quantumaudio

Release 0.0.2

Paulo Vitor Itaboraí

Jan 07, 2023

CONTENTS:

This page contains the API reference of the quantumaudio module.

Version: 0.0.2

CHAPTER

ONE

QUANTUMAUDIO

1.1 quantumaudio package

quantumaudio: A Python class implementation for Quantum Representations of Audio in Qiskit. Developed by the quantum computer music team at the Interdisciplinary Centre for Computer Music Research, University of Plymouth, UK

1.1.1 quantumaudio module contents

class EncodingScheme

Bases: object

get_encoder(*encoder_name: str*) → AnyEncoder

Returns: encoder class associated with name.

class QPAM

Bases: object

convert(*original_audio: ndarray[Any, dtype[ScalarType]]*) → ndarray[Any, dtype[ScalarType]]

Converts the digital audio into an array of probability amplitudes.

The audio signal is normalized. The normalized samples can then be interpreted as probability amplitudes. In other words, by squaring every sample, their total sum is now 1.

Parameters
original_audio – Numpy Array containing audio information.

Returns
A Numpy Array containing normalized probability amplitudes.

measure(*qc: QuantumCircuit*, *treg_pos: int = 0*) → None

Appends Measurements to a QPAM audio circuit

From a quantum circuit with a register containing a QPAM representation of quantum audio, creates a classical register with compatible size and adds isntructions for measuring the QPAM register.

Parameters

- **qc** – A qiskit quantum circuit containing at least 1 quantum register.
- **treg_pos** – Index of the QPAM ('treg') register in the circuit. Default is 0

prepare(*audio_amplitudes: ndarray[Any, dtype[ScalarType]]*, *regsize: Tuple[int, int]*, *regnames: Tuple[str, str]*, *print_state: bool = False*) → QuantumCircuit

Prepares a QPAM quantum circuit.

Creates a qiskit QuantumCircuit that prepares a Quantum Audio state using QPAM (Quantum Probability Amplitude Modulation) representation. The quantum circuits used for audio representations typically contain two qubit registers, namely, 'treg' (which encodes time/index information) and 'areg' (which encodes amplitude information).

Note: In QPAM, the 'areg' (amplitude) register is NOT used as the amplitude information is encoded in the probability amplitudes of the 'treg' (time) register.

Parameters

- **audio_amplitudes** – Array with propbability amplitudes
- **regsize** – The size of both qubit registers in a tuple (treg_size, areg_size). 'treg_size' qubits for 'treg'; 'areg_size' qubits for 'areg'. For QPAM, 'areg_size' is ALWAYS 0
- **regnames** – Label names for 'treg' and 'areg', passed as a tuple. For visualization purposes only.
- **print_state** – Toggles a simple print of the prepared quantum state to the console, for visualization purposes only.

Returns

A qiskit QuantumCircuit with specific QPAM preparation instructions.

reconstruct(*treg_size: int*, *counts: Counts*, *shots: int*, *g: float | None = None*) → ndarray[Any, dtype[ScalarType]]

Builds a digital Audio from qiskit histogram data.

Considering the QPAM encoding scheme, it uses the histogram data stored in a Counts object (qiskit.result.Counts) to reconstruct an audio signal. It renormalizes the histogram counts and remaps the signal back to the [-1 to 1] range.

Parameters

- **treg_size** – Size of the 'treg' (time) register.
- **counts** – Histogram from a qiskit job result (result.get_counts())
- **shots** – Amount of identical experiments ran by the qiskit job.
- **g** – Gain factor. This is a renormalization factor. (When bypassing audio signals through quantum circuits, this factor is usually proportional to the origal audio's norm).

Returns

A Digital Audio as a Numpy Array. The signal is in float format.

class QSM

Bases: `object`

convert(*original_audio*)

For the QSM encoding scheme, this function is dummy.

