Introductory Tutorial for SPSA and the Quantum Approximation Optimization Algorithm

Salonik Resch, University of Minnesota - Twin Cities
resc0059@umn.edu

Abstract

This short tutorial provides an introduction to the Quantum Approximation Optimization Algorithm (QAOA). Specifically, how to use QAOA with the Simultaneous Perturbation Stochastic Approximation (SPSA) algorithm to solve the Max-Cut problem. All steps of the algorithm are explicitly shown and no theory or complex mathematics are used. The focus is entirely on setting up a practical implementation. Fully functional examples in both R and python (using Qiskit) are provided, both using roughly 100 lines of code.

1 Introduction

The purpose of this tutorial is to provide an explicit description how to use the Simultaneous Perturbation Stochastic Approximation (SPSA) [10] in tandem Quantum Approximation Optimization Algorithm (QAOA) [3]. A brief overview of both is provided, before going over a concrete example. The most notable feature of this tutorial is that the optimization does not occur "off-screen", every step of the algorithm is visible in the examples and provided code.[1] The end result is a simple implementation, but it should be helpful for getting started.

Mathematical background and theory is skipped for clarity, it is not necessary to understand in order to get started. A familiarity with quantum computation (qubits and quantum gates) is assumed. For a background on the basics of quantum computing, I highly recommend the text by Michael Loceff [6].

The input problem, Max-Cut, is described in Section 2. The QAOA circuit which can solve Max-Cut is presented in Section 3. The basics of SPSA are then covered in Section 4. After a summary in Section 5, Section 6 goes over code which implements all components. Appendices A and B provide the entire code, which is also available on github [12].

2 Max-Cut

Before discussing the solution, let us first describe the problem. Max-Cut is a binary combinatorial optimization problem. This means we are searching for a set of binary values (which can be represented by a bitstring) which produces an optimal solution to a given cost function. If $\mathbf{z}$ is a bitstring with n bits

$$\mathbf{z} = z_0 z_1 z_2 ... z_{n-1}$$

we are looking for a value of $\mathbf{z}$ such that a cost function C is maximized

$$argmax_{\mathbf{z}} \, C(\mathbf{z})$$

The cost function consists of a set of *clauses*, which represent constraints on the bitstring that we want to be satisfied. The input to the cost function is a bitstring (a suggested solution) and the output is an integer specifying how many of the clauses were satisfied by the bitstring.

$$C(\mathbf{z}) = \# \, of \, clauses \, satisfied \, by \, \mathbf{z}$$

For max-cut, the clauses are very simple - each clause specifies two bits in the bitstring that should be opposite values. For example, one clause might be bit 2 and bit 3 should be opposite, $z_2! = z_3$. A wide range of optimization problems

---

[1]With the major exception of the backend quantum mechanics simulations.

(a) A graph with 4 vertices (variables) and 4 edges (constraints)

(b) 0000 has score 0

(c) 0011 has score 2
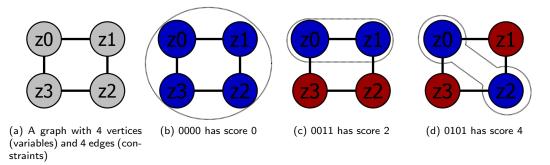
(d) 0101 has score 4

Figure 1: Input to max-cut can be represented by a graph where vertices are variables and edges are constraints. The problem can then be considered a graph coloring, where there are two colors (eg Blue=0, Red=1) and edges should connect vertices of different colors. The number of edges that cross color boundaries ("cut" the set boundaries) represents the score of the coloring.

can be mapped into such a format, such as 3SAT [8], hence using this restricted set of clauses does not significantly limit the problems we can solve. Given a bitstring, it is easy to compute how many of the clauses were satisfied. The hard part is finding the bitstring that satisfies the most clauses. As the problem size grows (the length of the bitstring), the number of possible solution bitstrings grows exponentially. Hence, a brute force search will be intractable.

2.1 Max-Cut Example

The input to max-cut can be described as a graph, where vertices represent the binary variables ($z_0 z_1 z_2 ... z_{n-1}$) and edges between vertices represent the constraints. If an edge connects vertices i and j, this means that $z_i != z_j$ is one of the constraints. The problem of finding an optimal bitstring $\mathbf{z} = z_0 z_1 z_2 z_3$ is equivalent to finding an optimal 2-coloring[2] of the graph. Optimal means that as many edges as possible connect vertices of different colors.

Let's use an example problem from Pennylane [7], which has a good tutorial on QAOA. A graph with four vertices (variables) and four edges (constraints) is shown in Figure 1a. The four constraints are $z_0 != z_1$, $z_1 != z_2$, $z_2 != z_3$, and $z_3 != z_0$. We can try to color the graph in order to satisfy as many of the constraints as possible. For example, in Figure 1b we can set $\mathbf{z} = 0000$ (all blue). In this case no edges cross from blue to red, meaning that no constraints are satisfied, and the score is 0. Figures 1c and 1d show better colorings, where edges do connect vertices of different colors. In this case, all 4 constraints can be satisfied (though this is not typical). The score is also referred to as the *cut*, because if a line is drawn around all vertices of the same color, it passes through (or cuts) edges of the graph. Hence the name Max-Cut.

If given a bitstring (graph coloring), we can quickly check how many constraints it satisfies (the score). This involves iterating through each of the constraints (edges), which all have the form $z_i != z_j$, and checking whether bits (vertices) i and j are different values (colors). We get a single point for each constraint satisfied. This has low computational complexity (linear in the number of constraints). Even for very large problems a classical computer can easily do this.

3 Quantum Approximation Optimization Algorithm

The quantum approximation optimization algorithm (QAOA) [3] is a quantum algorithm which provides an approximate solution to max-cut. At small sizes, which can be run on current quantum computers, the algorithm is much slower than alternative approaches. However, at larger scales it is believed QAOA will provide an advantage [4].

