Quantum Circuits Made Out Of Pulses On QuTiP

Jorge Fernandez Pendas

March 19, 2020

I am working with the Qutip capabilities to define circuits in order to have a meeting with Pontus later today.

There are essentially two ways to do this. One involves the use of products of unitary matrices and the other one involves proper simulation of pulses. The first approach is imported as qutip.qip.QubitCircuit and the other approach is imported as qutip.qip.device.Processor. Although we will obviously choose to work with the processor, let me write down a short list of the most evident features of QubitCircuit that make it interesting:

- The quantum gates are saved as an instance of class Gate, with info such as gate name or target and control qubits.
- We can access the matrix representation through method propagators and perform the whole evolution from that point.
- The pre-defined gates are: RX, RY, RZ, SQRTNOT, SNOT (Hadamard), PHASEGATE, CRX, CRY, CRZ, CPHASE, CNOT, CSIGN, BERKELEY, SWAPalpha, SWAP, iSWAP, SQRTSWAP, SQRTSWAP, SQRTISWAP, FREDKIN, TOFOLI and GLOBALPHASE.
- Gates can be decomposed into a set of elementary gates.
- One can define new gates.

Let's now move on to discuss the other approach: Processor. Let us use for that the example shown in the documentation:

```
from qutip import *
import numpy as np
from qutip.qip.device import Processor
proc = Processor(2)
```

```
proc.add_control(sigmaz(), cyclic_permutation=True)  # sigmaz for all qubits
proc.pulses[0].coeffs = np.array([[1.0, 1.5, 2.0], [1.8, 1.3, 0.8]])
proc.pulses[0].tlist = np.array([0.1, 0.2, 0.4, 0.5])
```

proc is defined to be a processor for 2 qubits. After that, the first control pulse is defined to be a σ_{z} on the first qubit (otherwise, we would have to define a value for targets different from the default e.g if we wanted to have it on the second qubit, we could add *targets=1*).

Once we have added a control pulse, *proc.pulses[0]* keeps all the data about it. As long as more control pulses are added, more components of *proc.pulses* will be populated. In particular, *tlist* and *coeff* can be defined with information about the time and the coefficients at each time step in a discrete way. Please notice that the coefficients are associated to the time steps and, therefore, they have one less component.

There is a nice functionality, *proc.plot*$_{pulses}$*()* that allow you to look at a plot of how the coefficients look as a function of time. I find some problems when using this functionality in conjunction with the *coeffs* definition, as in the previous code piece, while it works very well with *coeff*, so from now on I will use that. I haven't further explored on what is the cause for this behavior.

Let us now define a system where we have three qubits. The first one is controlled with sigmax(), the second one with sigmay() and the third one with sigmaz(). The coefficients are given by sin(t), cos(t) and (- 1 + mod(t-π,2π)/π), respectively, and they are succesively turnt on for lapses of 2 π. At the end, the three pulses are simultaneously turnt on for a lapse of 2 π. The definition of such a set of pulses can be written as:

```
import qutip as qt
import numpy as np
import matplotlib.pyplot as plt
from qutip.qip.device import Processor

proc = Processor(3)
proc.add_control(qt.sigmax(), targets=0)
proc.add_control(qt.sigmay(), targets=1)
proc.add_control(qt.sigmaz(), targets=2)

tlist = np.linspace(0.0, 8 * np.pi, 4000)
for i in proc.pulses:
    i.tlist = tlist[:]
```

```
proc.pulses[0].coeff = np.array([
  (np.sin(t) if t<2*np.pi or t>6*np.pi
  else 0.0)
  for t in tlist[1:]
  ])
proc.pulses[1].coeff = np.array([
  (np.cos(t) if (t > 2 * np.pi and t < 4 * np.pi) or t > 6 * np.pi
  else 0.0)
  for t in tlist[1:]
  ])
proc.pulses[2].coeff = np.array([
  (-1 + (t - np.pi) % (2 * np.pi) / np.pi if t > 4 * np.pi
  else 0.0)
  for t in tlist[1:]
  ])

proc.plot_pulses()
plt.show()
```

