or start at a different index (called .rotate). For example:

use\_synth :piano

play\_chord [60, 65, 69, 72]

sleep 2

play\_pattern\_timed [60, 65, 69, 72].shuffle, [0.5, 0.25, 0.125, 1]

Type the following into an empty buffer: puts scale(:c3, :major), and copy the note numbers from the array in the Log panel. Run the following code to see what the different methods do:

myTune = [48, 50, 52, 53, 55, 57, 59, 60]

puts "the array", myTune

sleep 1

puts "shuffled", myTune.shuffle

sleep 1

puts "start at item 3", myTune.rotate(3)

sleep 1

puts "first 4 items", myTune.take(4)

sleep 1

puts "lose first 2 items", myTune.drop(2)

sleep 1

Now try writing your own tunes, or bass lines (with a different synth), using .times do-end, and different scale arrays. Take care to use only one .tick inside any .times loop. You should now be able to use the scale function in Sonic Pi to create a ring array of note numbers; you should also be able to create your own array of values, and then assign that array to a uniquely named variable. Remember, an array is simply a collection of things, or items, and each item has an index (place number), starting at zero. These methods above allow us to temporarily change an array, and in programming we call such methods *non-destructive*. Later we will learn how to use *destructive* methods.

Review the following methods that we can use on an array; if you don’t understand any of them, ask your tutor, and you can read more about arrays and rings in Chapter 8 of the built-in tutorial in the Help files.

tick (a loop counter)

.look (use the counter to read through an array)

.choose (select a random item)

knit (a handy shortcut function, for arrays with repeating numbers)

.shuffle (put the items in a random order)

.take(x) (use the first x items)

.drop(y) (remove the first y items)

.rotate(z) (start at item z)

**A brief introduction to live loops and samples**

Later in this tutorial, there is a more detailed discussion of two important functions in Sonic Pi: **live\_loop** and **sample**, but they will be briefly explained and demonstrated here, to allow us to compose and perform more interesting and exciting music.

A live\_loop is a block of code that repeats for ever, with the most important feature being that it can be changed and updated *in real time*, while other loops are still playing, and without interrupting the flow of the music. Similar to the .times do-end block, the live\_loop also has a do-end block, but it must also have a unique name:

use\_synth :bass\_foundation

live\_loop :tommy do

tick

play knit(40,2, 52,2, 43,2, :r,2).look, release: 0.5

sleep 0.25

end

use\_synth :sc808\_tomlo

live\_loop :kate do

play rrand(50,70), release: 0.5

sleep 0.5

end

This example uses two live loops, one playing a bass guitar line (:tommy), the other a simple single drum pattern (:kate); notice the ‘new’ or unfamiliar value inside the play array, :r – this is a rest or silence. The live\_loop :kate introduces an unfamiliar function, called **rrand( )**, which chooses random numbers between 50 and 70, and will be discussed in detail in the next chapter.

The key points to notice and remember are that all live loops must have a unique name (no spaces) and must also have a sleep value. If there is no sleep value inside a live loop, we are asking Sonic Pi to play the loop an infinite number of times, immediately, which is impossible, and the program will not run. If you do forget to include a sleep value, Sonic Pi will complain, and print the following error:

Runtime Error: [workspace\_seven] - Thread death +--> :live\_loop\_fred

loop did not sleep or sync! (SonicPi::Lang::Core::ZeroTimeLoopError)

While the loops are still running, change the note numbers inside the knit ring array, in the live\_loop called :tommy; change the play rrand(50,70) values to some other numbers. Notice that we do not need to stop and restart after making changes – this is the cool thing about Sonic Pi, the live\_loop allows us to change our compositions and performances live, in real time, without missing a beat.

Next, we will use another function called **sample**. We have already used play, to hear musical notes from the built-in synthesizers. This new sample function also generates sound, but it does not use the built-in synthesizers; it uses pre-recorded sound files (called *samples*), and there are around 200 of them in Sonic Pi. Add the following live\_loop to the two loops above:

live\_loop :sarah do

sample :loop\_amen, beat\_stretch: 2, amp: 2

sleep 2

end

The sample called :loop\_amen is a pre-recorded drum loop, and in order to make it play in time with the other loops, I have used the sample option beat\_stretch: 2, which we will learn more about later. Go to the folder in Help/Samples/Sounds for Looping and you will find several drum and percussion loops – change :loop\_amen to one of these other samples; don’t forget to start the sample name with the colon. Now create your own bass lines, percussion and drum loops.

**SIMPLE FX IN HERE – REVERB, PING PONG, BPF, SLICER and summarize**

**Using random numbers**

To help create our own music, we can ask Sonic Pi to use random numbers – for pitch, duration and many other sound options. We have already encountered the .choose method, but we can exercise a degree of control over how random choices are made in Sonic Pi. Do you remember the function rrand\_i( )? Run the following code once, which randomly chooses four numbers between 0 and 9, and prints them in the Log panel:

4.times do

puts rrand\_i(0, 9) #choose a random whole number between 0-9

sleep 0.5

end

Now, how do I know that this code will generate the following numbers: 7, 7, 4, 2? If you run the code again, it will select those same numbers again. What’s going on, not very random, is it?