QSM expects a quantized signal (N-Bit PCM) as input. No pre-processing is needed after this point.

measure(*qc: QuantumCircuit*, *treg_pos: int = 1*, *areg_pos: int = 0*) → None

Appends Measurements to a QSM audio circuit

From a quantum circuit with registers containing a QSM representation of quantum audio, creates two classical registers with compatible sizes and adds instructions for measuring them.

Parameters

- **qc** – A quantum circuit containing at least 2 quantum registers.
- **treg_pos** – Index of the SQPAM ('treg') register in the circuit. Default is 1
- **areg_pos** – Index of the SQPAM ('areg') register in the circuit. Default is 0

omega_t(*qa: QuantumCircuit*, *t: int*, *a: int*, *treg: QuantumRegister*, *areg: QuantumRegister*, *print_state: bool = False*) → None

QSM Value-Setting operation.

Applies a multi-controlled CNOT gate to qubits of amplitude register, controlled by the time register at the respective time index state. In other words. At index 't', it flipps the amplitude qubits to match the original audio sample bits at the same index.

Parameters

- **qa** – The quantum circuit to be manipulated.
- **t** – Time index that will be encoded.
- **a** – Quantized sample from original audio to be converted to binary.
- **treg** – Time register, 'treg'.
- **areg** – Amplitude Register, 'areg'.
- **print_state** – Toggles a simple print of the prepared quantum state to the console, for visualization purposes only. To be used together with all other SQPAM methods with a 'print_state' kwarg.

prepare(*quantized_audio: ndarray[Any, dtype[ScalarType]]*, *regsize: Tuple[int, int]*, *regnames: Tuple[str, str]*, *print_state: bool = False*) → QuantumCircuit

Prepares a QSM quantum circuit.

Creates a qiskit QuantumCircuit that prepares a Quantum Audio state using QSM (Quantum State Modulation). The quantum circuits used for audio representations typically contain two qubit registers, namely, 'treg' (which encodes time/index information) and 'areg' (which encodes amplitude information).

Parameters

- **quantized_audio** – Integer Array with the input signal.
- **regsize** – The size of both qubit registers in a tuple (treg_size, areg_size). 'treg_size' qubits for 'treg'; 'areg_size' qubits for 'areg'.
- **regnames** – Label names for 'treg' and 'areg', passed as a tuple. For visualization purposes only.
- **print_state** – Toggles a simple print of the prepared quantum state to the console, for visualization purposes only.

Returns

A qiskit quantum circuit containing specific QSM preparation instructions.

reconstruct(*treg_size: int*, *counts: Counts*) → ndarray[Any, dtype[ScalarType]]

Builds a digital Audio from qiskit histogram data.

Considering the QSM encoding scheme, it uses the histogram data stored in a Counts object (qiskit.result.counts.Counts) to reconstruct an audio signal. It uses the bin labels of the histogram, which contains the measured quantum states in binary form. It converts the binary pairs to (amplitude, index) pairs, building an Array.

Parameters

- **treg_size** – Size of the 'treg' (time) register.
- **counts** – Histogram from a qiskit job result (result.get_counts())

Returns
A Digital Audio as a Numpy Array. The signal is in quantized (int) format.

treg_index_X(*qc: QuantumCircuit*, *t: int*, *treg: QuantumRegister*, *print_state: bool = False*) → None

Auxilary function for matching control conditions with time indexes.

Applies X gates on qubits of the time register whenever the respective bit of the current time index (in binary representation) is 0. As a result, the qubit will be flipped to |1> and succesfully trigger necessary control conditions of the circuit for this time index.

Parameters

- **qa** – The quantum circuit to be maniputated
- **t** – Time index that will be converted to binary form for comparison.
- **treg** – Quantum register of the time indexes.
- **print_state** – Toggles a simple print of the prepared quantum state to the console, for visualization purposes only. To be used together with all other QSM methods with a 'print_state' kwarg.

