**Basic definitions in the framework of SuperCollider**

This is a humble glossary of terms that I believe are useful when working with SuperCollider. It is, of course, in development and I should keep finding and adding terms that I haven’t included yet. It is also not ordered alphabetically, but sort of thematically, not even really (should rethink that as well).

///////////////////////////////////////////////////////////////////////////////////////////////////////////////

**OOP (Object-oriented programming):** Programming paradigm (a specific style or way of programming, being a programming language a set of rules allowing the communication with the computer) based on the concept of “object”. All entities in the language are objects.

**Object:** An object is something containing data about its state and the set of operations that can be performed with it. Objects are always instances of a class, which describes the structure of the object and its operations.

**Class:** An object class contains the description of the object’s data and operations. It also describes how to create an object, which is an instance (a version, a realization) of that class (blueprints). Their name always starts with a capital letter. Classes are themselves objects (see the class tree, with all hierarchy of classes branching from the main class, “Object”).

**Message:** A message is a request for an object (the receiver) to perform one of its operations. The way in which this operation is performed is defined by the object’s class. The implementation of the objects is hidden from the client, being only able to change the object’s state by sending messages (ctrl + i, opens implementation). A message consists of a message selector (name of the operation) and, in some cases, a list of arguments (input values) which give additional information about the operation. A message always returns a result (by default, the receiver itself).

**Syntax:** For example, messages, can be expressed in many ways: as binary operators (! @ % & \* - + = | < > ? /, or an identifier followed by a colon, such as *10 rrand: 100*), alongside functional notation messages, such as *rrand(10, 100),* and receiver notation messages, such as *10.rrand(100).* All these different ways of expressing the same message are called syntaxes (it is best to be consistent with one).

**Polymorphism:** Objects of different classes can respond to the same message in different ways according to their class. This ability for different objects to react differently to the same message is called polymorphism. For example, all objects can understand the message “value”: many will just return themselves, and others such as functions or streams will evaluate themselves and return the result of that evaluation.

**Method:** A method is a description of the operations necessary to implement a message for a particular class: they tell how to implement messages sent to the instances of a class. In a class, there is a method definition for each message to which the instances of that class can respond to: when an object is sent a message, the method whose name matches the message selector in the receiver’s class is executed. Methods may inquire about some property of the receiver, ask the receiver to change its internal state, or ask the receiver to return some computed value. There are two main types of methods: class methods are sent to class objects to create instance objects of that class; instance methods are sent to an instance object to modify it or inquire information about it.

**Language architecture:** SC application is three programs: a text editor, also called IDE, Integrated Development Environment; the language, called *sclang,* also known as the client app; and the server, which carries out the synthesis and calculation of audio, and which can be *scsynth* by default, *supernova* (used for parallel processing, so that your own computer is not the limit of the audio engine)*,* or any other audio engine. *Scsynth* is, nonetheless, one of the most finely tuned and efficient synthesis engines out there (it is a command line program, meaning that it runs without any graphical user interfaces). The client and server applications normally run in the same machine, but they can also run in different machines.

**Client:** The client is in charge of sending OSC messages to the server (see “communication protocols”). The client is both the *paper sheet* and the *mailer* to send these messages to the server. The client also features the **interpreter**, which allow us to write the code in the language of SC and execute it. As OSC messages are low-level language (difficult to write and understand by a human), the object-oriented language (higher level) of the SC client allows much more expressive power than OSC messages (although it is possible to write raw OSC messages in the syntax). The interpreter translates the SC language into OSC messages for the server.

**Server:** The server doesn’t know anything about the SC language, doesn’t know what’s an object or what the code means: the server only understands OSC messages and UNIX commands (a command language, used in Linux or Mac, but not in Windows, for example; the way of talking to the console, for instance, in Unix systems). The server can receive OSC messages from other apps different from SC client (for example Max/MSP), but keep in mind that *sclang* is specifically designed to obtain all the expressive power of the SC language and to fit the server’s needs. The default server is the local server (assigned to global variable “s”).