QAOA is a *variational* quantum algorithm, where a quantum computer and a classical computer work together to get the solution. A quantum circuit[3] runs on the quantum computer, where the output provides a guess at the solution. The classical computer then evaluates the quality of that solution, and uses an optimization algorithm to modify the quantum

---

[2]This is also called a graph *cut*, as the coloring is equivalent to a partition and edges are "cut" by set boundaries when crossing from one set (color) to the other.

[3]"circuit" is the term used for "program" in quantum computing literature. A quantum circuit is just a list of quantum gates (operations/instructions) to perform.
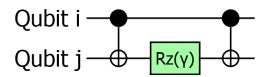
Figure 2: A controlled phase operation between qubit i and qubit j. Two CNOT gates and a single-qubit z-rotation is used. The angle of the z-rotation is set by γ, which is a classical parameter.

circuit to try to provide a better solution (hence the use of SPSA, covered in Section 4). The classical computer is trying to "guide" the quantum computer towards better solutions. This process is similar to training for machine-learning, however we don't have access to precise gradients (or any information) within the quantum computer - so we can't use tricks like backpropagation or similar machine-learning approaches. We will see that the *structure* of the quantum circuit comes from the input problem, and the specific operations performed in the quantum circuit will be determined (parameterized) by *classical parameters*, which are set by the classical computer.

The reason we covered the graph representation of max-cut in Section 2.1 is because it provides a nice visual of the conversion between the max-cut problem and a quantum circuit which solves it. We "map" each vertex in the graph to a single qubit. Hence, the number of qubits required is equal to the number of vertices (bits) of the input problem. The 4-bit problem in Figure 1 will require 4 qubits to solve. At the end of the quantum circuit, we perform a measurement on the qubits. This produces a bitstring, one bit for each qubit. These bits (bitstring) can be interpreted as graph colorings, just as in Figure 1. So once we get the bitstring, we can quickly check how good of a score it produces. For example, if we measure $|0011\rangle$, this corresponds to the coloring in Figure 1c, and we can check to see that it produces a score of 2 on our max-cut problem.

Let's now describe the quantum circuit for QAOA, which is shown in Figure 3 for the input problem in Figure 1a. The qubits are initialized into the all zero state $|0000\rangle$. The first operation is a Hadamard (H) gate on all qubits, which puts them into an even superposition of all states.

We then do a sequence of controlled-phase operations. A single controlled phase operation can be performed with three sequential operations. Say we are doing a controlled-phase operation between qubit i and qubit j. First, a CNOT gate is performed with qubit i as the control and qubit j as the target. Then a single-qubit z-rotation is performed on qubit j. Finally, another CNOT is performed, again with qubit i as the control and qubit j as the target. This is shown in Figure 2. Note that the z-rotation is parameterized by a parameter γ, which sets the angle of rotation

The sequence of controlled-phase operations that we need to perform is determined by the input graph. For every edge in the graph, we perform a single controlled-phase rotation. If there is an edge between vertices i and j, we perform a controlled phase rotation between qubit i and qubit j. However, it does not matter which qubit is the control and which is the target. Additionally, the order of the controlled-rotations can permuted [1], any order is fine. The same γ is used for all controlled phase rotations. γ is set by the classical computer, which is covered in Section 4.

Once all the controlled phase operations have been completed, single qubit x-rotations are performed on all qubits. The x-rotations are parameterized by β (also set by the classical computer), which is the same for all qubits. After the circuit is finished, a measurement is performed on the qubits. The measurement will destroy the quantum state and will produce a single classical bit for each qubit in the circuit. This bitstring can be interpreted as a graph coloring of the input problem.

We can increase the depth of the QAOA circuit (length of the QAOA program) in order to achieve a better answer. A hyperparameter p sets the depth. If $p = 1$, the QAOA circuit is exactly as shown in Figure 3. If $p > 1$, we repeat the controlled phase operations and x-rotations p times (the H gates are not repeated). Each stage will use different values for the parameters γ and β. Hence, there will be a total of $2p$ classical parameters which determine the QAOA circuit, $\boldsymbol{\gamma} = \gamma_0 \gamma_1 ... \gamma_{p-1}$ and $\boldsymbol{\beta} = \beta_0 \beta_1 ... \beta_{p-1}$. A higher p will produce better results in absence of noise. However, as modern quantum computers are very noisy, a higher p can actually produce worse results, as the susceptibility to noise increases with the length of the program.

It should be noted that QAOA circuit will typically not produce the same output every time, even if we perform the exact same operations with the same parameters. Each time we run the quantum circuit and perform the measurement we're likely to get different bitstrings - the circuit is effectively creating a *distribution* of output bitstrings. Hence, we will need to run the circuit many times to get a good estimate on what the distribution is. We will be interested in the *expectation value* of the distribution, which is simply the average "score" of all the bitstrings in the distribution. As we
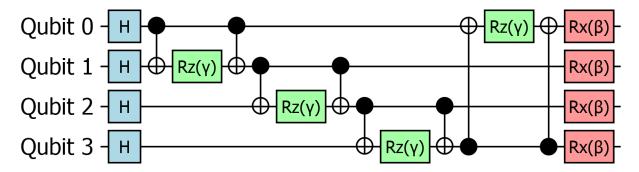
Figure 3: QAOA circuit with $p = 1$. H(adamard) gates are applied to all qubits. Then a sequence of controlled z-rotations is performed. The qubits which interact via these controlled rotations is determined by the input graph, shown in Figure 1a. A controlled-z rotation is performed for each edge in the graph. Finally, x-rotations are performed on all qubits. Classical parameters γ and β determine the rotation angles.

run QAOA, we will try to find values for $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ that produce the largest expectation value - covered in Section 4.