Examples

treg_index_X(qa, 6, treg) (a time register 'treg' with, say, 5 qubits in 'qa' at instant 6)

't' = 6 == '00110'. *treg_index_X()* applies X gates to qubits 0, 3 and 4
(right to left) of register 'treg'.

class QuantumAudio(*encoder_name: str*)

Bases: object

listen(*rate: int = 44100*) → None

Plays the audio file using ipython.display.Audio()

load_input(*input_audio: ndarray[Any, dtype[floating]]*, *bit_depth: int = 1*) → *QuantumAudio*

Loads an audio file and calculates the qubit requirements.

Brings a digital audio signal inside the class for further processing. Audio files should be in numpy.ndarray type and be in the (-1. to 1.) amplitude range. You can also optionally load a quantized audio signal as input (-N to N-1) range, as long as you specify the bit depth of your quantized input 'areg_size'

Parameters

- **input_audio** – The audio signal to be converted. If not in 32-bit or 64-bit float format ('n'-bit integer PCM), specify bit depth.
- **bit_depth** – Audio bit depth IF using integer PCM. Ignore otherwise.

Returns
Returns itself for using multiple QuantumAudio methods in one line of code.

Examples

```
>>> float_audio = [0., -0.25, 0.5 , 0.75,  -0.75  ,  -1.,  0.25]
>>> quantum_audio = qa.QuantumAudio('qpam').load_input(float_audio)
For this input, the QPAM representation will require:
        3 qubits for encoding time information and
        0 qubits for encoding amplitude information.
```

```
>>> int_3bit_PCM_audio = [0, -1, 2, 3, -3, -4, 1]
>>> quantum_audio = qa.QuantumAudio('qsm').load_input(int_3bit_PCM_audio, 3)
For this input, the QSM representation will require:
        3 qubits for encoding time information and
        3 qubits for encoding amplitude information.
```

measure(*treg_pos: int | None = None*, *areg_pos: int | None = None*) → *QuantumAudio*

Updates quantum circuit by adding measurements in the end.

Will add a measurement instruction to the end of each qubit register.

Returns
Returns itself for using multiple QuantumAudio methods in one line of code.

plot_audio() → None

Plots comparisons between the input and output audio files.

Uses matplotlib.

prepare(*tregname: str = 't'*, *aregname: str = 'a'*, *print_state: bool = False*) → *QuantumAudio*

Creates a Quantum Circuit that prepares the audio representation.

Loads the 'circuit' attribute with the preparation circuit, according to the encoding technique used: QPAM, SQPAM or QSM.

Returns
Returns itself for using multiple QuantumAudio methods in one line of code.

reconstruct_audio(***additional_kwargs: Any*) → *QuantumAudio*

Builds an audio signal from a qiskit result histogram.

Depending on the chosen encoding technique, reconstructs an audio file using the histogram in QuantumAudio.counts (qiskit.result.Counts)

Returns
Returns itself for using multiple QuantumAudio methods in one line of code.

run(*shots: int = 10*, *backend_name: str = 'aer_simulator'*, *provider=<qiskit_aer.aerprovider.AerProvider object>*) → *QuantumAudio*

Runs the Quantum Circuit in an IBMQ job.

Transpiles and runs QuantumAudio.circuit in a qiskit job. Supports IBMQ remote backends.

Returns
Returns itself for using multiple QuantumAudio methods in one line of code.

class SQPAM

Bases: object

convert(*original_audio: ndarray[Any, dtype[ScalarType]]*) → ndarray[Any, dtype[ScalarType]]

Converts digital audio into an array of probability amplitudes.

The audio signal is mapped to an array of angles. The angles can then be interpreted as real-valued parameters for a trigonometric representation subspace of a qubit. In other words, the angles are used to rotate a qubit - originally in the |0> state - to the following state: (cos(angle)|0> + sin(angle)|1>). Notice that this preserves probabilities, as cos^2 + sin^2 = 1.