**Communication protocols:** The network is the exchange of messages between the client and the server. Both programs communicate by a protocol called OSC (Open Sound Control), over either UDP (User Datagram Protocol) or TCP (Transmission Control Protocol), which are network protocols used on the internet. The messaging process introduces a small amount of latency (delay between one side sending a message and the other side receiving it and acting in consequence).

**Server abstraction objects:** Due to the client-server architecture in SC, it is crucial to remember that there is a distinction between the actual things like nodes, busses, buffers or servers, and the objects that represent them in the language app (in the form of instances of the classes Node, Bus, Buffer, Server, etc.). For example, a Buffer object is a client-side abstraction of a server-side buffer: it is a representation of the buffer in the server containing its common tasks, OSC messages, capabilities, etc. The abstraction objects are used to control from the language the actual objects in the server.

**Extensions / Quarks:** Quarks are packages of SuperCollider code containing classes, extension methods, documentation and server UGen plugins (see “UGen”). There is a Quarks class to manage downloading the packages and installing / uninstalling them. Extensions are additions to the class library, documentation and server UGen plugins. They are placed in the specific directories of SC in your machine.

**Signal:** (do not confused with the object Signal). An audio signal is a representation of sound, typically using either a changing level of electrical voltage (analogue) or a series of binary numbers (digital). Audio signals can be synthesized, or they can be captured using a transducer (a microphone, a phonograph cartridge, a tape head, etc.). Loudspeakers or headphones convert electrical audio signal back into sound (differences in the air pressure). Audio signals can be periodic or aperiodic: they repeat the exact same pattern (cycle) or they present chaotic patterns. The most basic periodic waveforms are sinusoidal, triangle, square (or pulse) and sawtooth oscillators. Aperiodic signals generate noise (irregular oscillation pattern). In SuperCollider, noise generators produce random frequency values at sample rate.

**UGen:** Unit Generators represent calculations with signals, that is to say, they are classes that create or process signals. All unit generator classes branch from the UGen class: the many subclasses of UGen are the client-side representation of unit generators, and are used to specify their parameters when constructing synth definitions (see “SynthDef”). All UGens respond to one or more of the following class methods: .ar (for audio rate signals), .kr (for control rate signals) or .ir (for initial rate signals). Then, they return a new instance of UGen that calculates the signal at audio or control rate, or only once at initialization. Some UGens respond to the .new method instead (like Rand). If any argument (input value) is an array (see “Array”), they return an array of UGens (see “Multichannel expansion”). The combination of rates between arguments and UGen might not be allowed and this will throw and error (for example, when using a UGen to modulate the argument of another UGen, see “Modulation”). UGens only work within a UGen function (.play or within a SynthDef, etc.). Here one can find the most relevant categories of UGens and the most prominent UGens among them: <https://doc.sccode.org/Guides/Tour_of_UGens.html>

**The arguments mul and add:** They appear in almost all UGens responding to .ar or .kr methods. Mul is a constant or signal by which the output of the UGen is multiplied, so it often corresponds to scaling the amplitude of the UGen signal. Add is a constant or signal added to the output of the UGen, so it often corresponds to adding a constant or DC offset to the signal.

**Audio rate (.ar):** UGens to which an .ar message is sent run at the audio rate, which by default is 44100 samples per second. This means that signals at audio rate are usually meant to be heard.

**Control rate (.kr):** UGens to which a .kr message is sent run at the control rate. By default, these UGens will generate one sample value for every 64 sample values made by an audio rate UGen (this is called block size, the number of samples in one control period). Control rate UGens are thus less computationally expensive than audio rate ones. This means that signals at control rate are not meant to be heard, but to be used as modulators.

**Initial rate (.ir):** UGens to which an .ir message is sent only calculate their output at creation (initialization). These signals cannot be modulated.

**Other rates:** Trigger rate signals are controlled by a trigger, which normally happens when a nonpositive to positive transition occurs at the input of the trigger Ugen. Demand rate UGens are demanded values when listed as inputs of the Demand or Duty UGen. In the case of Demand, the values are demanded when receiving a trigger in the trigger input. In the case of Duty, values are demanded according to a stream of duration values.