4   Simultaneous Perturbation Stochastic Approximation

Notation: I use a short-hand representation of vector operations. I will use bold for vectors and regular print for scalers. For example, if $\boldsymbol{x} = [1, 2, 3, 4, 5]$, a vector (element-wise) operation can be written as

$$\boldsymbol{y} = 1/\boldsymbol{x} = \left[\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5} \right] \tag{1}$$

Before discussing how we optimize the QAOA circuit to produce better results, it's useful to cover some background on the classical optimizer. The Simultaneous Perturbation Stochastic Approximation (SPSA) algorithm [10] is commonly used to optimize QAOA. SPSA finds widespread use in industry due to its many benefits. It's simple, has resilience to noise, and remains efficient even for very large optimization problems. It's a good choice whenever one does not have access to the inner workings of the function they are trying to optimize (we don't on quantum computers as they are obscured within the quantum mechanics). Say we have a function F that we'd like to optimize. F is parameterized by a vector of m parameters, $\boldsymbol{\Theta} = \theta_0 \theta_1 ... \theta_{m-1}$. We'd like to maximize $F(\boldsymbol{\Theta})$.

We start by randomly initializing $\boldsymbol{\Theta}$ by assigning a random number to each θ. We then generate a random perturbation vector $\boldsymbol{\Delta} = \delta_0 \delta_1 ... \delta_{m-1}$. Each δ has a magnitude of c, but can be positive or negative (what c is will be covered in a bit). So $\boldsymbol{\Delta}$ is a vector of Bernoulli random variables (which can be ± 1) multiplied by a constant c. From this perturbation vector we create two new versions of our parameter vector

$$\begin{aligned} \boldsymbol{\Theta}_+ = \boldsymbol{\Theta} + \boldsymbol{\Delta} &= [\theta_0 + \delta_0, \theta_1 + \delta_1, ..., \theta_{m-1} + \delta_{m-1}] \\ \boldsymbol{\Theta}_- = \boldsymbol{\Theta} - \boldsymbol{\Delta} &= [\theta_0 - \delta_0, \theta_1 - \delta_1, ..., \theta_{m-1} - \delta_{m-1}] \end{aligned} \tag{2}$$

We have one parameter vector with the perturbations added and one parameter vector with the perturbations subtracted. This will allow us to estimate the gradients ($\boldsymbol{g}$) of the perturbation vector by performing two evaluations of our objective function

$$\boldsymbol{g} = \frac{F(\boldsymbol{\Theta}_+) - F(\boldsymbol{\Theta}_-)}{2c\boldsymbol{\Delta}} \tag{3}$$

$F(\boldsymbol{\Theta}_+)$ and $F(\boldsymbol{\Theta}_-)$ are the outputs of the objective function with the two parameter configurations, and they are both scalers. c is also a scaler. $\boldsymbol{\Delta}$ is a vector of length m, so $\boldsymbol{g}$ is also a vector of length m.[4] If $F(\boldsymbol{\Theta}_+) > F(\boldsymbol{\Theta}_-)$, we'll move the parameters towards $\boldsymbol{\Theta}_+$. Otherwise, we'll move the parameters towards $\boldsymbol{\Theta}_-$. This is done with our parameter update function.

$$\boldsymbol{\Theta} = \boldsymbol{\Theta} + a \times \boldsymbol{g} \tag{4}$$

---

[4]Note that $\frac{F(\boldsymbol{\Theta}_+) - F(\boldsymbol{\Theta}_-)}{2c}$ is a scaler. So $\boldsymbol{g}$ is this scaler divided by the vector $\boldsymbol{\Delta}$, just like in Equation 1
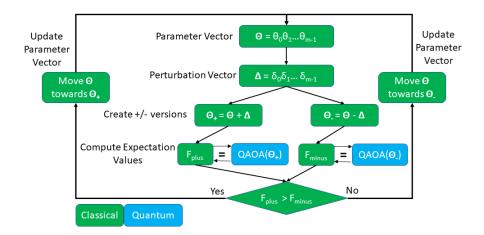
Figure 4: The process of using SPSA to train QAOA. Green blocks are performed on the classical computer, Blue blocks are performed on the quantum computer

We update the add a fraction of the gradients to each of our parameters. The parameter a (a scaler) determines how large this update is, and it is analogous to the learning rate in machine learning. The update of the parameter vector Θ concludes one optimization iteration. Many iterations will be required to find the optimal parameters. A new perturbation vector Δ will be generated for each iteration.

In the equations above, a and c are scalers which determine how much we perturb our parameters (c) and how much we update the parameters based on the gradients (a). We'll need to reduce a and c over time (over the optimization iterations) to allow the problem to converge to a final result. For optimization iteration i, we'll typically use something like

$$a = a_{start}/(i+1)^{decay}$$
$$c = c_{start}/(i+1)^{decay}$$

(5)

where a_{start} and c_{start} are the initial values, i is the index of the optimization iteration, and $decay$ is the rate at which the parameters decay.[5] Each of the parameters (their initial values and the decay rates) can be finely tuned, and the optimal values for each will depend on the specific optimization problem. If you want to be smart, one can use the gradient found on the the first optimization iteration to set a or c appropriately [5]. However, in the examples provided later, I simply tried multiple values until I found a few that worked well.

We can see where the name of SPSA comes from, we are doing a *simultaneous perturbation* of all our parameters. This is in contrast to gradient descent, where we would perturb each parameter individually. Doing it simultaneously means we only have to perform two evaluations of our objective function for each optimization iteration, regardless of the number of parameters.

4.1   SPSA applied to QAOA

When SPSA is used with QAOA, the function F produces the expectation value coming from the QAOA circuit and Θ is the γ and β parameters that set the rotation angles of the quantum gates. SPSA is running on the classical computer, choosing values for γ and β. It then sends the parameters to the quantum computer, which will run the QAOA circuit and generate the expectation value. Based on the resulting expectation values, SPSA will update γ and β. Hence, SPSA will be evaluating the performance of two slightly different QAOA circuits (which have slightly varying γ and β) and it will move the parameters towards which circuit performs better. This process is shown pictorially in Figure 4.

---

[5]More complex variations of these sequences exist [11], such as having different decay rates for each sequence. However, I will be sticking with this relatively simple variant which works sufficiently well.

5 High Level (Re)View

Before getting into the specifics of the implementation, it is useful to look at the situation from a high level to see how all the working parts interact.

The **input graph** represents the optimization problem we'd like to solve. The binary variables are represented by vertices and the constraints are represented by edges. We want to find graph colorings (bitstrings) which satisfy the problem optimally. We want vertices connected by edges to be different colors.