Note: By convention, we are using the *np.arcsin* function to calculate the angles. This means that the *SQPAM.reconstruct()* method will use the even (sine) bins of the histogram to retrieve the signal.

Parameters
original_audio – Numpy Array containing audio information.

Returns
A Numpy Array containing angles between 0 and pi/2.

mc_Ry_2theta_t(*qa: QuantumCircuit*, *t: int*, *a: float*, *treg: QuantumRegister*, *areg: QuantumRegister*, *print_state: bool = False*) → None

SQPAM Value-Setting operation.

Applies a controlled Ry(2*theta) gate to the amplitude register, controlled by the time register at the respective time index state. In other words. At index 't', it rotates the aplitude qubit by the angle mapped from the audio sample at the same index. In quantum computing terms, this translates to a multi-controlled rotation gate.

Parameters

- **qa** – The quantum circuit to be manipulated.
- **t** – Time index that will be encoded.
- **a** – Angle of rotation.
- **treg** – Time register, 'treg'.
- **areg** – Amplitude Register, 'areg'.
- **print_state** – Toggles a simple print of the prepared quantum state to the console, for visualization purposes only. To be used together with all other SQPAM methods with a 'print_state' kwarg.

measure(*qc: QuantumCircuit*, *treg_pos: int = 1*, *areg_pos: int = 0*) → None

Appends Measurements to an SQPAM audio circuit

From a quantum circuit with registers containing an SQPAM representation of quantum audio, creates two classical registers with compatible sizes and adds instructions for measuring them.

Parameters

- **qc** – A quantum circuit containing at least 2 quantum registers.
- **treg_pos** – Index of the SQPAM ('treg') register in the circuit. Default is 1
- **areg_pos** – Index of the SQPAM ('areg') register in the circuit. Default is 0

prepare(*angles: ndarray[Any, dtype[ScalarType]]*, *regsize: Tuple[int, int]*, *regnames: Tuple[str, str]*, *print_state: bool = False*) → QuantumCircuit

Prepares an SQPAM quantum circuit.

Creates a qiskit QuantumCircuit that prepares a Quantum Audio state using SQPAM (Single-Qubit Probability Amplitude Modulation). The quantum circuits used for audio representations typically contain two

qubit registers, namely, 'treg' (which encodes time/index information) and 'areg' (which encodes amplitude information).

Note: In SQPAM (as hinted by its name), the 'areg' (amplitude) register contains a single qubit. The audio samples are mapped into angles that parametrize single qubit rotations of 'areg' - which are then correlated to index states of the 'treg' register.

Parameters

- **angles** – Array with propbability amplitudes
- **regsize** – The size of both qubit registers in a tuple (treg_size, areg_size). 'treg_size' qubits for 'treg'; 'areg_size' qubits for 'areg'. For SQPAM, 'areg_size' is ALWAYS 1
- **regnames** – Label names for 'treg' and 'areg', passed as a tuple. For visualization purposes only.
- **print_state** – Toggles a simple print of the prepared quantum state to the console, for visualization purposes only.

Returns

A qiskit quantum circuit containing specific SQPAM preparation instructions.

reconstruct(*treg_size: int*, *counts: Counts*, *shots: int*, *inverted: bool = False*, *both: bool = False*) → ndarray[Any, dtype[ScalarType]]

Builds a digital Audio from qiskit histogram data.

Considering the SQPAM encoding scheme, it uses the histogram data stored in a Counts object (qiskit.result.Counts) to reconstruct an audio signal. It separates the even bins (sine coefficients) from the odd bins (cosine coefficients) of the histogram. Since the *SQPAM.convert()* method used the *np.arcsin()* function to prepare the state, the even bins should be used for reconstructing the signal.

However, the relations between sine and cosine means that a reconstruction with the cosine terms will build a perfectly inverted version of the signal. The user is able to choose between retrieving original or phase-inverted (or both) signals.