**Sample rate:** By default, 44100 Hz: it determines the number of snapshots taken to recreate the original sound wave (its “resolution”). Sample rate affects the frequency range of resolution (that is to say, the resolution of the “x” axis) of digital audio signals.

**Bit depth:** Usually set at 24 or 32 nowadays (standard for CD is 16, and up to 24 for DVD), although it can reach 64 in most DAWs. Bit depth affects the dynamic range of resolution (that is to say, the resolution of the “y” axis) of digital audio signals.

**Node:** A node is an addressable point running by the synth engine (the server). Nodes are ordered in a tree of nodes (the node tree) that defines the order of execution. Nodes can be of two types, Synths and Groups. All nodes have an integer ID.

**Group:** A group is a collection of nodes. New nodes can be added to the head or the tail of a group, and the nodes within a group may be controlled together. The nodes in a group can be synths or other groups. There is a root group with ID = 0, which is created by default when the server is booted up, and a default group with ID = 1, also create at booting time. In the root group we usually find all the recording, monitoring and primitive functions. In the default group, all new nodes are created if not indicated otherwise.

**Synth:** A synth is a collection of unit generators that run together. They read input and write output to global audio and control busses. A synth represents a single sound producing unit, and what it does is defined in a SynthDef (see “SynthDef”), which specifies what UGens are used and how they are put together, and what inputs and outputs the synth will have.

**SynthDef:** Or synth definitions: synths are created from SynthDefs, as SynthDefs are create in the language application, and then compiled and loaded into the server. They are referred to by a name (see “Symbol”). A SynthDef specifies what UGens are used and how they are put together (this is called the UGen graph function), and what inputs and outputs the synth will have. They are kind of fixed patterns, upon which synths are based, although they can provide a surprising amount of variation (all values can be modified except for those changing the structure of the synth, such as array sizes, number of inputs or outputs, etc.). The client-side representation of a synth, the class Synth, is used to obtain the sound from a SynthDef, representing a single instance of it, referring to its through its name (symbol), specifying its arguments, and indicating its target and action (if working with interconnected synths: see “Order of execution”).

**Busses:** Busses have the function of routing signals from one place to another (from one synth to another, for instance, or from / to hardware). They can be audio rate or control rate busses, routing audio or control rates signals respectively. The number of control and audio busses available, and the number of input and output channels, is set at the time the server app is booted (see Server Options): by default, there are 2 input and 2 output channels, 16384 internal control rate busses, and 1024 audio rate busses, including the input and output channels (see “Audio rate bus”). There is no such thing as “multichannel bus”: a multichannel signal simply reserves a series of adjacent bus indices in the global array of busses.

**Control rate bus:** Control rate busses have an index number starting from 0 (as well as audio busses). Synths can send control signals to each other via a single global array of control busses.

**Audio rate bus:** The server app has a certain number of output and input channels, which will be reserved for the first audio busses (first outputs, then inputs). By default, channels 0, 1 are output channels for left and right out, and channels 2, 3 are input channels for left and right in. These are used to send audio to the speakers or receiving audio from, let’s say, a microphone. But there are more audio busses which are “private” and which are used to send audio and control signals between various synths (so these signals will not produce sound). Those are usually used when a signal requires further processing before sending it out through the speakers (for example, to pass through a filter, reverb, delay, etc.). When using hardware, the number of input and output channels in SC should match the hardware’s.

**Buffers:** Buffers represent buffers in the server, which are ordered arrays of floats and serve to store data in the server. The most common use is to hold soundfiles in the memory (either to read or write them), but any sort of data that can be represented as floats can be stored in a buffer. Like busses, the number of buffers is set before the server is booted (by default, the number of global sample buffers available is 1024. The real time memory allocated to the server for synths and UGens such as delays, also set before booting the server, is separated from the memory used for buffers). Before buffers can be used, we need to allocate memory to them (the “container” needs to be created), which is an asynchronous step (at boot time, all buffers have a size of 0). Like busses, buffers are identified with numbers starting from 0: when calling .free on a buffer object, it will release the buffer’s memory on the server, and free the number for future reallocation.

