Designing Variational Quantum Algorithms with Tequila

Alba Cervera-Lierta
University of Toronto

Quantum conversations

Oct 28, 2020

Outlook

©hi_yo1

The golden era of quantum languages…

Qiskit

Cirq

PYQUIL

Qibo

<Q#>

PENNY LANE

And many more…

The golden era of quantum languages…

…and quantum software tools…

Qiskit

Cirq

PYQUIL

Qibo

<Q#>

PENNY LANE

And many more…

Orquestra

Mitiq

|Q⟩

PSI4

And many more…

OpenFermion

The golden era of quantum languages…
…and quantum software tools…
…plus classical tools for the NISQ era

Qiskit
Cirq
PYQUIL
Qibo
<Q#>
PENNY LANE
And many more…

Orquestra
Mitiq
|Q⟩
PSI4
And many more…
OpenFermion

JAX
SciPy
GPyOpt
And many more…
Phoenics

Qiskit Cirq JAX

PYQUIL Qibo SciPy

PENNY LANE <Q#> GPyOpt

Orquestra Mitiq Phoenics

|Q> Psi4

OpenFermion

Which language should I use?

What if I want to run the same code in different quantum computers?

What if the language doesn't contain the features that I need?

Unification, standarization, acceleration

A quantum language to simplify and accelerate implementation of new ideas for quantum algorithms.

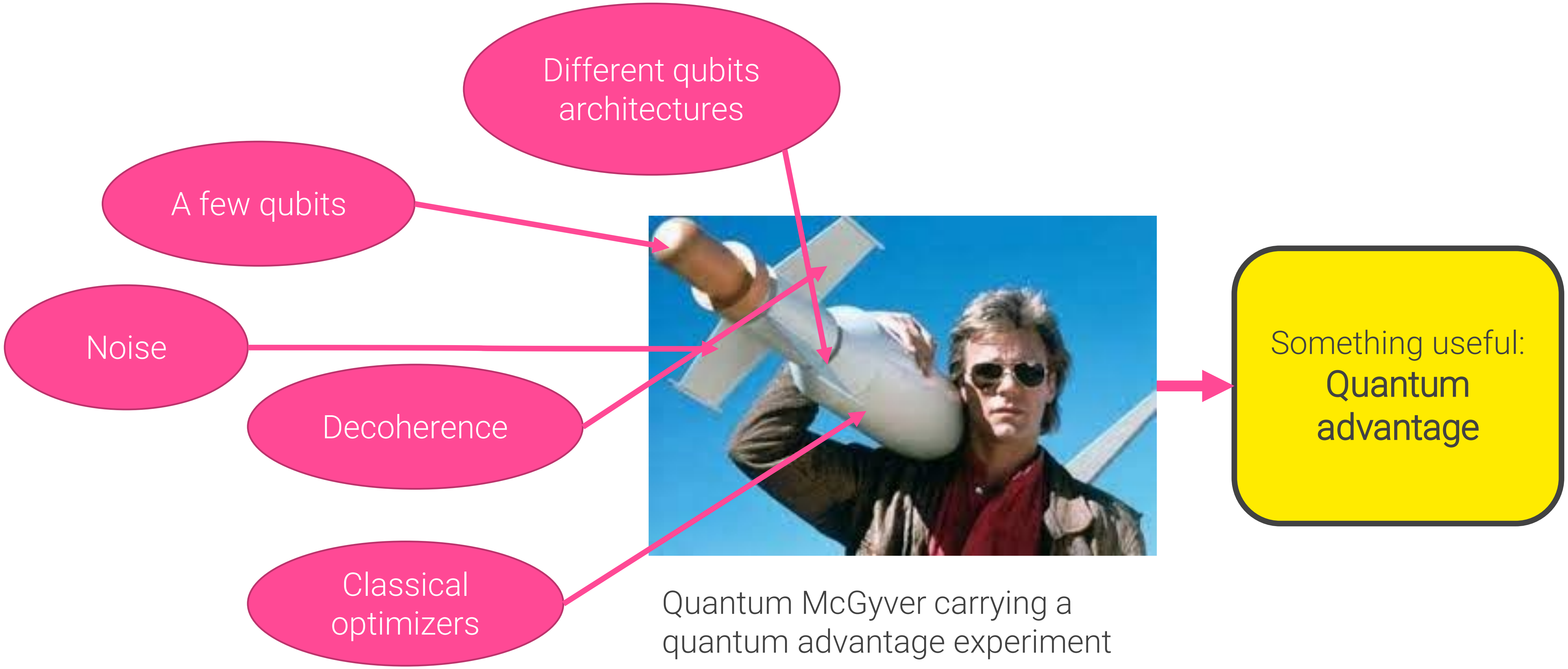
Code

https://github.com/aspuru-guzik-group/tequila

TEQUILA

Jakob S. Kottmann,[1,2,*] Sumner Alperin-Lea,[1,†] Teresa Tamayo-Mendoza,[3,1,2] Alba Cervera-Lierta,[1,2] Cyrille Lavigne,[1,2] Tzu-Ching Yen,[1] Vladyslav Verteletskyi,[1] Abhinav Anand,[1] Philipp Schleich,[4] Matthias Degroote,[1,2] Skylar Chaney,[1,5] Maha Kesebi,[1,2] Artur F. Izmaylov,[1,6] and Alán Aspuru-Guzik[1,2,7,8,‡]

Noisy Intermediate Scale Quantum computation


Different qubits architectures
A few qubits
Noise
Decoherence
Classical optimizers
Something useful: Quantum advantage

Quantum McGyver carrying a quantum advantage experiment

Noisy Intermediate Scale Quantum computation

- What can we do with a few qubits
- How can we deal with the noise → What can we do with a few noisy qubits
- Hybrid quantum-classical algorithms → Variational algorithms
- Applications: chemistry, QML, etc require the knowledge of the classical techniques to compare and test

Many quantum computers in development; need to benchmark, compare and test.

NISQ software players

Abstract manipulation

Wavefunctions
Quantum gate definition
Noise models
…

Classical tools

Optimizers
Gradient methods
CompChem
…

Quantum backends

Real (experiments)