Briefly, Sonic Pi does this for two main reasons: if I create a piece that uses randomisation, and I send it to you, I don’t want it to sound completely different. If you are trying to find a cool melody, or guitar riff or drum pattern using rrand( ), you want to keep the good ones, right? In programming, Sonic Pi (like many other programming languages) uses *pseudo-random* numbers, and this will be explored in a later chapter.

We can ask Sonic Pi to choose a random whole number (within a given range) for the pitch of a note, and also choose between different sleep values. This introduces some random variation into the loop.

use\_synth :dsaw

16.times do

play rrand\_i(48, 72), release: 0.2

sleep [0.25, 0.5].choose

end

The creator of Sonic Pi, Dr Sam Aaron, has written many examples of musical code and you can find them by clicking on Help/Examples. Open and copy-paste the example called Haunted; change some of the values inside rrand( ), to see what effect it has on the sounds. The **rrand( )** function is similar to the rrand\_i( ) function but it returns (generates) random decimal or fractional numbers instead of whole numbers; in programming a decimal number is commonly called a *float*.

Open the Ambient Experiment code, run it and change some numbers here too. You will notice another unfamiliar function here, called **with\_fx**. We will explore its use in more detail later on.

Conditional logic spread, dice( ), one\_in( ), if rrand >n

SUMMARIZE RRAND AND CONDITIONALS

**Playing with envelopes?**

Besides using different synthesizer sounds and sound file samples, we can also vary the ‘shape’ of sounds. All sounds have a shape in time – how and when the sound starts, how long it lasts, and how and when it ends. Run the following code and listen to the shape of the different sounds – how they evolve over time:

use\_synth :hollow

play 65, attack: 0, release: 5

sleep 5

play 60, attack: 3, release: 0

sleep 3

play 62, attack: 2, release: 2

sleep 3

play 55, attack: 0.4, release: 0.2

Notice how each note has a different duration, and they start and end in different ways. The evolution of a musical sound over time is called its *envelope*. The musical terms we use to describe the shape of a sound’s envelope are:

attack how quickly the sound starts

decay how long until the sound stabilises or sustains

sustain how long the sustain lasts

sustain level how loud the main level is

release how quickly the sound returns to silence

A diagram of a diagram

Description automatically generated

Returning to the Ambient Experiment example, add some random numbers to the attack and release values:

use\_synth :hollow

with\_fx :reverb, mix: 0.7 do

live\_loop :note1 do

play choose([:D4,:E4]), attack: rrand(2,6), release: rrand(2,6)

sleep 8

end

live\_loop :note2 do

play choose([:Fs4,:G4]), attack: rrand(2,6), release: rrand(2,6)

sleep 10

end

live\_loop :note3 do

play choose([:A4, :Cs5]), attack: rrand(2,6), release: rrand(2,6)

sleep 11

end

end

**USING SIMPLE FX**

**DRUM PATTERNS, GRIDS AND VARIATION**

**SAMPLES IN GREATER DETAIL**

Sonic Pi also comes with around 200 samples, or pre-recorded sound files that you can use in your music compositions and performances. To use them, simply type the word sample followed by a space, and a list of those samples will appear in a dropdown menu.

The word sample, like play, is a function telling Sonic Pi to make a sound, but using built in pre-recorded sound files, instead of a synth.

Run the following code, and then replace the sample with other sounds.

sample :bd\_fat

Now let’s combine two drum sounds in a loop.

use\_bpm 70

loop do

r = rrand(0.8, 2.0)

sample :bd\_haus, rate: r

sleep 0.5

sample :sn\_zome, rpitch: r

sleep 0.5

end

This example introduces a little bit of variation with the rrand function, which selects a random fractional number (unlike rrand\_i, which generates whole numbers) between 0.8 and 2.0. This number is then used for the playback speed of the bass drum sound, and pitch of the snare drum sound. Changing the playback rate of a sound file or sample alters its perceived pitch; higher playback rate = higher pitch, and lower rate = lower pitch.

Before we move on to the next big topic (live\_loop in detail), let’s add a little more interest to that drum loop, using effects, referred to in Sonic Pi as FX. There are many fun FX to experiment with, and this example introduces two of them. When using FX in Sonic Pi it is important to use a do/end block (just as we do when using a loop) for each effect.

use\_bpm 70

with\_fx :wobble, phase: 8, wave: 3, res: 0.2, mix: 0.7 do

with\_fx :reverb, room: 0.6, mix: 0.6 do

loop do

r = rrand(0.8, 2)

sample :bd\_haus, rate: r

sleep 0.5

sample :sn\_zome, rpitch: r

sleep 0.5

end

end

end

Take care to ensure all the commas and colons are in the correct place. Notice that both FX blocks have their own do/end open and close. Try changing the wave: value to either 0, 1 or 2. Try changing the phase: value to a number smaller than 8. Try changing the room, mix and res values to a number between 0 and 1.0.

Try these values:

with\_fx :wobble, phase: 0.25, wave: 0, res: 0.95, mix: 0.9

NEXT TOPICS ?

**FX IN GREATER DETAIL**

**CUE, SYNC, STOP; SEQUENCING COMPLEX PATTERNS**