**Order of execution:** The order of execution is a crucial issue when creating synths which interact with each other (when using In.ar in a synth to read the output of another; this also applied to groups). In SC, the nodes are organized from top to bottom within the groups of the node tree. On top is what is heard (a sound), on the bottom what processes it (a filter, for instance): the synth that does the filtering must be later in the order of execution than the synth which is its input. When creating an instance of a SynthDef with the Synth class, we can define a target, with will represent the group in which the synth is created (“s” for default group in the *scsynth,* and an integer for any other group with that integer as ID; remember that 0 = root group and 1 = default group), and we can add an action, which will define the order of execution within the target, being the possibilities (as symbols): \addToHead (default, head of the group of the target), \addToTail (tail of the target group), \addAfter (immediately after the target’s node), \addBefore (immediately before the target’s node), or \addReplace (replacing the target’s node). For adding to head or tail, the target must be a group (or the server). The easiest way to approach this is to test and check the node tree (worst thing that can happen if messing up with the order of execution is that something does not get to be heard). There are also methods to be applied to the Synth class for each addAction (for instance, Synth.tail or Synth.after).

**Data types / literals:** Data types are different ways of representing and interpreting data (information). However, in SuperCollider, all data is represented by objects, therefore SC is not based on data types as such, but on objects. There are, nonetheless, literals, objects whose value is represented directly in the code rather than sending a message to an object, that is to say, they have a direct syntactic representation. Needless to say, each literal can be also represented by its corresponding class. These are the literals that can be represented in SC:

* **Number:** The class Number represents a mathematical quantity. Literal numbers can be integers or floats. Mathematical operations with numbers can be unary (applied to single numbers, usually represented by methods) or binary (happening between two or more numbers, most commonly represented by binary operators, but also some possible with methods). Some mathematical operations can be written with a different grammar: *16 % 7* OR *16.mod(7)* OR *16 mod: 7* OR *mod(16, 7).* The behavior of numbers around the representation limits (of integers or floats) depends a bit on each computer’s capacity.
* **Integer:** An integer is any series of digits, optionally preceded by a minus sign: -13. Integers are 32-bit, which sets its representation limit at 2147483647 (or -2147483648), which folds into negative or positive range when surpassed, respectively. Integers can be also represented with the class Integer.
* **Float (floating-point number):** A floating-point number consists of one or more decimal digits followed by a decimal point and one or more decimal digits (2.78). It can also be preceded by a minus sign. In SC, floating-point numbers are 64-bit (except within a FloatArray, in which they are 32-bit). This sets their representation limit much further than that of an Integer. However, because the infinite continuity of floats is still discrete in a computer, some float numbers might be represented by a very close approximation instead. Floats can also be represented with the class Float. Float numbers support the exponential notation (1.2e4), the *pi* keyword (to refer to the pi float), and the keyword *inf* (or *-inf*), which represents infinity and is also treated as an instance of Float.
* **Other number literals:** Other number representations in SC include radices other than base 10 (for example, for hexadecimal notation, *16rF*, or for binary numbers, *2r01101011*: so, radix or base specified in base 10, followed by “r” and following by the value written in that radix. Supported characters include 0-9 and A-Z or a-z. The radix is written in base 10); there are also scale degree notation (which ends up being not so useful).
* **Characters:** Single characters are represented with a dollar sign before a character. See SC help for examples of characters (barely used, anyway).
* **String:** A string is a sequence of characters between double quotes: *“string”.* Strings can be applied methods such as .size or .scramble. Strings can contain spaces. Two strings containing the same characters do not represent the same object, that is to say, they are equal (==), but not identical (===).
* **Symbol:** A symbol is also a sequence of characters (can also contain spaces, but only if within single quotes). They can be preceded by a backslash or placed between single quotes: *\symbol* OR *‘symbol’.* Two symbols containing the same characters do represent the same object, that is to say, they are equal (==) and identical (===), and this is why they are used as identifiers or tags (for example, for SynthDefs’ names). Strings can be converted to symbols using the .asSymbol method.
* **Identifiers:** Names of methods and variables (see “Variables”) are identifiers. They always begin with a lower-case letter and can contain any alphanumeric characters and underscores.
* **Special values:** true, false and nil (empty) are special values in SC, expressed with keywords.