Simulators

https://github.com/aspuru-guzik-group/tequila

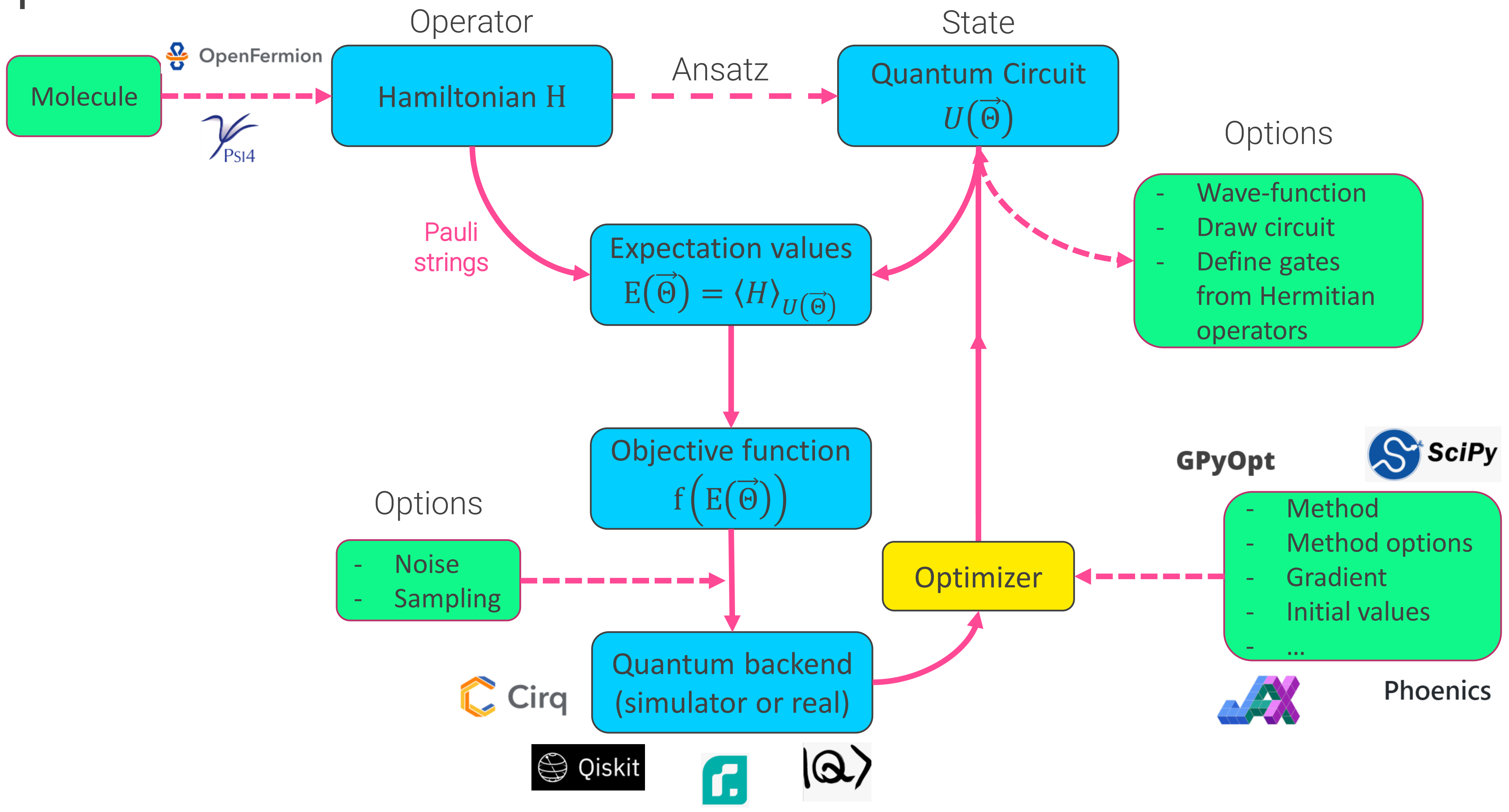
Tequila API

Code

https://github.com/aspuru-guzik-group/tequila

Jakob S. Kottmann,[1,2,*] Sumner Alperin-Lea,[1,†] Teresa Tamayo-Mendoza,[3,1,2] Alba Cervera-Lierta,[1,2] Cyrille Lavigne,[1,2] Tzu-Ching Yen,[1] Vladyslav Verteletskyi,[1] Abhinav Anand,[1] Philipp Schleich,[4] Matthias Degroote,[1,2] Skylar Chaney,[1,5] Maha Kesebi,[1,2] Artur F. Izmaylov,[1,6] and Alán Aspuru-Guzik[1,2,7,8,‡]

Tequila API

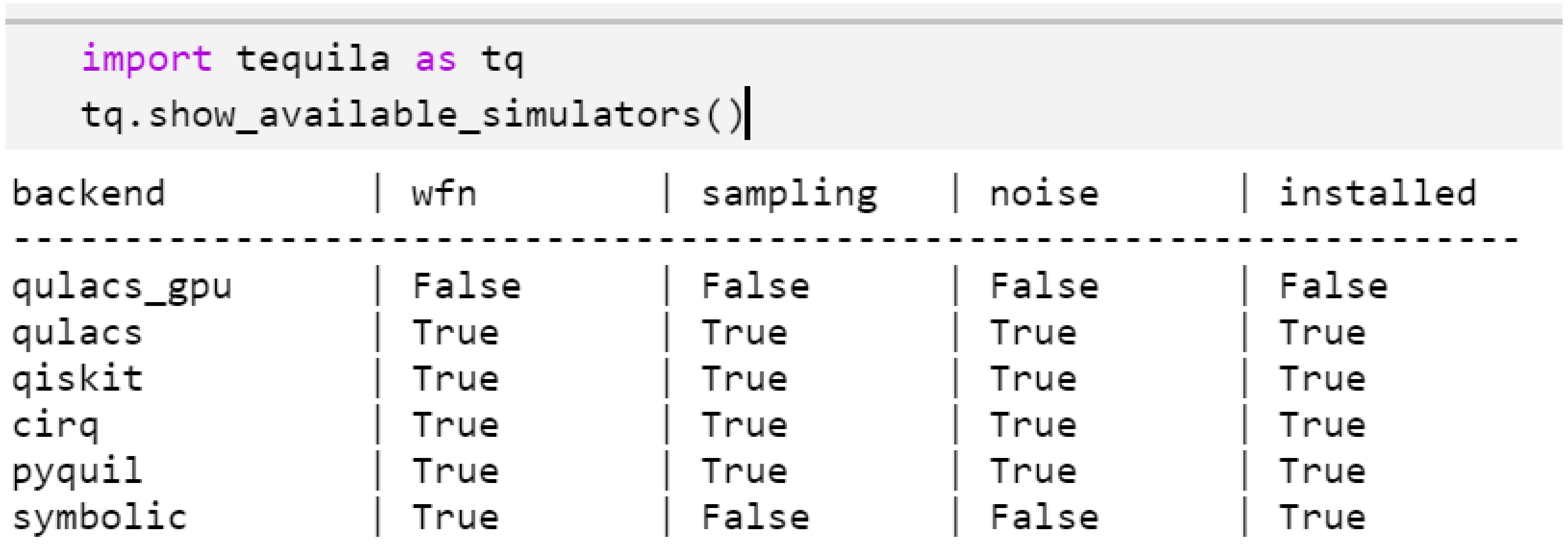


Operator
State
Molecule
OpenFermion
Psi4
Hamiltonian H
Ansatz
Quantum Circuit $U(\vec{\Theta})$
Options
- Wave-function
- Draw circuit
- Define gates from Hermitian operators
Pauli strings
Expectation values $E(\vec{\Theta}) = \langle H \rangle_{U(\vec{\Theta})}$
Objective function $f\left(E(\vec{\Theta})\right)$
Options
- Noise
- Sampling
Optimizer
GPyOpt
SciPy
- Method
- Method options
- Gradient
- Initial values
- ...
Phoenics
Quantum backend (simulator or real)
Cirq
Qiskit

https://github.com/aspuru-guzik-group/tequila

Quantum backends

src/tequila/simulators



```python
import tequila as tq
tq.show_available_simulators()
```

```
backend          | wfn          | sampling     | noise        | installed
--------------------------------------------------------------------
qulacs_gpu       | False        | False        | False        | False
qulacs           | True         | True         | True         | True
qiskit           | True         | True         | True         | True
cirq             | True         | True         | True         | True
pyquil           | True         | True         | True         | True
symbolic         | True         | False        | False        | True
```

Basic quantum gates

src/tequila/circuit/gates.py

Family	Predefined Members	Arguments: Control Target	Arguments: Angles	Arguments: Power	Syntax example	Gate
Rotational	Rx, Ry, Rz, CRx, CRy, CRz	Yes	angle θ	No	tq.gates.Rx	$e^{-i\frac{\theta}{2}\sigma_i}, i = x, y, z$
Phase	S ($\phi = \pi/2$), T ($\phi = \pi/4$)	Yes	phi ϕ	No	tq.gates.S	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}$
Pauli	X, Y, Z, CX, CY, CZ, CNOT, Toffoli	Yes	None	Yes	tq.gates.X	$\sigma_i, i = x, y, z$
Hadamard	H	Yes	None	Yes	tq.gates.H	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
SWAP	SWAP	Yes	None	Yes	tq.gates.SWAP	