A **QAOA circuit** can be constructed deterministically based on the input graph. It specifies the quantum operations we need to perform. H gates are performed at the beginning. For each stage, controlled phase rotations are performed for each edge and single qubit x rotations are performed on all qubits. The rotation angles are determined by the parameters γ and β, which are unknown at the start. The number of stages is determined by the hyperparamter p, which can be set arbitrarily.

SPSA is going to search for optimal values of $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ which will enable QAOA to solve the problem. It (typically) starts by setting setting $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ randomly. Then it it will perturb them by some amount, creating two different versions. SPSA will call the QAOA circuit with both sets of parameters and compare the relative quality of both. It will then update the original parameters in the direction of which set performed better.

6 Specific Implementations

In this section I will provide a walk-through of code which implements SPSA and QAOA to solve a given max-cut problem, a python (Qiskit) version in Section 6.3 and an R (QuantumOps) version in Sections 6.4. The code is available for download from github [12].

6.1 Pre-Requisites

The R implementation requires the "QuantumOps" package [9] which is available on the CRAN repository. QuantumOps is just a library that will supply the matrices which represent the quantum operations that we need to perform. The python version requires IBM's Qiskit [2].

6.2 Inputs

The problem requires the following inputs

- **n**: The number vertices of input graph. Also is the length of the graph coloring bitstrings and the number of qubits required

- **edges**: The edges between the vertices representing the constraints. In R, this is a two-column matrix where source vertices are in column 1 and destination vertices are in column 2. In python this is a list of lists

- **p**: The depth of the QAOA circuit we want to use

- **nIterations**: The number of optimization iterations to run SPSA for. Each iteration consists of two calls to QAOA

- **nSamples**: The number of samples (measurements/bitstrings) to collect from the QAOA circuit for each optimization iteration. This corresponds to the number of times the QAOA circuit needs to be run.

- **a_start**: The starting value for a

- **c_start**: The starting value for c

- **decay**: The decay factor for a and c.

6.3   python implementation (Qiskit)

6.3.1   Computing Expectation Value

We need to be able to compute the score of a bitstring (graph coloring) - how well it solves the optimization problem. As described in Section 2.1, this involves checking each edge to see if it connects vertices (bits) of different colors (values). When simulating a quantum circuit, Qiskit returns a python dictionary where the keys are the measurement values and the values are the number of times that measurement was observed. Given this dictionary, **counts**, and the **edges** of the input graph we can find the expectation value of the distribution of outputs. We iterate over all the observed bitstrings, then check the bits (vertices) on both ends of all the edges. If the vertices (bits) are different colors (values), we increment the score. The average score of all the bitstrings is the expectation value.

```
#Compute all scores for a set of edges
def computeExpectationValue(counts,edges):
    totalScore = 0
    totalSamples = 0
    #For each bitstring measured (keys in counts dictionary)
    for bitstring in counts.keys():
        score = 0 #Score for this bitstring
        #For each edge
        for j in range(len(edges)):
            #If vertices (bits) on both ends of edge are different
            if( bitstring[edges[j][0]] != bitstring[edges[j][1]] ):
                score += 1 #Increment score
        totalScore += score * counts[bitstring] #Multiply score times the # of times it was observed
        totalSamples += counts[bitstring]    #Keep track of the number of measurements (samples)
    return(totalScore/totalSamples)
```

6.3.2   QAOA Circuit

We need to create the QAOA circuit that runs on the quantum computer. The circuit depends on both on the input graph and the classical parameters. The following function creates a quantum circuit as described in Section 3. It starts with Hadamards on all qubits. Then for p iterations, we apply controlled z-rotations for each edge of the input graph (**graph**) and then do x-rotations on all qubits. **betas** and **gammas** are the β and γ parameter vectors that will be supplied by the classical computer.

```
#Create quantum circuit for QAOA from edges and parmeters
def QAOA(nQubits,edges,p,betas,gammas):
    #Define quantum and classical registers
    qr = QuantumRegister(nQubits)
    cr = ClassicalRegister(nQubits)
    circuit = QuantumCircuit(qr,cr)

    #Initial Hadamards
    for q in range(nQubits):
        circuit.h(q)
    #For the number of specified iterations
    for P in range(p):
        #Controlled phase rotations
        #For each edge
        for j in range(len(edges)):
            #First CNOT from source qubit to destination qubit
            circuit.cx(edges[j][0],edges[j][1])
            #Rz on destination qubit
            circuit.rz(phi=gammas[P],qubit=edges[j][1])
            #Second CNOT from source qubit to destination qubit
            circuit.cx(edges[j][0],edges[j][1])
```

```
        #X rotations
        for q in range(nQubits):
            circuit.rx(theta=2*betas[P],qubit=q)
    circuit.measure(qr,cr)
    return circuit
```

6.3.3 Auxiliary Functions

Some additional functions keep the code clear and clean. The following function takes an input quantum circuit, calls the simulator, and returns the counts dictionary (measurement results). **shots** is the number of samples to take (repetitions and measurements of the quantum circuit).

```
#Run the circuit and return counts
def runCKT(circuit,shots=10000):
    simulator = Aer.get_backend('qasm_simulator')
    counts = execute(circuit,backend=simulator,shots=shots).result().get_counts(circuit)
    return counts
```

The following function takes an input circuit, gets the counts by calling *runCKT* and then finds the expectation value by calling *computeExpectationValue*.

```
#Run a circuit and get expectation value
def ExpectationValue(circuit,edges,nSamples=10000):
    #Run circuit and collect counts
    counts = runCKT(circuit=circuit,shots=nSamples)
    #Get the score of the counts
    score = computeExpectationValue(counts,edges)
    return(score)
```

This function conveniently produces a bernoulli random variable for the perturbation vectors

```
#Return +1 or -1 with equal probability
def m1p1():
    return random.randrange(2)*2 - 1
```

6.3.4 SPSA

We now have all the parts to run SPSA to optimize QAOA. The following funtions searches for optimal values of γ and β. It starts by initializing the a and c sequences we require, which decay over time. It then initializes γ and β randomly, but near zero. It then iterates through the optimization process many times. Each iteration does the following

