Writing code in and around CoppeliaSim

CoppeliaSim is a highly customizable simulator: every aspect of a simulation can be customized. Moreover, the simulator itself can be customized and tailored so as to behave exactly as desired. This is allowed through an elaborate Application Programming Interface (API). 7+ different programming or coding approaches are supported, each having particular advantages (and obviously also disadvantages) over the others, but all seven are mutually compatible (i.e. can be used at the same time, or even hand-in-hand). The control code of a model, scene, or the simulator itself can be located inside:

- an embedded script (i.e. customizing a simulation (i.e. a scene or models) via scripting): this method, which consists in writing Lua or Python scripts, is very easy and flexible, with guaranteed compatibility with every other default CoppeliaSim installations (as long as customized API functions are not used, or are used with distributed plugins). This method allows customizing a particular simulation, a simulation scene, and to a certain extent the simulator itself. This is the easiest and most used programming approach.
- an add-on or the sandbox: this method, which consists in writing Lua or Python scripts, allows to quickly customize the simulator itself. Add-ons (or the sandbox) can start automatically and run in the background, or they can be called as functions (e.g. convenient when writing importers/exporters). Add-ons should not be specific to a certain simulation or model, they should rather offer a more generic, simulator-bound functionality.
- a plugin (i.e. customizing the simulator and/or a simulation via a plugin): this method basically consists in writing a plugin for CoppeliaSim. Oftentimes, plugins are only used to provide a simulation with customized API commands, and so are used in conjunction with the first methods. Other times, plugins are used to provide CoppeliaSim with a special functionality requiring either fast calculation capability (scripts are most of the time slower than compiled languages), a specific interface to a hardware device (e.g. a real robot), or a particular communication interface.
- the client application (i.e. customizing the simulator and/or a simulation via a custom client application): this basically consists in writing a custom main application for the CoppeliaSim library. This allows for instance to launch/shutdown CoppeliaSim several times in a row, or in a particular manner.
- a remote API client (i.e. customizing the simulator and/or a simulation via a remote API client application): this method allows an external application (e.g. located on a robot, another machine, etc.) to connect to CoppeliaSim in a very easy way, using remote API commands.
- a ROS node (i.e. customizing the simulator and/or a simulation via a ROS node): this method allows an external application (e.g. located on a robot, another machine, etc.) to connect to CoppeliaSim via ROS, the Robot Operating System.
- a node talking TCP/IP, ZeroMQ, etc.: this method allows an external application (e.g. located on a robot, another machine, etc.) to connect to CoppeliaSim via various communication means.

Above 7 methods are also discussed in the external controller tutorial. Following table describes in detail the respective advantages and disadvantages of each method:

	Embedded script	Add-on / sandbox script	Plugin	Client application	Remote API client	ROS / ROS2 node	ZeroMQ node
Control entity is external (i.e. can be located on a robot, different machine, etc.)	No	No	No	No	Yes	Yes	Yes
Supported programming language	Lua, Python	Lua, Python	C/C++	C/C++, Python	C/C++, Python, Java, JavaScript, Matlab, Octave	Any [1]	Any
Code execution speed	Relatively fast [2]	Relatively fast [2]	Fast	Fast	Depends on programming language	Depends on programming language	Depends on programming language
Communication lag	None [3]	None [3]	None	None	Yes	Yes	Yes
Communication channel	Python: ZeroMQ [3]	Python: ZeroMQ [3]	None	None	ZeroMQ or WebSockets	ROS / ROS2	ZeroMQ
Control entity can be fully contained in a scene or model, and is highly portable	Yes	No	No	No	No	No	No
Stepped operation [4]	Yes, inherent	Yes, inherent	Yes, inherent	Yes, inherent	Yes	Yes	Yes
Non-stepped operation [4]	Yes, via threads	Yes, via threads	No (threads available, but API access forbidden)	No (threads available, but API access forbidden)	Yes	Yes	Yes

[1] Depends on ROS / ROS2 bindings
[2] Depends on the programming language, but the execution of API functions is very fast
[3] Lua scripts are executed in CoppeliaSim's main thread, Python scripts are executed in separate processes
[4] *Stepped* as in *synchronized* with each simulation step

[Possible control methods in and around CoppeliaSim]