General quantum gates

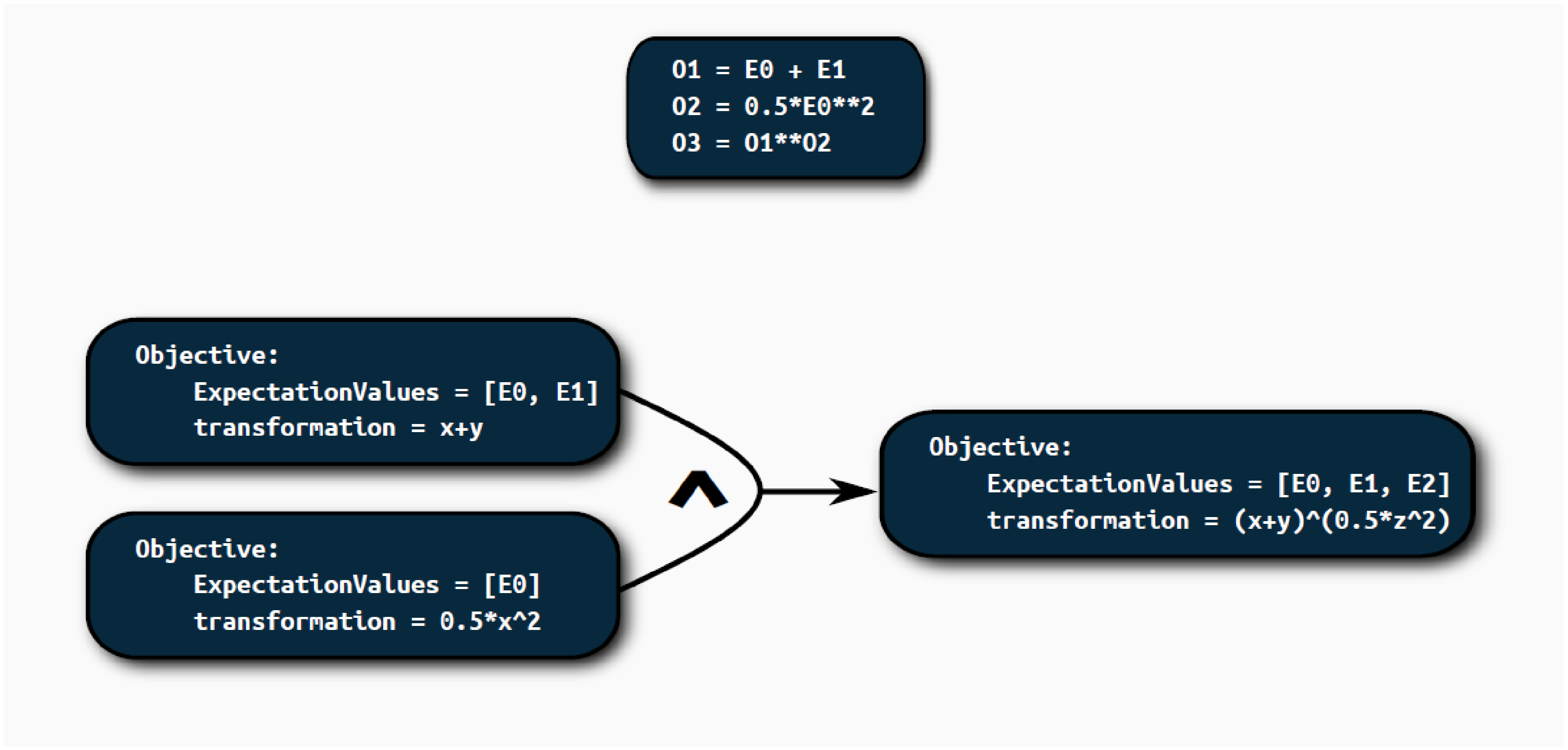
src/tequila/circuit/gates.py

Family	Predefined Members	Arguments			Syntax example	Gate
		Control Target	Angles	Power		
Exponential Pauli string	None	Yes	angle θ	No	tq.gates.Rp	$e^{-i\frac{\theta}{2}f(\sigma)}$
Trotterized	None	Yes	Generators, Angles, Trotter steps	No	tq.gates.Trotterized	

Objective: Tequila's core

src/tequila/objective

The class which represents mathematical manipulation of ExpectationValue and Variable objects.



```
O1 = E0 + E1
O2 = 0.5*E0**2
O3 = O1**O2
```

```
Objective:
    ExpectationValues = [E0, E1]
    transformation = x+y
```

```
Objective:
    ExpectationValues = [E0]
    transformation = 0.5*x^2
```

```
Objective:
    ExpectationValues = [E0, E1, E2]
    transformation = (x+y)^(0.5*z^2)
```

Optimizers

src/tequila/optimizers

tq.minimize

Mandatory arguments:

- *objective*
- *method*

For quantum simulation:

- *backend*: quantum simulator
- *samples*: circuit shots to measure (None = wf simulation)
- *device*: real or emulated quantum computer to sample from
- noise: NoiseModel object to apply to the circuits simulated.

Additional keywords:

- *method_options*: check the documentation of the method.
- *variables*: list of the Variable’s you want to optimize (the default is all of them).
- *initial_values*: which gives a start point to optimization (default is random initialization)
- *gradient*: specifies which type of gradient will be used
- *silent*: silence outputs

Optimizers

src/tequila/optimizers

```
tq.optimizers.show_available_optimizers()
```

```
NELDER-MEAD                | scipy
COBYLA                     | scipy
POWELL                     | scipy
SLSQP                      | scipy
L-BFGS-B                   | scipy
BFGS                       | scipy
CG                         | scipy
TNC                        | scipy
TRUST-KRYLOV               | scipy
NEWTON-CG                  | scipy
DOGLEG                     | scipy
TRUST-NCG                  | scipy
TRUST-EXACT                | scipy
TRUST-CONSTR               | scipy
adam                       | gd
adagrad                    | gd
adamax                     | gd
nadam                      | gd
sgd                        | gd
momentum                   | gd
nesterov                   | gd
rmsprop                    | gd
rmsprop-nesterov           | gd
Supported optimizer modules:  ['scipy', 'phoenics', 'gpyopt', 'gd']
Installed optimizer modules:  ['scipy', 'gd']
```

Gradient methods

src/tequila/optimizers

- Analytical gradients (Default):

```
gradient=None
```

- Numerical gradients:

```
gradient={'method':'2-point', "stepsize":1.e-4}
```

- Custom gradient objectives

```
gradient={tq.Variable:tq.Objective}
```

- Quantum natural gradient:

```
gradient=‘qng’
```

J. Stokes, J. Izaac, N. Killoran and G. Carleo, Quantum **4**, 269 (2020)

Gradient methods

src/tequila/optimizers

Accelerated gradient descent to achieve machine accuracy:
Direct Inversion of the Iterative Subspace (DIIS)