Class names and function definitions (enclosed by curly brackets { }) are also literals.

**Variables:** Variables are data containers. They represent the data that is assigned to them. Variables can be global or non-declared. The interpreter in SC offers variables a-z as global variables (being the server assigned to “s” by default). These are easy to overwrite accidentally. Variables can also be declared (local variables) within functions, preceded by the keyword *var,* and named with any combination of characters and underscores (beginning with a lower-case letter). These variables will be only considered within the block of code they are declared. Another place where global variables can be stored is the currentEnvironment (see “Environment”) as pairs or associations of key -> value, which can be called with the symbol ~ (tilde). This way, global variables can be created with a more descriptive name, such as ~variable1.

**Functions:** A function is an expression which defines operations that are performed when it is sent the .value message and a list of arguments in the order they are declared within the function. The arguments of a function are the input values that make the function flexible (and thus reusable). Functions are expressed between curly brackets. All operations or instructions (separated by semicolons) within a function are executed in order at once (eager evaluation), so that the result we obtain is that expressed by the last line of the function. Functions actually have a compilation limit. If they contain more than 256 selectors (class and method names, or functions within the function delimited by curly brackets), they will print an error, but this may refer to a code composed by several thousand lines, which would obviously work better if made “modular”.

**Collection (array, set, etc.):** Collection is an abstract class, meaning that no direct instances of Collection are made. Instead, we create instances of the many types of collections, such as Array, Set, or Dictionary. Array is the most common type of collection, so common that its class does not need to be expressed (the square brackets express an Array by default [ ]). An array is a collection in which the elements it contains can be indexed (meaning that their order matter). The index of each element is an integer starting by 0. Arrays can contain any kind of object, which can be repeated. Sets, however, cannot contain two or more objects that are equal. In addition, elements in a Set are unordered (they cannot be addressed through indices).

**Dictionary:** A dictionary is a type of collection containing associations between keys (tags) and values (key -> value). They are unordered, but can rely on the equality of the keys. Dictionaries using symbols as keys rely on identity and are called IdentityDictionary. The key can point to any kind of value.

**Environment:** An Environment is an IdentityDictionary in which functions can be defined and/or evaluated. For example, global variables are contained within an environment (called currentEnvironment).

**Event:** This class is a subclass of Dictionary as well and it is especially linked to the Pbind class. An Event describes actions to be taken in response to a play message, and in which the parameters of those actions are specified by key-value pairs.

**MIDI:** Stands for Musical Instrument Digital Interface and it is a communication protocol originally designed for digital music synthesizers. This standard allows the communication between devices, holding information about pitch, velocity (amplitude), and duration. SuperCollider supports MIDI, meaning that all MIDI devices accessible to your operating system are also accessible to SuperCollider. More information on how to use MIDI on SuperCollider can be found here: <https://doc.sccode.org/Guides/UsingMIDI.html>

**Multichannel expansion:** In SuperCollider, multiple channels of audio (or control) are represented as Arrays. Each channel of this array will go out a different speaker, therefore, you are limited by the number of speakers you have and the number of channels your audio interface or soundcard supports. In SC, all UGens have only a single output to facilitate the array operations manipulating multi-channel structures. Now, when an array is given as an input to a UGen, it causes an array of multiple copies of that UGen holding the different values from the input array. This phenomenon is called multichannel expansion: “when a UGen is called with an array of inputs, it returns an array of instances”. If the generated array is the input of another UGen, another array is created (a sawtooth wave receiving an array of two frequencies will create two sawtooth waves, and this array of two sawtooth waves, if placed into a lowpass filter, will create an array of two lowpass filters. Both filters will share a single instance of any other UGen that is not expanded). If a UGen receives more than one array as input, the longest array will determine the number of instances of the UGen that are created. The object Mix reduces multichannel arrays to a single channel: Mix([a, b, c]) is equal to [a, b, c].sum or a + b + c (although using Mix reduces the performing time of the addition). Array do not cause Mix to create copies of itself. If Mix receives an array of arrays, it will only reduce one layer of arrays (for example, if receiving an array containing an array of sine waves and an array of sawtooth waves, it will mix the sine waves on one channel and the sawtooth on another channel). That is the limit as Mix cannot be used on arrays of arrays of arrays.