1. Set perturbation vectors **Delta_gammas** and **Delta_betas**, whose values are all $\pm c$

2. Creates two versions of the QAOA circuit, one with the perturbations added and one with them subtracted

3. Finds the expectation value for each circuit

4. Computes the gradients **g_gammas** and **g_betas** from the expectation values and the perturbations

5. Updates the parameters depending on the gradients and learning rate a

```
def SPSAforQAOA(n,edges,p,nIterations,nSamples,a_start,c_start,decay):
    #Initiate
    a = []; c = [];
    for i in range(1,nIterations+1):
```

```python
        a.append( a_start / (i ** decay) )
        c.append( c_start / (i ** decay) )
    for i in range(nIterations):
        if( c[i] < 0.01 ):
            c[i] = 0.01


    #Initiate gamma,beta
    gammas = []
    betas = []
    for P in range(p):
        gammas.append( random.uniform(-.1,.1) )
        betas.append( random.uniform(-.1,.1) )


    #Run iterations of SPSA
    for i in range(nIterations):
        #Randomly perturb gammas,betas by c[i]
        Delta_gammas = []; gammas_plus = []; gammas_minus = [];
        Delta_betas = []; betas_plus = []; betas_minus = [];
        for P in range(p):
            #Generate perturbation vectors of bernoulli variables with magnitude c[i]
            Delta_gammas.append( m1p1() * c[i] )
            Delta_betas.append( m1p1() * c[i] )
            #Create +/- versions of the parameters
            gammas_plus.append( gammas[P] + Delta_gammas[P])
            gammas_minus.append(gammas[P] - Delta_gammas[P])
            betas_plus.append( betas[P] + Delta_betas[P])
            betas_minus.append(betas[P] - Delta_betas[P])

        #Get the circuits for the +/- versions
        pCircuit = QAOA(nQubits=n,edges=edges,p=p,betas=betas_plus,gammas=gammas_plus)
        mCircuit = QAOA(nQubits=n,edges=edges,p=p,betas=betas_minus,gammas=gammas_minus)
        #Run the +/- versions
        Fplus = ExpectationValue(circuit=pCircuit,edges=edges,nSamples=nSamples)
        Fminus = ExpectationValue(circuit=mCircuit,edges=edges,nSamples=nSamples)

        #Compute estimated gradients
        g_gammas = []; g_betas = [];
        for P in range(p):
            g_gammas.append( (Fplus - Fminus) / (2*Delta_gammas[P]) )
            g_betas.append( (Fplus - Fminus) / (2*Delta_betas[P]) )

        #Update the parameters
        for P in range(p):
            gammas[P] = gammas[P] + a[i]*g_gammas[P]
            betas[P] = betas[P] + a[i]*g_betas[P]

        #Report progress
        print('Iteration:',i,'Exp(+):',Fplus,'Exp(-):',Fminus)
```

The code can be run out of the box with the given example

```python
def example():
    #4 qubits
    n = 4
    #Edges of the maxcut problem
    edges = [ [0,1] , [1,2] , [2,3] , [3,0] ]
    #p=2 is sufficient for this problem
    p = 2
```

```
#A sufficient number of optimization iterations to solve problem
nIterations = 100
#Typically need quite a few samples (measurements of quantum circuit) per iteration to
nSamples = 10000
#Heuristically chosen a and c
a_start = 0.25
c_start = 0.25
decay = 0.5
SPSAforQAOA(n=n,edges=edges,p=p,nIterations=nIterations,nSamples=nSamples,
a_start=a_start,c_start=c_start,decay=decay)
```

6.4 R implementation (QuantumOps)

6.4.1 Computing Score

As mentioned, computing the score (the number of constraints satisfied by a bitstring) is a classically efficient process. We need a function to perform this action, so we can evaluate the quality of the bitstrings coming out of QAOA. In addition to the bitstring to evaluate, we need to know the edges of the graph to compute how many of the edges were cut by the bitstring (graph coloring). We can compute the score by checking each edge in the graph and seeing whether it connects vertices (bits) of opposite colors (values). If the edge does connect vertices of opposite colors, we increment the score by 1. Note that the edges are represented as a 2-column matrix, where rows are edges and the two columns are the source and destination vertices. dim(edges)[1] is the number of rows in the matrix.

```
#classically check the score (cut) of a bitstring
computeScore <- function(bitstring,edges){
    score <- 0
    for(e in 1:dim(edges)[1]){ #For each edge
        #If bits on either end of the edge are different
        if(bitstring[edges[e,1]+1] != bitstring[edges[e,2]+1]) #R indices start at 1, offset index by +1
            score <- score + 1                          #Score increases by 1
    }
    score #return score
}
```

6.4.2 Quantum Circuit

We need to create a quantum circuit for QAOA which depends both on the input problem and the classical parameters. In QuantumOps quantum circuits are represented by matrices. We build the matrix by going through the quantum circuit and adding operations as we go. Say we are creating a matrix m which representations two consecutive operations $op1$ and $op2$, also represented by matrices. We can construct m with

```
m <- op1
m <- op2 %*% m
```

%*% is matrix multiplication in R. The following function creates a QAOA circuit. It uses QuantumOps functions *many*, *controlled*, and *single*. *many* creates a tensor product of many identical operations, used to create the idential H and Rx gates on all qubits. *controlled* creates controlled versions of gates, a controlled-x gate is a CNOT gate. *single* creates a single gate which is in a tensor product with many *Idle* gates, the relevant gate is applied to a single qubit and the rest of the qubits are idle.

```
#Create QAOA circuit (represented by a matrix) from edges and parameters
QAOA <- function(nQubits,edges,p,gammas,betas){
  #Initial Hadamard gates
  m <- many(gate=H(),n=nQubits)
  #For each stage
```

```
  for(P in 1:p){
    #Phase separation for each edge
    for(e in 1:dim(edges)[1]){
      #CNOT from source to destination
      m <- controlled(gate=X(),n=nQubits,cQubits=edges[e,1],tQubit=edges[e,2]) %*% m
      #Rz on destination
      m <- single(gate=Rz(gammas[P]),n=nQubits,t=edges[e,2]) %*% m
      #CNOT from source to destination
      m <- controlled(gate=X(),n=nQubits,cQubits=edges[e,1],tQubit=edges[e,2]) %*% m
    }
    #Rx on all qubits
    m <- many(gate=Rx(2*betas[P]),n=nQubits) %*% m
  }
  m #Return the matrix
}
```