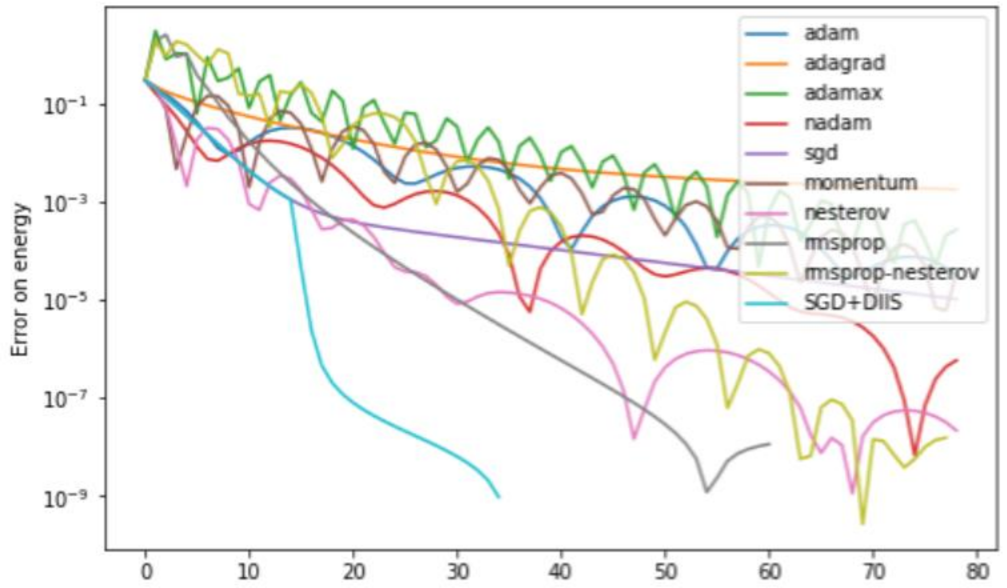
One issue with gradient descent is that they are slow when it comes to converging to machine precision.

This is not really a problem in machine learning, where convergence to many digits is not needed, but it is an issue in chemistry.

DIIS works best once we are pretty close to our solution.

DIIS kicks in when max(gradient) achieves 'tol'

```
diis={'tol':1e-1},
```



Gradient methods

src/tequila/optimizers

Numerical and Customized Gradients

Tequila offers its own way of compiling numerical gradients which can then be used throughout all gradient based optimizers.

Numerical gradients of that type can lead to significantly cheaper gradients, especially if many expectation values are involved in the objective and/or if heavy recompilation of parametrized gates is necessary.

Tequila currently offers `2-point` as well as `2-point-forward` and `2-point-backward` stencils as `method`.
The method can also be set to a python function performing the task.

Bayesian optimization

src/tequila/optimizers

Bayesian optimization is a method of global optimization, often used to tune hyperparameters in classical learning. It has also seen use in the optimization of quantum circuits.

Tequila currently supports 2 different Bayesian optimization algorithms

- Phoenics https://github.com/aspuru-guzik-group/phoenics
- GPyOpt https://github.com/SheffieldML/GPyOpt

D. Zhu et. al., Science Advances **5**, eaaw9918 (2019)

Noise models

src/tequila/circuit/noise.py

Different simulation packages handle noise in radically different ways.

- Cirq and Qulacs: use noise channels, parametrized operations which are inserted into circuits.
- Pyquil: asks its users to define noisy gate operations.
- Qiskit: takes a dictionary-like object as an argument to its simulator, and applies the noise on the user-chosen gates.

Tequila implements a simple framework for the application of noise, meant to be compatible with all our supported platforms. To do this, we make a few assumptions:

1. If noise is present, any gate may be affected by noise.
2. The noise that affects n-qubit gates is independent of the noise on m-qubit gates.
3. Noise probabilities are independent of position in the circuit.
4. The number of qubits involved in a gate, not the operation performed, dictates what noises may occur (e.g. CNOT gate is not noisier than Controlled-Z gate).

Noise models

Tequila at present supports six common quantum noise operations, all of which can at present be employed by all the noise-supporting simulation backgrounds:

1. *Bit flips*: a probabilistic application of pauli X.
2. *Phase flips*: a probabilistic application of pauli Z.
3. *Amplitude damps*: take qubits in state |1> to |0>.
4. *Phase damps*: different formalization of the phase flip.
5. *Phase-Amplitude damps*: simultaneously perform said operations.
6. *(Symmetric) depolarizing*: (equi)probabilistically performs pauli X, Y, and Z.

NoiseModel's combine with each other through addition, creating a new NoiseModel with all the operations of the two summands.

```
noise=my_noise_model
```

Noise is only supported when sampling.

Tequila supports the use of device-noise-emulation for those backends.

```
noise='device'
```

Basic usage

Creating quantum circuits

```
circuit = tq.gates.H(target=0) + tq.gates.CNOT(target=1,control=0)
```

```
print(circuit)
```

```
circuit:
H(target=(0,))
X(target=(1,), control=(0,))
```

```
tq.draw(circuit)
```

```
0: ───H───@───
          │
1: ───────X───
```

```
tq.draw(circuit, backend='qiskit')
```

```
     ┌───┐     
q_0: ┤ H ├──■──
     └───┘┌─┴─┐
q_1: ─────┤ X ├
          └───┘
c: 2/══════════
```

```
circuit = tq.gates.H(target=[0,1]) + tq.gates.X(target=1, control=0)
tq.draw(circuit)
```

```
0: ───H───@───
          │
1: ───H───X───
```

Quantum circuit gates

Predefined gate vs Pauli string + control-target definition

```
tq.gates.Ry(angle=1.0, target=0) + tq.gates.X(target=1, control=0)
```

```
tq.gates.Rp(angle=1.0, paulistring="Y(0)") + tq.gates.X(target=1, control=0)
```

Power gates

```
tq.gates.Y(power=0.5, target=0) + tq.gates.Ry(angle=1.0, target=1, control=0)
```

Pauli string vs Trotterization

```
tq.gates.Rp(angle=1.0, paulistring="X(0)Y(1)")
```

```
generator = tq.paulis.X(0)*tq.paulis.Y(1)
tq.gates.Trotterized(generators=[generator], angles=[1.0], steps=1)
```

Wavefunction and measurements

```
wfn = tq.simulate(circuit, backend='qulacs')
print(wfn)
```

```
+0.5000|00> +0.5000|10> +0.5000|01> +0.5000|11>
```

```
measurements = tq.simulate(circuit, samples=10)
print(measurements)
```

```
+10.0000|00>
```

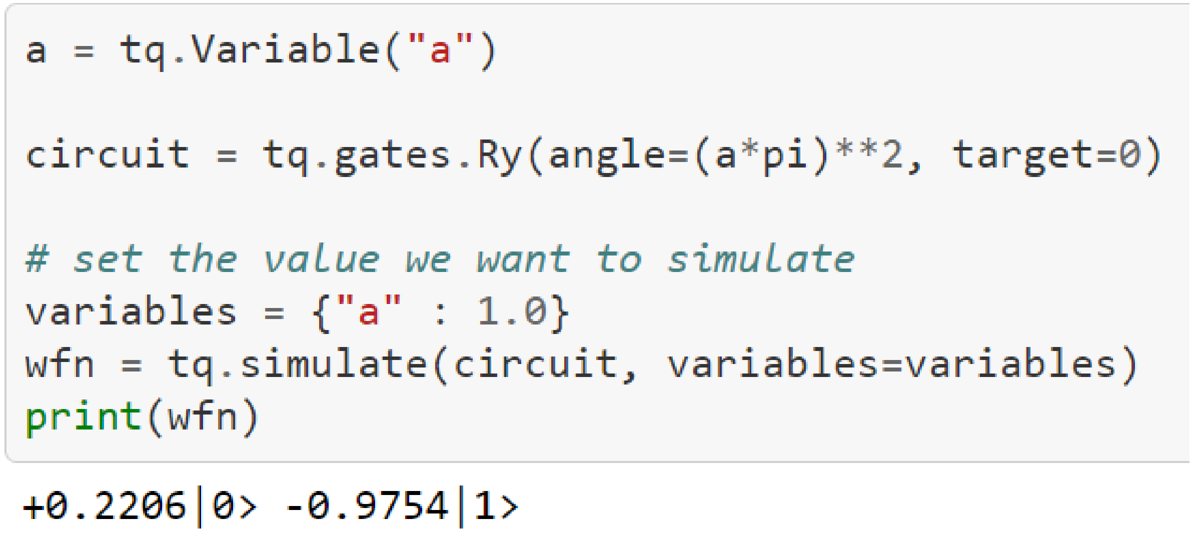
```
print(measurements(0))
print(measurements("00"))
print(measurements(2))
print(measurements("10"))
```

```
10
10
0.0
0.0
```

```
measurements = tq.simulate(circuit+tq.gates.Measurement(target=[0]), samples=10)
print(measurements)
```