**Envelope:** Envelopes describe events over time. An envelope is usually defined by a series of segments or lines going from one point to another in a specific amount of time. The shape of this line can vary. The way the envelope moves from one point (value) to another is also called interpolation. In SC, two main classes are used to create envelopes: Env is used to create an envelope specification (to specify its values of levels, times and curves, for instance), and EnvGen is the UGen that creates the envelope specified in Env.

**Trigger:** A trigger is anything that causes a reaction (for example, makes something function at a specific moment). A trigger in SC is produced when zero is crossed in the input of the trigger (for example, through the use of an Impulse or a Pulse wave, etc.). If a trigger is applied, for instance, to an envelope (to its “gate” argument), the envelope will be rebooted at the frequency of the trigger’s input.

**Gate:** A gate allows an input signal value to pass when the gate is positive, and otherwise hold its last value (when zero or negative). If a gate is used, for instance, in an envelope (the envelope type needs to feature a sustain segment), a gate equal to 1 will trigger the attack segment and maintain the envelope’s sustain level, and a gate equal to 0 will trigger the decay segment and “turn off” the envelope. Notice that the envelope would be still running in the server. It does not go away until we kill the synth.

**Stream:** A stream is the “lazy” version of an envelope. It represents a sequence of values that are obtained when it is asked to give them. Its values are obtained by evaluating the message .next and the stream can be reset by using the message .reset. Routines in SC work as streams (see “Routine”).

**Routine:** A routine is the “lazy” version of a function: it allows you to perform each instruction one by one, being able to suspend the evaluation in the middle and resume it again from that point, or reset it. A routine holds a series of instructions, and evaluates each instruction or group of instructions when it is called with the message .next and until it finds a .yield message in any of the instructions, being the evaluation suspended then. It is resumed using the message .next again, and it can be reset by using the .reset message. When Routine receives the message .play, it is scheduled (controlled) by a Clock (see “Clock”), which will start or resume the routine at the scheduled time. The value yielded will be used as the time difference for rescheduling the routine (and therefore, when controlling routines with a clock, the message .yield can only be applied to numbers). When using a clock, .yield can be replaced by the method .wait, which fulfills the same function.

**Task:** A task is a pauseable process. A Routine can suspend the execution of its instructions, but once it is stopped, it cannot be resumed. Whenever a Routine is stopped, it can only be run again from the beginning if reset, so we cannot replay a Routine from a point it was left at if it is not running anymore. A Task fulfills the same function as a Routine but can indeed be paused and resumed at any time.

**Clock:** A clock keeps track of time and allows processes such as streams to be scheduled for some time in the future, mainly by applying the message .play to them. There are three subclasses of Clock in SuperCollider: SystemClock only works in seconds, while TempoClock is based on beats per second and therefore allows you to change the time unit in use (the tempo), then AppClock is used for lower hierarchy tasks such as graphical content (it is not accurate enough for musical scheduling and requires less processing power).

**Delay:** A delay is the act of postponing, that is to say, causing something to happen later (if applied to something that sounds, produces a simple echo with no feedback). In SuperCollider there are different types of delay, mainly differentiated by their interpolation: DelayN uses no interpolation, DelayL uses linear interpolation, and DelayC, cubic interpolation. A delay line with feedback can be created in SuperCollider by using a **Comb filter** (also available as CombN, CombL and CombC) or **Allpass filter** (also available in the same three interpolations), mostly used as resonators. Take into account that the interpolation difference in a delay can only be heard if its delay time is, for instance, modulated. A delay with a very small delay time (below 1/20) will start functioning as a resonator (as we are no longer able to tell the echoes apart and they will create their own spectrum).

**Sample / sampler:** A sample normally refers to a portion of a sound recording that is used and manipulated to create music (sampling). A sampler is an electronic instrument that records and plays back samples (portions of those recordings). The other meaning of sample refers to the momentary value of an analogue signal that is taken several times a second to convert it to digital.