6.4.3 Expectation Value

We are interested in the expectation value produced by the QAOA circuit for a given set of γ and β. In order to get an accurate estimate of the expectation value we'll need to run the circuit many times. The following function takes in the relevant parameters and reports back the expectation value. It starts by constructing the required quantum circuit, with a call to *QAOA*. Then it creates the $|00...0\rangle$ quantum state with as many qubits as required with a call to *intket*. The circuit is then applied to the quantum state. In experiment, the quantum state would then be destroyed by a measurement and we would have to re-perform the circuit many times to get all the measurements. With simulation we can cheat and directly look at the measurement probabilities, which is provided by calling *probs* on the quantum state. We can then sample integers from that distribution, emulating many repeated measurements on the quantum state. We then need to evaluate each of the measurements (bistrings) on how well they solve the optimization problem. We return the average score, the expectation value.

```
ExpectationValue <- function(n,edges,p,gammas,betas,nSamples){
    #Generate the QAOA quantum circuit
    circuit <- QAOA(nQubits=n,edges=edges,p=p,gammas=gammas,betas=betas)
    #Initialze a qubit register in the |00...0> state
    qr <- intket(x=0,n=n)
    #Apply the circuit (represented by a matrix) to the quantum register (represented by a vector)
    qr <- circuit %*% qr
    #Simulate the measurements of the quantum state
    #each measurement would correspond to individual run in experiment
    #Sample from 0 to n-1 with probabilities from the quantum state
    measurements <- sample(x=0:(2^n-1),size=nSamples,prob=probs(qr),replace=TRUE)

    #Test the score of the measurements
    scoreSum <- 0    #Add up all the scores from all measurments
    for(i in 1:nSamples) #convert integer result to binary bitstring before parsing
        scoreSum <- scoreSum + computeScore(bitstring=convert_dec2bin(measurements[i],len=n),edges=edges)
    expectationValue <- scoreSum/nSamples #Average them

    return(expectationValue)
}
```

6.4.4 SPSA

All the parts now exist in order to effectively use SPSA to optimize QAOA. The following function searches for optimal values of γ and β. It starts by initializing the a and c sequences we require, which decay over time. It then initializes γ and β randomly. It then iterates through the optimization process many times which consists of

1. Set perturbation vectors **dGammas** and **dBetas**, whose values are all $\pm c$

2. Finds the expectation value for the two versions of QAOA, one with the perturbations added and one with them subtracted

3. Computes the gradients **gGammas** and **gBetas** from the expectation values and the perturbations

4. Updates the parameters depending on the gradients and learning rate a

Note that R conveniently does element-wise multiplation/addition when two vectors are supplied as arguments

```
#Run Experiment
SPSAforQAOA <- function(n,edges,p,nIterations,nSamples,a_start,c_start,decay=2){
    #Make learning rate and perturbation sequences
    a <- a_start / (1:nIterations)^decay
    c <- c_start / (1:nIterations)^decay
    c[c<0.01] <- 0.01 #put a lower bound on c (dont make perturbations too small)
    #Generate initial parameters at random
    gammas <- runif(min=-0.05,max=0.05,n=p)
    betas  <- runif(min=-0.05,max=0.05,n=p)

    #Run iterations of SPSA
    for(i in 1:nIterations){
        #Generate perturbation vectors (bournoulli random variables of magnitude c[i])
        dGammas <- sample(x=c(-1,1),size=p,replace=TRUE)*c[i]
        dBetas  <- sample(x=c(-1,1),size=p,replace=TRUE)*c[i]

        #Compute the expectation values of the + and - configurations
        Fplus <-
            ExpectationValue(n=n,edges=edges,p=p,gammas=gammas+dGammas,betas=betas+dBetas,nSamples=nSamples)
        Fminus <-
            ExpectationValue(n=n,edges=edges,p=p,gammas=gammas-dGammas,betas=betas-dBetas,nSamples=nSamples)

        #Compute the gradients of the parameters based on expectation values
        gGammas <- (Fplus - Fminus) / (2*dGammas)
        gBetas  <- (Fplus - Fminus) / (2*dBetas)

        #Update the parameters based on gradients
        gammas <- gammas + a[i] * gGammas
        betas  <- betas + a[i] * gBetas

        #Report the progress
        print(paste('Iteration:',i,'Exp(+):',sprintf('%.4f',Fplus),'Exp(-):',sprintf('%.4f',Fminus)))
    }
}
```

The code can be run out of the box with the following example

```
#The provided example problem
example <- function(){
    #4 qubits
    n <- 4
    #Edges of the max-cut problem in matrix from (rows are edges, column 1 is source vertex, column 2 is
        destination vertex)
    edges <- matrix(c( 0,1, 1,2 , 2,3 , 0,3),byrow=TRUE,ncol=2)
    #p=2 is sufficient for this problem
    p <- 2
    #A sufficient number of optimization iterations to solve problem
    nIterations <- 100
```

```
    #Typically need quite a few samples (measurements of quantum circuit) per iteration to
    nSamples <- 10000
    #Heuristically chosen a and c
    a_start <- 0.25
    c_start <- 0.25
    decay <- 0.5
    SPSAforQAOA(n=n,edges=edges,p=p,nIteration=nIterations,nSamples=nSamples,a_start=a_start,c_start=c_start,decay=deca
}
```

A   Complete code for python (Qiskit)