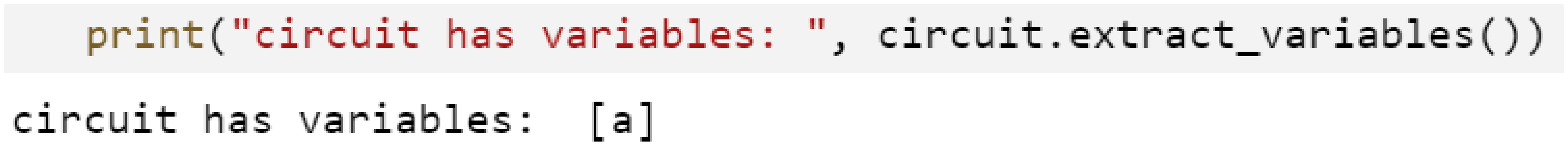
```
+6.0000|00> +4.0000|10>
```

Parameterized quantum circuits



```
a = tq.Variable("a")

circuit = tq.gates.Ry(angle=(a*pi)**2, target=0)

# set the value we want to simulate
variables = {"a" : 1.0}
wfn = tq.simulate(circuit, variables=variables)
print(wfn)
```

```
+0.2206|0> -0.9754|1>
```



```
print("circuit has variables: ", circuit.extract_variables())
```

```
circuit has variables:  [a]
```

Define a Hamiltonian

```python
# Pauli Operators can be initialilzed and added/multipled
H = tq.paulis.X(qubit=[0,1,2,3]) + tq.paulis.Y(2) + tq.paulis.Z(qubit=[0,1])*tq.paulis.X(2)
print(H, " is hermitian = ", H.is_hermitian())

H = tq.paulis.Z(0)*tq.paulis.Y(0) + tq.paulis.X(0)
print(H, " is hermitian = ", H.is_hermitian())

hermitian_part, anti_hermitian_part = H.split()
print("hermitian part      = ", hermitian_part)
print("anti-hermitian part = ", anti_hermitian_part)

H = tq.paulis.Projector("|00>")
print(H, " is hermitian = ", H.is_hermitian())

H = tq.paulis.Projector("1.0*|00> + 1.0*|11>")
print(H, " is hermitian = ", H.is_hermitian())
```

```
+1.0000X(0)X(1)X(2)X(3)+1.0000Y(2)+1.0000Z(0)Z(1)X(2)  is hermitian =  True
+1.4142e^(-0.2500πi)X(0)  is hermitian =  False
hermitian part      =  +1.0000X(0)
anti-hermitian part =  -1.0000iX(0)
+0.2500+0.2500Z(1)+0.2500Z(0)+0.2500Z(0)Z(1)  is hermitian =  True
+0.5000+0.5000Z(0)Z(1)+0.5000X(0)X(1)-0.5000Y(0)Y(1)  is hermitian =  True
```

Create an objective

```
# the circuit
U = tq.gates.Ry(angle="a", target=0)
# the Hamiltonian
H = tq.paulis.X(0)
# the Objective (a single expectation value)
E = tq.ExpectationValue(H=H, U=U)
print("Hamiltonian ", H)
print(E)
```

```
Hamiltonian  +1.0000X(0)
Objective with 1 unique expectation values
variables = [a]
types     = not compiled
```

```
compiled_objective = tq.compile(E)

# the compiled objective can now be used like a function
for value in [0.0, 0.5, 1.0]:
    evaluated = compiled_objective(variables={"a": value})
    print("objective({}) = {}".format(value, evaluated))
```

```
objective(0.0) = 0.0
objective(0.5) = 0.479425538604203
objective(1.0) = 0.8414709848078965
```

Objectives can be differenciated!

```
L = E

dLda = tq.grad(L, "a")
d2Ld2a = tq.grad(dLda, "a")
```

All in one

$$L = \langle H \rangle_{U(a)} + e^{-\left(\frac{\partial}{\partial a}\langle H \rangle_{U_a}\right)^2}$$

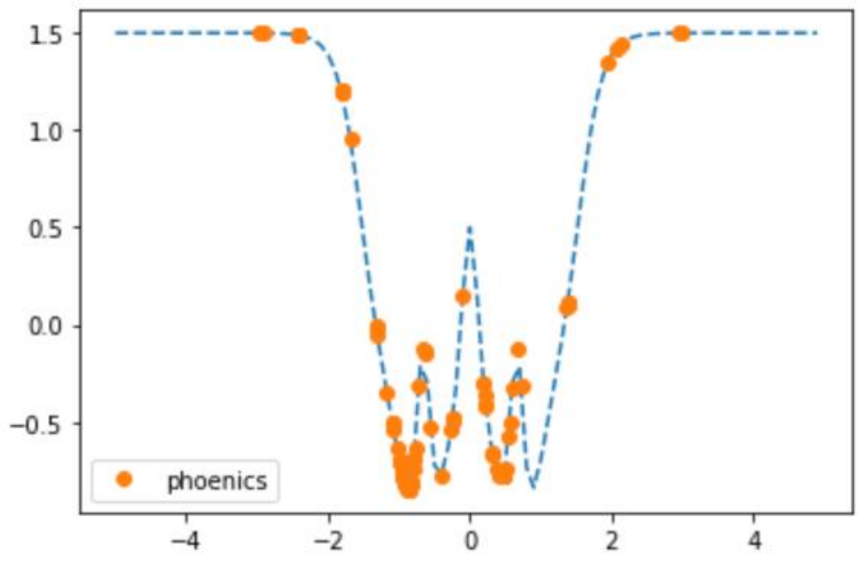
$$H = -X(0)X(1) + \frac{1}{2}Z(0) + Y(1)$$

$$U = e^{-\frac{e^{-a^2}}{2}Y(0)}\mathrm{CNOT}(0,1)$$

```
a = tq.Variable("a")
U = tq.gates.Ry(angle=(-a**2).apply(tq.numpy.exp)*pi, target=0)
U += tq.gates.X(target=1, control=0)

H = tq.QubitHamiltonian.from_string("-1.0*X(0)X(1)+0.5Z(0)+Y(1)")

E = tq.ExpectationValue(H=H, U=U)
dE = tq.grad(E, "a")

objective = E + (-dE**2).apply(tq.numpy.exp)
```

```
phoenics_config = {'general': {'auto_desc_gen': 'False', 'batches': 5, 'boosted': 'Fal
'False'}, 'parameters': [{'name': a, 'periodic': 'True', 'type': 'continuous', 'size':
'high': 3.}], 'objectives': [{'name': 'Energy', 'goal': 'minimize'}]}
result = tq.minimize(method='phoenics', objective = objective, phoenics_config = phoer
maxiter=10)
energies = result.history.energies
angles = result.history.angles
```