**Filter:** Filter is used to emphasize or eliminate some frequencies from a signal, that is to say, they affect the frequency spectrum of a sound. The main filters used in SC include the high pass filter (HPF), low pass filter (LPF), band pass filter (BPF), band reject filter (BRF), and resonating filters (such as RHPF, RLPF, Resonz, Ringz, etc.). The basic filters LP, HP, BP and BR are 2nd order Butterworth filters: Butterworth filters are designed to have a frequency response that is as flat as possible, and its order refers to the slope of the filter (how fast the frequencies are cut off).

**Resonator / Reverberator:** Resonators can boost certain frequencies of the spectrum of a signal. Delay lines such as Allpass filters and comb filters are used as resonators (a Schroeder filter combines both of them). Reverberation, the prolongation of a sound after it is produced, can be caused by resonance: when several reflections of the sound are built up and then decay.

**Limiter / Normalization:** A limiter limits an input signal’s amplitude to a given level. Amplitude peaks going beyond the given level are normalized, that is to say, proportionally adapted to the given range of amplitudes.

**Pitch shifting:** Pitch-shifting allows to change the pitch of a sound without affecting its speed of reproduction (its rate). Normally, raising a sound an octave higher means doubling its rate.

**Time stretching:** As counterpart of pitch shifting, time stretching allows to change the rate or speed of a sound without altering its pitch.

**Phasor:** A phasor is a linear ramp between start and end values coming back to a reset value (therefore describing a sawtooth waveform). A phasor is what makes oscillate an oscillator according to its frequency.

**Panning:** Panning consists of positioning sounds in the left to right spectrum of the stereo image. The easiest way to pan in SuperCollider (and therefore, to obtain a stereo signal) is to use Pan2, a two-channel equal power pan, in which the position is measured from -1 (left) to 1 (right).

**Pattern:** Patterns are one of the most useful tools in SuperCollider. Here is a guide on how to use them: <https://doc.sccode.org/Tutorials/A-Practical-Guide/PG_01_Introduction.html>. Pattern are higher-level representations of certain calculations, that is to say, they perform very specific small tasks. It might be difficult to find the specific pattern needed, but when we do, it eases the process quite a bit. Patterns are the blueprint of sequences (streams) of information, often of numbers, although they can hold any kind of object. Values from a pattern can be easily obtained when converting the pattern into a stream (by applying the method .asStream) and asking subsequently with the method .next (for example, in the context of a Routine).

**Modulation:** When a periodic or aperiodic signal is applied to a certain parameter of another signal, a modulation is provoked, as the value of that parameter will periodically (or aperiodically) change. The signal that is being modulated is the “carrier”, whereas the signal producing the modulation is the “modulating” signal. Modulations can occur at a pace at which we can hear the different values. In these cases, the modulating signal is usually and LFO (low frequency oscillator). Modulations whose frequency goes beyond the 20 Hz limit produce new frequency spectrums, being the most usual the AM (amplitude modulation) and Ring modulation, FM (frequency modulation), and PM (phase modulation), although all of these might also occur in the low frequency domain.

**AM / Ring modulation:** It is produced when the modulating signal varies the amplitude of the carrier signal. When the frequency of the modulating signal goes beyond 20 Hz, new frequencies are produced, namely the sum and difference of the frequencies of the carrier and modulating signals. The difference between AM and RM is related to the modulation depth: in AM the amplitude remains between 0 and 1 (unipolar signal), while in RM, the amplitude remains between -1 and 1 (bipolar signal). In Ring modulation, therefore, the frequency of the carrier disappears. Resulting spectrums get more complex according to the kinds of signals involved in the modulation (when using anything different from sine waves only).

**FM:** It is produced when adding an audio signal to the frequency of the carrier signal. A new spectrum is created due to the produced “side bands”, which are new frequencies appearing in pairs on both sides of the carrier’s frequency. The position of the bands depends on the modulating signal’s frequency, affecting how fast the oscillation between frequencies is produced; and the number of audible bands is proportional to the modulating signal’s mul (amplitude), affecting the range of frequencies in between the carrier’s frequency oscillates. The carrier’s frequency determines the point in the spectrum around which this cluster of side band activity occurs (mean value).