Also available at [12]

```python
from qiskit import *
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit import execute
from qiskit.providers.aer import QasmSimulator

import random

#Create quantum circuit for QAOA from edges and parmeters
def QAOA(nQubits,edges,p,betas,gammas):
    #Define quantum and classical registers
    qr = QuantumRegister(nQubits)
    cr = ClassicalRegister(nQubits)
    circuit = QuantumCircuit(qr,cr)

    #Initial Hadamards
    for q in range(nQubits):
        circuit.h(q)
    #For the number of specified iterations
    for P in range(p):
        #Phase 1
        #For each edge
        for j in range(len(edges)):
            #First CNOT
            circuit.cx(edges[j][0],edges[j][1])
            #Rz on target qubit
            circuit.rz(phi=gammas[P],qubit=edges[j][1])
            #Second CNOT
            circuit.cx(edges[j][0],edges[j][1])

        for q in range(nQubits):
            circuit.rx(theta=2*betas[P],qubit=q)
    circuit.measure(qr,cr)
    return circuit

#Return +1 or -1 with equal probability
def m1p1():
    return random.randrange(2)*2 - 1

#Compute all scores for a set of edges
def computeExpectationValue(counts,edges):
    totalScore = 0
    totalSamples = 0
    #For each bitstring measured (keys in counts dictionary)
```

```python
    for bitstring in counts.keys():
        score = 0 #Score for this bitstring
        for j in range(len(edges)):
            if( bitstring[edges[j][0]] != bitstring[edges[j][1]] ):
                score = score + 1
        totalScore += score * counts[bitstring] #Multiply score times the # of times it was observed
        totalSamples += counts[bitstring]    #Keep track of the number of measurements (samples)
    return(totalScore/totalSamples)

#Run the circuit and return counts
def runCKT(circuit,shots=10000,noise=True):
    simulator = Aer.get_backend('qasm_simulator')
    counts = execute(circuit,backend=simulator,shots=shots).result().get_counts(circuit)
    return counts

#Run a circuit and get expectation value
def ExpectationValue(circuit,edges,nSamples=10000):
    #Run circuit and collect counts
    counts = runCKT(circuit=circuit,shots=nSamples)
    #Get the score of the counts
    score = computeExpectationValue(counts,edges)
    return(score)

def SPSAforQAOA(n,edges,p,nIterations,nSamples,a_start,c_start,decay):
    #Initiate
    a = []; c = [];
    for i in range(1,nIterations+1):
        a.append( a_start / (i ** decay) )
        c.append( c_start / (i ** decay) )
    for i in range(nIterations):
        if( c[i] < 0.01 ):
            c[i] = 0.01

    #Initiate gamma,beta
    gammas = []
    betas = []
    for P in range(p):
        gammas.append( random.uniform(-.1,.1) )
        betas.append( random.uniform(-.1,.1) )

    #Run iterations of SPSA
    for i in range(nIterations):
        #Randomly perturb gammas,betas by c[i]
        Delta_gammas = []; gammas_plus = []; gammas_minus = [];
        Delta_betas = []; betas_plus = []; betas_minus = [];
        for P in range(p):
            #Generate perturbation vectors of bernoulli variables with magnitude c[i]
            Delta_gammas.append( m1p1() * c[i] )
            Delta_betas.append( m1p1() * c[i] )
            #Create +/- versions of the parameters
            gammas_plus.append( gammas[P] + Delta_gammas[P])
            gammas_minus.append(gammas[P] - Delta_gammas[P])
            betas_plus.append( betas[P] + Delta_betas[P])
            betas_minus.append(betas[P] - Delta_betas[P])

        #Get the circuits for the +/- versions
        pCircuit = QAOA(nQubits=n,edges=edges,p=p,betas=betas_plus,gammas=gammas_plus)
        mCircuit = QAOA(nQubits=n,edges=edges,p=p,betas=betas_minus,gammas=gammas_minus)
```

```python
        #Run the +/- versions
        Fplus = ExpectationValue(circuit=pCircuit,edges=edges,nSamples=nSamples)
        Fminus = ExpectationValue(circuit=mCircuit,edges=edges,nSamples=nSamples)

        #Compute estimated gradients
        g_gammas = []; g_betas = [];
        for P in range(p):
            g_gammas.append( (Fplus - Fminus) / (2*Delta_gammas[P]) )
            g_betas.append( (Fplus - Fminus) / (2*Delta_betas[P]) )

        #Update the parameters
        for P in range(p):
            gammas[P] = gammas[P] + a[i]*g_gammas[P]
            betas[P] = betas[P] + a[i]*g_betas[P]

        #Report progress
        print('Iteration:',i,'Exp(+):',Fplus,'Exp(-):',Fminus)

def example():
    #4 qubits
    n = 4
    #Edges of the maxcut problem
    edges = [ [0,1] , [1,2] , [2,3] , [3,0] ]
    #p=2 is sufficient for this problem
    p = 2
    #A sufficient number of optimization iterations to solve problem
    nIterations = 100
    #Typically need quite a few samples (measurements of quantum circuit) per iteration to
    nSamples = 10000
    #Heuristically chosen a and c
    a_start = 0.25
    c_start = 0.25
    decay = 0.5
    SPSAforQAOA(n=n,edges=edges,p=p,nIterations=nIterations,nSamples=nSamples,
    a_start=a_start,c_start=c_start,decay=decay)
```

B   Complete code for R (QuantumOps)