Points that phoenics visited



A chemistry example

Define the molecule

```python
molecule = tq.chemistry.Molecule(geometry = "H 0.0 0.0 0.0\nLi 0.0 0.0 1.6", basis_set="sto-3g")
print(molecule)
```

```
Qubit Encoding
transformation=<function jordan_wigner at 0x7f0454998ae8>
basis_set        : sto-3g
geometry         : H 0.0 0.0 0.0
Li 0.0 0.0 1.6
description      :
multiplicity     : 1
charge           : 0
closed_shell     : True
name             : molecule

Psi4 Data
Point Group (full) : c_inf_v
Point Group (used) : c2v
nirrep           : 4
irreps           : ['A1', 'A2', 'B1', 'B2']
mos per irrep    : [4, 0, 1, 1]
```

```python
for orbital in molecule.orbitals:
    print(orbital)
```

```
0 : 0A1 energy = -2.348839
1 : 1A1 energy = -0.285276
2 : 2A1 energy = +0.078216
3 : 0B1 energy = +0.163950
4 : 0B2 energy = +0.163950
5 : 3A1 energy = +0.547769
```

Obtain the Hamiltonian

Using the Jordan-Wigner transformatin (default)

```
H = molecule.make_hamiltonian()
```

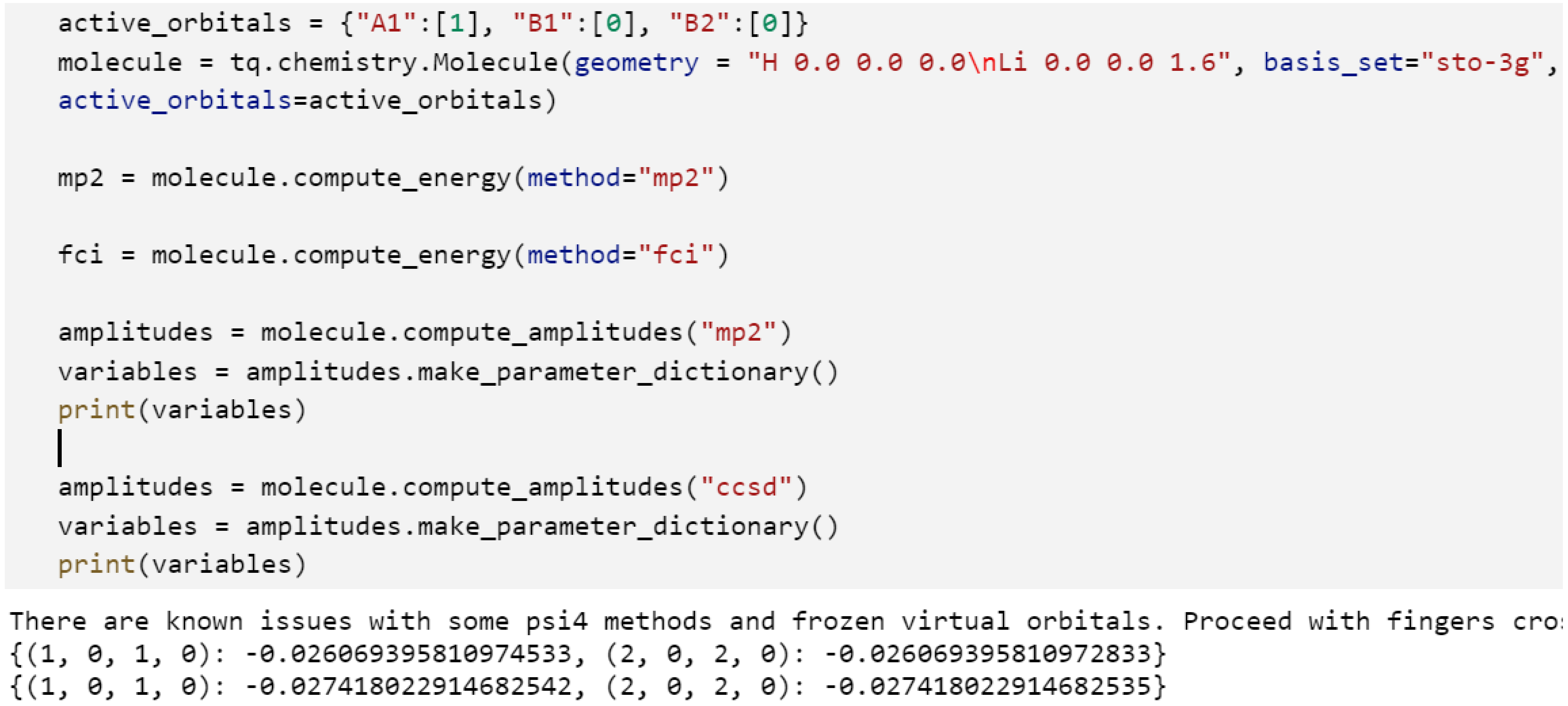
Specifying the transformation

```
molecule = tq.chemistry.Molecule(geometry = "H 0.0 0.0 0.0\nLi 0.0 0.0 1.6", basis_set="sto-3g",
transformation="bravyi-kitaev")
H = molecule.make_hamiltonian()
```

Select active space (to reduce the number of terms of the Hamiltonian)

```
active_orbitals = {"A1":[1,2], "B1":[0]}
molecule = tq.chemistry.Molecule(geometry = "H 0.0 0.0 0.0\nLi 0.0 0.0 1.6", basis_set="sto-3g",
active_orbitals=active_orbitals)
H = molecule.make_hamiltonian()
```

Classical methods



```python
active_orbitals = {"A1":[1], "B1":[0], "B2":[0]}
molecule = tq.chemistry.Molecule(geometry = "H 0.0 0.0 0.0\nLi 0.0 0.0 1.6", basis_set="sto-3g",
active_orbitals=active_orbitals)

mp2 = molecule.compute_energy(method="mp2")

fci = molecule.compute_energy(method="fci")

amplitudes = molecule.compute_amplitudes("mp2")
variables = amplitudes.make_parameter_dictionary()
print(variables)

amplitudes = molecule.compute_amplitudes("ccsd")
variables = amplitudes.make_parameter_dictionary()
print(variables)
```

```
There are known issues with some psi4 methods and frozen virtual orbitals. Proceed with fingers cro
{(1, 0, 1, 0): -0.026069395810974533, (2, 0, 2, 0): -0.026069395810972833}
{(1, 0, 1, 0): -0.027418022914682542, (2, 0, 2, 0): -0.027418022914682535}
```

https://github.com/aspuru-guzik-group/tequila

VQE example: LiH

```python
# define a molecule within an active space
active = {"a1": [1], "b1":[0]}
molecule = tq.quantumchemistry.Molecule(geometry="lih.xyz", basis_set='6-31g', active_orbitals=active, transformation="bravyi-kitaev")

# get the qubit hamiltonian
H = molecule.make_hamiltonian()

# make the UCCSD ansatz with cc2 ordering
U = molecule.make_uccsd_ansatz(initial_amplitudes="cc2", trotter_steps=1)

# define the expectationvalue
E = tq.ExpectationValue(H=H, U=U)

# compute reference energies
fci = molecule.compute_energy("fci")
cisd = molecule.compute_energy("detci", options={"detci__ex_level": 2})

# optimize
result = tq.minimize(objective=E, method="BFGS", gradient="2-point", method_options={"eps":1.e-3}, initial_values={k:0.0 for k in E.extract_variables()})

print("VQE : {:+2.8}f".format(result.energy))
print("CISD: {:+2.8}f".format(cisd))
print("FCI : {:+2.8}f".format(fci))
```

A QML example

Based on

Data re-uploading for a universal quantum classifier
A. Pérez-Salinas, A. Cervera-Lierta, E. Gil-Fuster and J. I. Latorre
Quantum **4**, 226 (2020).