**PM:** In phase modulation, a modulating signal modifies the phase of a carrier signal. It works in a very similar way as frequency modulation, as frequency and phase are interrelated (in order to change the phase of a signal, its frequency needs to be modified shortly, and vice-versa: *freq = Hz = cycles / second. Phase = freq \* time = (cycles / second) \* second = number of cycles. Two sine waves differing in frequency by 0.5 Hz will get progressively out of phase with each other by 0.5 cycles every second*). If bearing in mind phase as the circle described by a sine wave, what is modulated is the speed of rotation around the circle.

**Additive synthesis:** In additive synthesis, timbre is created by adding sine waves together (this according to Fourier theory) with different pitches and amplitudes.

**Subtractive synthesis:** In subtractive synthesis, filters are mainly used to sculpt timbre. Conventionally, the source of a rich sound wave to be filtered is a saw-tooth or square wave, but, technically, filtering noise could be called subtractive synthesis.

**Granular synthesis:** Granular synthesis operates in the microsound time scale and is based on the same principle as sampling, although, conventionally, the pieces that the original sample is split into are around 1-100 milliseconds long. Those pieces of sound are called grains. These grains can be overlapped (layered on top of each other), they can be separated by a regular interval of time or randomize this interval (synchronic / asynchronic), they can receive different values for their duration, amplitude, rate, etc., and they can be taken from a small window of the original sample or randomized from any part of it. Important names in the development of this technique are Iannis Xenakis, Curtis Roads or Barry Truax, among others.

**Wavetable synthesis:** It involves the use of an ordered collection of values defining any shape of the sound wave. The collection of values is the wavetable, and its size is usually a power of two (512, 1024, 2048, etc.) for computer efficiency reasons. The wavetable represents a single cycle of a periodic wave that a wavetable oscillator then reads at a specific frequency.

**Stochastic synthesis:** Also developed by Xenakis, this synthesis technique is applied at the level of the digital samples, applying probability distributions to the creation of digital sound particles (samples). In stochastic synthesis, the sound waveform is conceived as a polygonal line made out of breakpoints linked by linear interpolation. For each point, a random value (normally based on a random walk, so that there is a range or limit for the variation or deviation of a random value with respect to the next value) is going to be produced for both the time (horizontal spacing between one breakpoint and the next) and amplitude (vertical domain). The number of breakpoints per waveform is a parameter that can be chosen: the greater this number, the lower the fundamental frequency of the resultant sound because additional breakpoints make the waveform longer. Therefore, the two main parameters to control (which will affect the sound result the most) in stochastic synthesis are the number of breakpoints and the step or deviation value for the random walk. (Main source: <https://www.iannis-xenakis.org/en/stochastic-synthesis/>).

**Zero crossing:** A zero crossing is a point where the waveform crosses the zero level axis (in “y”). When playing around with waveshapes performing editing operations such as cutting, pasting, dragging, and the like, it is advisable to use a zero crossing detector so that the material is inserted at a zero crossing. Otherwise, clicks and other artifacts can appear. For instance, envelopes can be used to create zero crossings.

**Flanger:** A flanger is an effect created by combining two copies of the same audio signal. One of the copies is delayed by a very short delay time (usually under 20 milliseconds), putting it out of phase with the other copy. This slight phase difference causes interference in the signal: the frequency spectrum is altered, creating peaks and notches that resemble the teeth of a comb. The delay time is usually changed over time (modulated), creating shifts up and down the spectrum, and therefore the familiar airplane sound. (A static flanger is just a comb filter, right?).

**Chorus:** The chorus effect is also based on one or more signals being slightly delayed. It is said that the difference with respect to the flanger is that the delay time is longer than in a flanger (15-35 milliseconds), and that the delayed signal(s) modulate their pitch with respect to the input signal. As the input signal and delayed signal(s) aren’t so similar, there is less interference in the waveform, which will be less “spiky”.

**Phaser:** It is an effect similar to flanger or chorus, but instead of delaying the signal, all pass filters are used to alter its phase without changing its timing.