Also available at [12]

```r
library("QuantumOps")

#Create QAOA circuit (represented by a matrix) from edges and parameters
QAOA <- function(nQubits,edges,p,gammas,betas){
    #Initial Hadamard gates
    m <- many(gate=H(),n=nQubits)
    #For each stage
    for(P in 1:p){
        #Phase separation for each edge
        for(e in 1:dim(edges)[1]){
            m <- controlled(gate=X(),n=nQubits,cQubits=edges[e,1],tQubit=edges[e,2]) %*% m #CNOT from
                control to target
            m <- single(gate=Rz(gammas[P]),n=nQubits,t=edges[e,2]) %*% m              #Rz on target
            m <- controlled(gate=X(),n=nQubits,cQubits=edges[e,1],tQubit=edges[e,2]) %*% m #CNOT from
                control to target
        }
```

```r
        #Rx on all qubits
        m <- many(gate=Rx(2*betas[P]),n=nQubits) %*% m
    }
    m #Return the matrix
}


#classically check the score (cut) of a bitstring
computeScore <- function(bitstring,edges){
    score <- 0
    for(e in 1:dim(edges)[1]){ #For each edge (R indices start at 1)
        if(bitstring[edges[e,1]+1] != bitstring[edges[e,2]+1]) #If bits on either end of the edge are
            different
            score <- score + 1                    #Score increases by 1
    }
    score #return score
}


#Find the expectation value of QAOA for a given set of parameters
ExpectationValue <- function(n,edges,p,gammas,betas,nSamples){
    #Generate the QAOA quantum circuit
    circuit <- QAOA(nQubits=n,edges=edges,p=p,gammas=gammas,betas=betas)
    #Initialze a qubit register in the |00...0> state
    qr <- intket(x=0,n=n)
    #Apply the circuit (represented by a matrix) to the quantum register (represented by a vector)
    qr <- circuit %*% qr
    #Simulate the measurements of the quantum state (each measurement would correspond to individual run
        in experiment)
    measurements <- sample(x=0:(2^n-1),size=nSamples,prob=probs(qr),replace=TRUE) #Sample from 0 to n-1
        with probabilities from the quantum state

    #Test the score of the measurements
    scoreSum <- 0    #Add up all the scores from all measurments
    for(i in 1:nSamples) #convert integer result to binary bitstring before parsing
        scoreSum <- scoreSum + computeScore(bitstring=convert_dec2bin(measurements[i],len=n),edges=edges)
    expectationValue <- scoreSum/nSamples #Average them

    return(expectationValue)
}


#Run Experiment
SPSAforQAOA <- function(n,edges,p,nIterations,nSamples,a_start,c_start,decay=2){
    #Make learning rate and perturbation sequences
    a <- a_start / (1:nIterations)^decay
    c <- c_start / (1:nIterations)^decay
    c[c<0.01] <- 0.01 #put a lower bound on c (dont make perturbations too small)
    #Generate initial parameters at random
    gammas <- runif(min=-0.05,max=0.05,n=p)
    betas  <- runif(min=-0.05,max=0.05,n=p)

    #Run iterations of SPSA
    for(i in 1:nIterations){
        #Generate perturbation vectors (bournoulli random variables of magnitude c[i])
        dGammas <- sample(x=c(-1,1),size=p,replace=TRUE)*c[i]
        dBetas  <- sample(x=c(-1,1),size=p,replace=TRUE)*c[i]

        #Compute the expectation values of the + and - configurations
        Fplus <-
            ExpectationValue(n=n,edges=edges,p=p,gammas=gammas+dGammas,betas=betas+dBetas,nSamples=nSamples)
```

```r
        Fminus <-
            ExpectationValue(n=n,edges=edges,p=p,gammas=gammas-dGammas,betas=betas-dBetas,nSamples=nSamples)

        #Compute the gradients of the parameters based on expectation values
        gGammas <- (Fplus - Fminus) / (2*dGammas)
        gBetas <- (Fplus - Fminus) / (2*dBetas)

        #Update the parameters based on gradients
        gammas <- gammas + a[i] * gGammas
        betas <- betas + a[i] * gBetas

        #Report the progress
        print(paste('Iteration:',i,'Exp(+):',sprintf('%.4f',Fplus),'Exp(-):',sprintf('%.4f',Fminus)))
    }
}

#The provided example problem
example <- function(){
    #4 qubits
    n <- 4
    #Edges of the max-cut problem in matrix from (rows are edges, column 1 is source vertex, column 2 is
        destination vertex)
    edges <- matrix(c( 0,1, 1,2 , 2,3 , 0,3),byrow=TRUE,ncol=2)
    #p=2 is sufficient for this problem
    p <- 2
    #A sufficient number of optimization iterations to solve problem
    nIterations <- 100
    #Typically need quite a few samples (measurements of quantum circuit) per iteration to
    nSamples <- 10000
    #Heuristically chosen a and c
    a_start <- 0.25
    c_start <- 0.25
    decay <- 0.5
    SPSAforQAOA(n=n,edges=edges,p=p,nIteration=nIterations,nSamples=nSamples,a_start=a_start,c_start=c_start,decay=deca
}
```

References

[1] ALAM, M., ASH-SAKI, A., AND GHOSH, S. Circuit compilation methodologies for quantum approximate optimization algorithm. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2020), IEEE, pp. 215–228.

[2] CROSS, A. The ibm q experience and qiskit open-source quantum computing software. In *APS March Meeting Abstracts* (2018), vol. 2018, pp. L58–003.

[3] FARHI, E., GOLDSTONE, J., AND GUTMANN, S. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028* (2014).

[4] GUERRESCHI, G. G., AND MATSUURA, A. Y. Qaoa for max-cut requires hundreds of qubits for quantum speed-up. *Scientific reports 9*, 1 (2019), 1–7.

[5] KANDALA, A., MEZZACAPO, A., TEMME, K., TAKITA, M., BRINK, M., CHOW, J. M., AND GAMBETTA, J. M. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature 549*, 7671 (2017), 242–246.

[6] LOCEFF, M. A course in quantum computing (for the community college). *Foothill College* (2015).

[7] https://pennylane.ai/qml/demos/tutorial_qaoa_maxcut.html, 2019. Accessed: 2020-9-26.

[8] http://www.cs.cornell.edu/courses/cs4820/2014sp/notes/reduction-maxcut.pdf, 2014. Accessed: 2021-5-31.

[9] RESCH, S. *QuantumOps: Performs Common Linear Algebra Operations Used in Quantum Computing and Implements Quantum Algorithms*, 2020.

[10] SPALL, J. C., ET AL. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE transactions on automatic control 37*, 3 (1992), 332–341.

[11] https://www.jhuapl.edu/spsa/index.html, 2021. Accessed: 2021-6-1.

[12] https://github.com/SalonikResch/SPSAforQAOA, 2021.