Fidelity and wavefunctions

We can obtain a wavefunction from:

1. String
2. Array
3. Quantum circuit

```
wfn_string = tq.QubitWaveFunction.from_string(string="1.0*|00> + 1.0*|11>")
wfn_array = tq.QubitWaveFunction.from_array(np.asarray([1,0,0,1]))
wfn_array_norm = wfn_array.normalize() # remeber no normalize!
print(wfn_string)
print(wfn_array_norm)
```

```
+1.0000|00> +1.0000|11>
+0.7071|00> +0.7071|11>
```

```
# Quantum circuit
qc = tq.gates.Ry(target=1,angle=0.5) + tq.gates.CNOT(target=1,control=0)
wfn_qc = tq.simulate(qc) # Simulate the wavefunction
print(wfn_qc)
```

```
+0.9689|00> +0.2474|01>
```

2 methods to compute the fidelity:

1

```
wfn_targ = wfn_array_norm
fidelity = abs(wfn_targ.inner(wfn_qc))**2
print('fidelity = ', fidelity)
```

```
fidelity =  0.4693956404725931
```

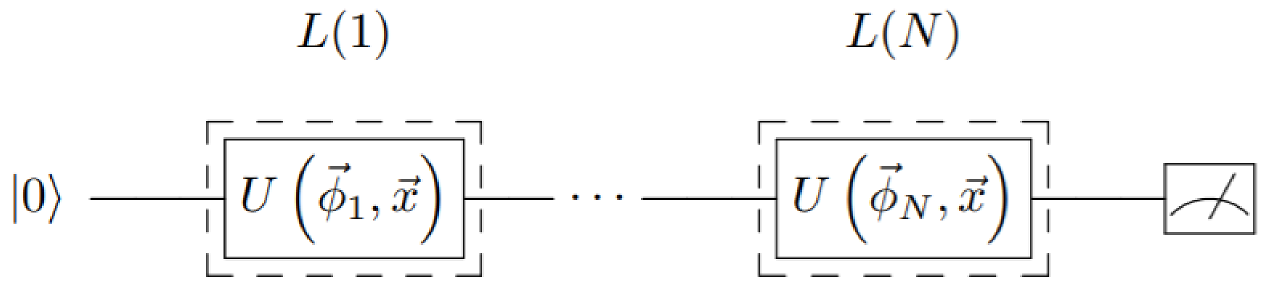
2

```
# construct the density operator of target state
rho_targ =  tq.paulis.Projector(wfn=wfn_targ)
print(rho_targ)
O = tq.Objective.ExpectationValue(U=qc, H=rho_targ)
fidelity= tq.simulate(O)
print('fidelity = ', fidelity)
```

```
+0.2500+0.2500Z(0)Z(1)+0.2500X(0)X(1)-0.2500Y(0)Y(1)
fidelity =  0.469395640472593
```

Single-qubit quantum classifier

The model:


$L(1)$
$L(N)$
$|0\rangle$
$U\left(\vec{\phi}_1, \vec{x}\right)$
$\cdots$
$U\left(\vec{\phi}_N, \vec{x}\right)$

The layer:

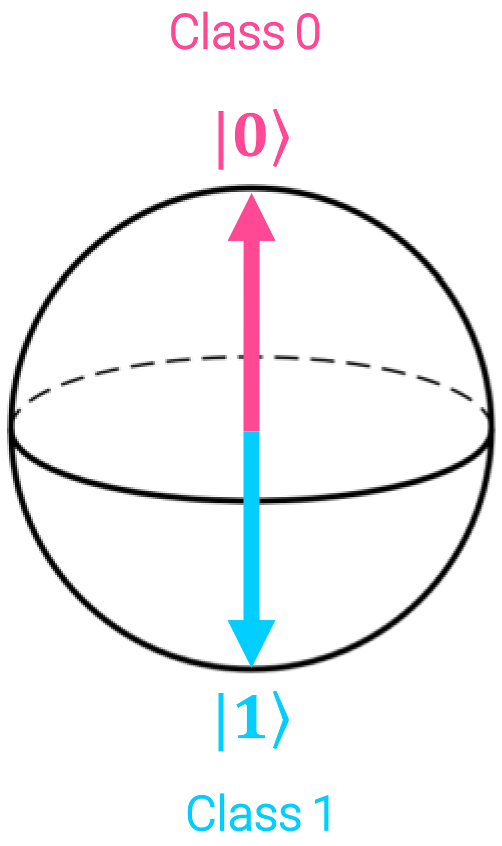
$$L(i) = U\left(\vec{\theta}_i + \vec{w}_i \circ \vec{x}\right)$$

The cost function:

Circuit state wavefunction

$$\chi_f^2(\vec{\theta}, \vec{w}) = \sum_{\mu=1}^{M} \left(1 - |\langle \tilde{\psi}_s | \psi(\vec{\theta}, \vec{w}, \vec{x_\mu}) \rangle|^2\right)$$

Target state wavefunction



Single-qubit quantum classifier

Target state wavefunction generator

```
def targ_wfn(y, nclass):
    if y==0:
        wfn = tq.QubitWaveFunction.from_array(np.asarray([1,0]))
    if y==1:
        wfn = tq.QubitWaveFunction.from_array(np.asarray([0,1]))
    return wfn
```

Quantum classifier

```
def qcircuit(xval, param):
    layers = int((len(param))/2) # 2 parameters/layer
    # initialize the circuit
    qc = tq.QCircuit()
    for p in range(0,2*layers-1):
        # add layers to the circuit
        qc += tq.gates.Ry(xval[0] + param[p],0)
        qc += tq.gates.Rz(xval[1] + param[p+1],0)
    return qc
```

Loss function generator

```
# Fidelity objective
def fid(wfn_targ, qc):
    rho_targ =  tq.paulis.Projector(wfn=wfn_targ)
    O = tq.Objective.ExpectationValue(U=qc, H=rho_targ)
    return O

# cost function
def cost(x, y, param, nclass):
    loss = 0.0
    for i in range(len(y)):
        # state generated by the classifier
        qc = qcircuit(x[i], param)
        # fidelity objective respect to the label state
        f = fid(targ_wfn(y[i],nclass), qc)
        loss = loss + (1 - f)**2
    return loss / len(x)
```