This is a first step to know how to deal with proper circuits for which we understand the gates as different pulses. The next natural step is going for some scheme to prescribe the pulses in an ordered way. We can, for example, tidy up the previous code by defining functions for the different pulses.

```
import qutip as qt
import numpy as np
import matplotlib.pyplot as plt
from qutip.qip.device import Processor

# system definition
proc = Processor(3)
tlist = np.linspace(0.0, 8 * np.pi, 4000)

# pulses definition
def sigmaxpulse(proc, tlist, qubit, t0, tf):
    proc.add_control(qt.sigmax(), targets=qubit)
    proc.pulses[-1].tlist = tlist[:]
    proc.pulses[-1].coeff = np.array([
      (np.sin(t) if t >= t0 and t < tf
      else 0.0)
      for t in tlist[1:]
      ])

def sigmaypulse(proc, tlist, qubit, t0, tf):
    proc.add_control(qt.sigmay(), targets=qubit)
    proc.pulses[-1].tlist = tlist[:]
    proc.pulses[-1].coeff = np.array([
      (np.cos(t) if t >= t0 and t < tf
      else 0.0)
      for t in tlist[1:]
      ])

def sigmazpulse(proc, tlist, qubit, t0, tf):
    proc.add_control(qt.sigmaz(), targets=qubit)
    proc.pulses[-1].tlist = tlist[:]
    proc.pulses[-1].coeff = np.array([
      (-1 + (t - np.pi) % (2 * np.pi) / np.pi if t >= t0 and t < tf
      else 0.0)
      for t in tlist[1:]
      ])

# circuit definition = equivalent to the previous example
sigmaxpulse(proc, tlist, 0, 0.0, 2 * np.pi)
sigmaypulse(proc, tlist, 1, 2 * np.pi, 4 * np.pi)
sigmazpulse(proc, tlist, 2, 4 * np.pi, 6 * np.pi)
sigmaxpulse(proc, tlist, 0, 6 * np.pi, 8 * np.pi)
sigmaypulse(proc, tlist, 1, 6 * np.pi, 8 * np.pi)
sigmazpulse(proc, tlist, 2, 6 * np.pi, 8 * np.pi)

proc.plot_pulses()
```

At this point, several criticisms could be made to this piece of code. First of all, once we have calibrated our gates, we do not want to be specifying all the time the initial and final times of the gates, since they will always be such that the time lapse is the optimal gate time obtained during tuneup of the gates. As a result, we now need to try to change the previous code such that what we write is a definition of the circuit as gates applied in a certain order. The second point, also related to time, is that we do not necessarily need to worry about the time the circuit will take, but the depth of the algorithm only.

This is a nontrivial issue. In the following I am going to propose a very simple way of dealing with it, assuming few different pulses and also few qubits. It is based on an encoding of the pulses as 'x', 'y' or 'z' and the qubits as '0', '1' and '2'. If we were to consider larger or more complicated systems, however, we would need to go with a more scalable approach. Another simplification we make is assuming that the optimal gate time of all the pulses is 2π, while we know that that will probably not be the case:

```
import qutip as qt
import numpy as np
import matplotlib.pyplot as plt
from qutip.qip.device import Processor

# system definition
proc = Processor(3)

# pulses definition
def sigmaxpulse(proc, tlist, qubit, t0):
    proc.add_control(qt.sigmax(), targets=qubit)
    proc.pulses[-1].tlist = tlist[:]
    proc.pulses[-1].coeff = np.array([
      (np.sin(t) if t >= t0 and t < t0 + 2 * np.pi
      else 0.0)
      for t in tlist[1:]
      ])

def sigmaypulse(proc, tlist, qubit, t0):
    proc.add_control(qt.sigmay(), targets=qubit)
    proc.pulses[-1].tlist = tlist[:]
    proc.pulses[-1].coeff = np.array([
      (np.cos(t) if t >= t0 and t < t0 + 2 * np.pi
      else 0.0)
      for t in tlist[1:]
      ])

def sigmazpulse(proc, tlist, qubit, t0):
    proc.add_control(qt.sigmaz(), targets=qubit)
    proc.pulses[-1].tlist = tlist[:]
    proc.pulses[-1].coeff = np.array([
        (-1 + (t - np.pi) % (2 * np.pi) / np.pi if t >= t0 and t < t0 + 2 * np.pi
        else 0.0)
        for t in tlist[1:]
        ])

# circuit definition = equivalent to the previous example
circuit = [['x0'], ['y1'], ['z2'], ['x0','y1','z2']]
tlist = np.linspace(0.0, 2 * np.pi * len(circuit), 1000 * len(circuit))

t = 0.0
for step in circuit:
    for gate in step:
if gate[0] == 'x':
    sigmaxpulse(proc, tlist, int(gate[1:]), t)
elif gate[0] == 'y':
    sigmaypulse(proc, tlist, int(gate[1:]), t)
elif gate[0] == 'z':
    sigmazpulse(proc, tlist, int(gate[1:]), t)
else:
    raise Exception('The gate is wrongly specified')
    t = t + 2 * np.pi

proc.plot_pulses()
```

The most natural next steps of this, which are left as excercise for the reader, are:

- The possibility that some of the pulses are longer. This would require first go through the circuit definition to recognize how long each time step must be taken and also how long the whole simulation must be, and then also take care of when each pulse is applied.
- Try the possibility of multi-qubit pulses.
- Try the possibility of the pulses being actually added to a time-independent Hamiltonian.

Of course, there is one further extension, which is learning how to properly simulate the circuit. So far, we have only defined it. This will be looked at in the future.