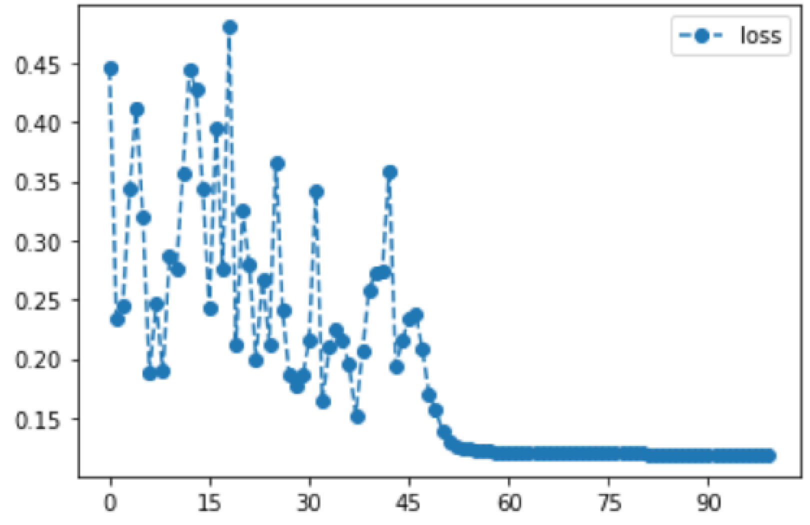
Single-qubit quantum classifier

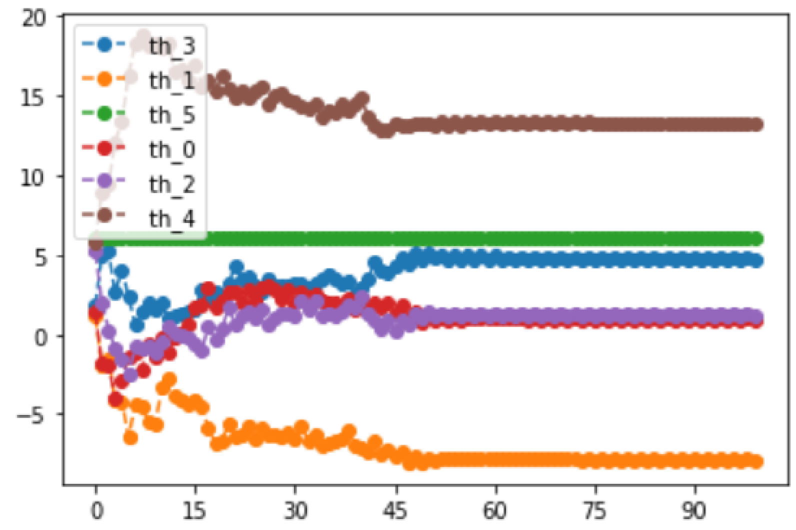
Training

```
layers = 3
training_set = 400
# generate the training set and its corresponding labels
xdata, ydata = circle(training_set)
# generate the variational parameters
param = [tq.Variable(name='th_{}'.format(i)) for i in range(0,2*layers)]
# initialize the variational parameters
inval = {key : random.uniform(0, 2*np.pi) for key in param}
# Optimization parameters
grad = '2-point' # numerical gradient
mthd = 'rmsprop' # minimization method
mthd_opt = {'eps':1.e-4} # method options
# objective to be optimized: cost function
obj = cost(xdata, ydata, param, nclass)

test = tq.minimize(objective=obj, initial_values=inval, method = mthd,
                   gradient = grad, method_options = mthd_opt, silent=False)
```

```
print("loss = ", test.energy)
print(test.history.plot('energies', label='loss'))
print(test.history.plot('angles', label=""))
```

```
loss =  0.11884156841701687
```



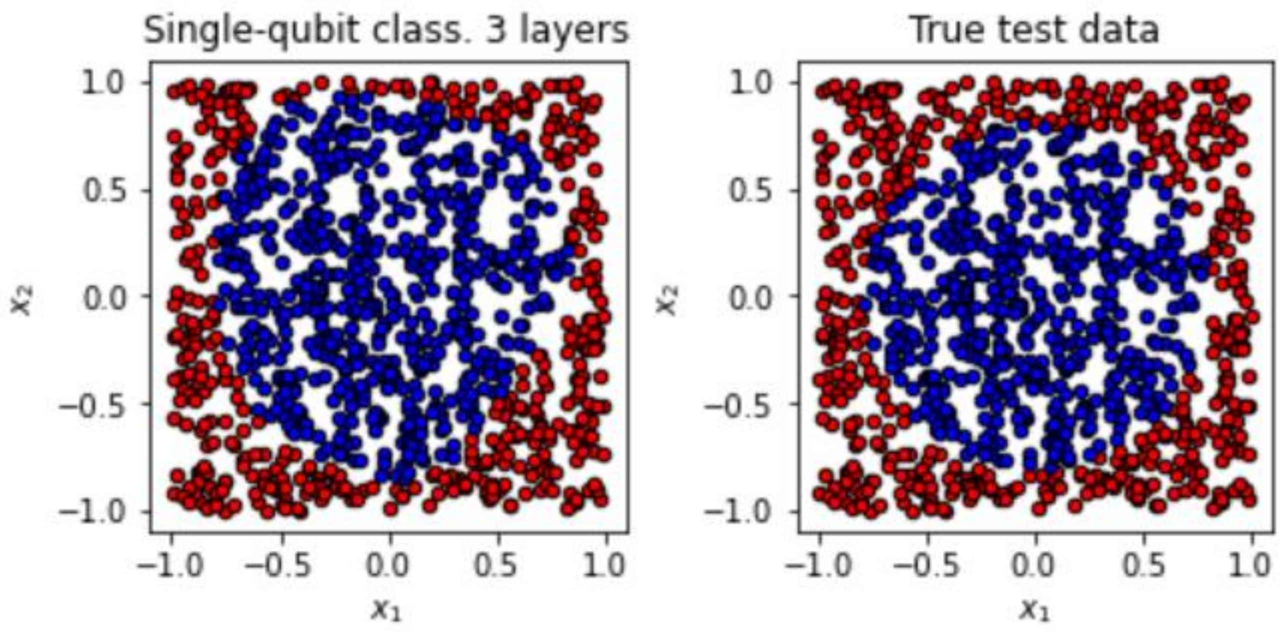


Single-qubit quantum classifier

```python
test_set = 1000
suc = 0 # success
suc_rand = 0 # random success
for i in range(test_set):
    # random test point
    x = 2 * (np.random.rand(2)) - 1
    # state generated by the trained classifier
    qc = qcircuit(x, param)
    wfn_qc = tq.simulate(qc, variables=test.angles)
    # compute the fidelity respect to one of the label states
    f = abs(wfn_qc.inner(targ_wfn(0,nclass)))**2
    y = 1
    # if fidelity is >= 0.5, this state belongs to |0> class
    # (|1> class otherwise)
    if f >= 0.5:
        y = 0
    # check the real class of the data point
    y_real = circle(random=False, x_input=x)
    # compute success rate
    if y == y_real:
        suc = suc + 1
    # compute random success rate
    yrand = np.random.randint(0, nclass-1)
    if yrand == y_real:
        suc_rand = suc_rand + 1
print("success %: ", 100*suc/test_set,"%")
print("random success %: ", 100*suc_rand/test_set,"%")
```

Test

```
success %:   90.5 %
random success %:   49.1 %
```



Projects that use Tequila

J.S. Kottmann, P. Schleich, T. Tamayo-Mendoza, A. Aspuru-Guzik.
A basis-set-free approach for VQE employing pair-natural orbitals.
arxiv.org/abs/2008.02819 example code

A. Cervera-Lierta, J.S. Kottmann, A. Aspuru-Guzik.
The Meta-Variational Quantum Eigensolver.
arxiv.org/abs/2009.13545 example code

J.S. Kottmann, M. Krenn, T.H. Kyaw, A. Aspuru-Guzik.
Quantum Computer-Aided design of Quantum Optics Hardware.
arxiv.org/abs/2006.03075 example code

Tequila 2.0

New quantum backends

- Orquestra
- Qibo
- PyQuest
- IntelQS

New libraries

- Mitiq
- TensorFlow

Any suggestions, comments and recommendations?

Would you like to be part of Tequila 2.0?

Contact us!

Alán Aspuru-Guzik

the matter lab

VECTOR INSTITUTE | INSTITUT VECTEUR

CIFAR

Jakob Kottmann

Sumner Alperin

Teresa Tamayo

Cyrille Lavigne

Abhinav Anand

Philipp Schleich

Matthias Degroote

Skylar Chaney

+

Tzu-Ching Yen
Vladislav Vertelezkyi

Maha Kesebi
Artur Izmaylov