Parameters

- **treg_size** – Size of the 'treg' (time) register, leading to the full audio size.
- **counts** – Histogram from a qiskit job result (result.get_counts())
- **shots** – Amount of identical experiments ran by the qiskit job.
- **inverted** – Retrieves the cosine amplitudes instead (leading to a phase-inverted version of the signal).
- **both** – Retrieves both Sine and Cosine amplitudes in a tuple. Overwrites the 'inverted' argument.

Returns

A Digital Audio as a Numpy Array, or a Tuple with two signals. The signals are in float format.

treg_index_X(*qa: QuantumCircuit*, *t: int*, *treg: QuantumRegister*, *print_state: bool = False*) → None

Auxilary function for matching control conditions with time indexes.

Applies X gates on qubits of the time register whenever the respective bit of the current time index (in binary representation) is 0. As a result, the qubit will be flipped to |1> and succesfully trigger necessary control conditions of the circuit for this time index.

Parameters

- **qa** – The quantum circuit to be maniputated

- **t** – Time index that will be converted to binary form for comparison.
- **treg** – Quantum register of the time indexes.
- **print_state** – Toggles a simple print of the prepared quantum state to the console, for visualization purposes only. To be used together with all other SQPAM methods with a 'print_state' kwarg.

Examples

treg_index_X(qa, 6, treg) (a time register 'treg' with, say, 5 qubits in 'qa' at instant 6)

't' = 6 == '00110'. *treg_index_X()* applies X gates to qubits 0, 3 and 4
(right to left) of register 'treg'.

requantize_input(*audio: ndarray[Any, dtype[ScalarType]]*, *bit_depth: int*) → ndarray[Any, dtype[ScalarType]]

Requantizes Array signals and PCM audio signals.

Utilitary Function for downsizing the bit depth of an audio file. Very useful for using with the QSM encoder 'QuantumAudio('qsm')'.

Returns
(Numpy Array) Requantized audio signal.

CHAPTER

TWO

USING THE QUANTUM AUDIO MODULE

2.1 For building and manipulating quantum audio representations

The `quantumaudio` module implements a QuantumAudio class that is able to handle the encoding/decoding process of some quantum audio representations, such as: Building a quantum circuit for preparing and measuring the quantum audio state; simulating the circuit in Qiskit's *aer_simulator*; running the circuit in real hardware using *IBMQ* (as long as you have an account, provider and backend); necessary pre and post processing according to each encoding scheme; plotting and listening the retrieved sound.

The available encoding schemes are:

- QPAM - Quantum Probability Amplitude Modulation (Simple quantum superposition or "Amplitude Encoding") - `'qpam'`
- SQPAM - Single-Qubit Probability Amplitude Modulation (similar to FRQI image representations) - `'sqpam'`
- QSM - Quantum State Modulation (also known as FRQA) - `'qsm'`

For more information regarding the representations above, you can refer to this book chapter, or its abridged pre-release draft in ArXiv

2.1.1 Using the package

First of all, make sure you have all of the following dependencies installed:

- numpy
- matplotlib
- IPython.display
- bitstring
- qiskit

If you are on a Linux system, you might be able to install the dependencies by uncommenting and running this line:

[1]:
```
# !pip3 install numpy matplotlib ipython bitstring qiskit
```

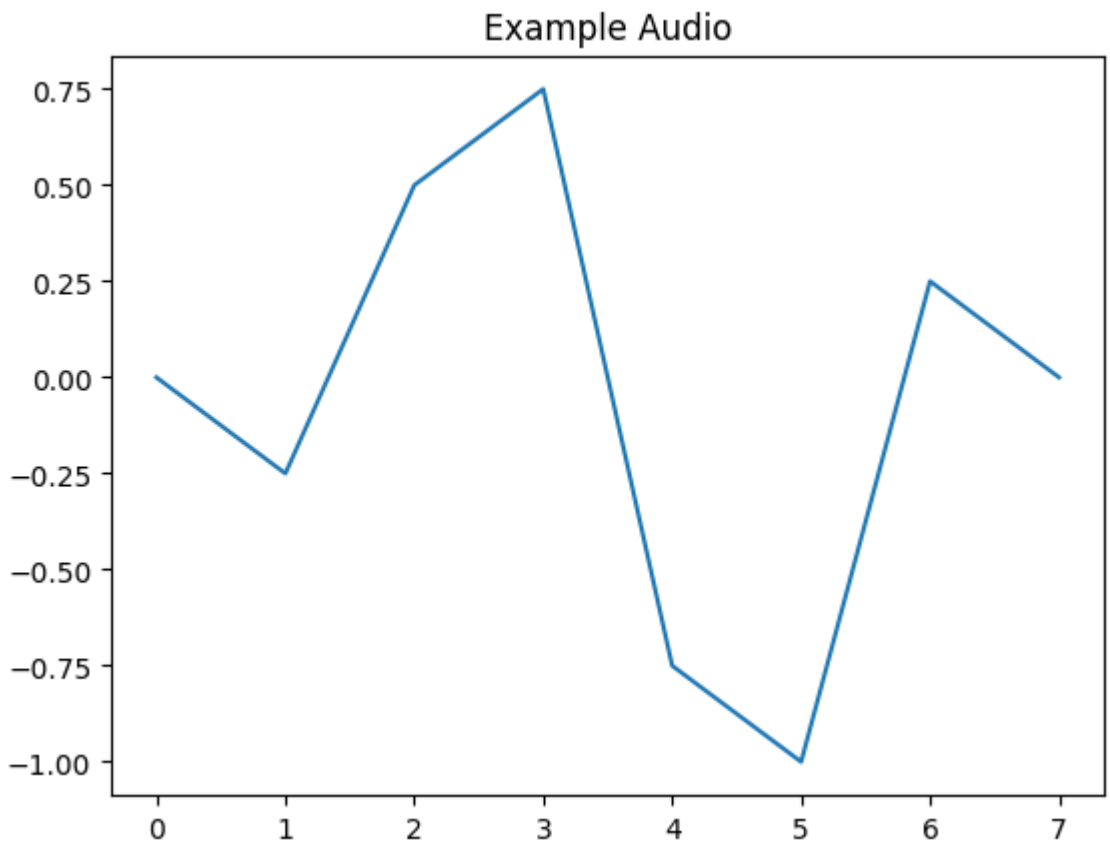
If you have "pip installed" the *quantumaudio* module, it should have downloaded all of the required dependencies:

[2]:
```
# !pip3 install quantumaudio
```

[3]:
```
import numpy as np
import quantumaudio as qa
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, Aer, execute
from qiskit.visualization import plot_histogram
```

Then, we create/load some example digital audio. QPAM and SQPAM are representations that can handle arrays with floating point or decimal numbers from -1 to 1 (somewhat similar to a PCM .wav or .flac file). In this example, we are using an audio signal with 8 samples:

[4]:
```
digital_audio = np.array([0., -0.25, 0.5 , 0.75,  -0.75  ,  -1.,  0.25 ,  0.])
plt.plot(digital_audio)
plt.title("Example Audio")
plt.show()
```



This is the current workflow for using the *quantumaudio* module: The QuantumAudio class encapsulates everything, from input, preprocessing, circuit generation, qiskit jobs, audio reconstruction and output.

The user chooses which encoding technique to use while instantiating a QuantumAudio object. It will then refer to specific encoder subclass methods.

[5]:
```
# qsound = qa.QuantumAudio('ENCODONG_SCHEME_HERE')
```

After instantiation, the first method to be used `load_input()` will load a copy of the input audio inside the object. It will print out the space requirements of the circuit.

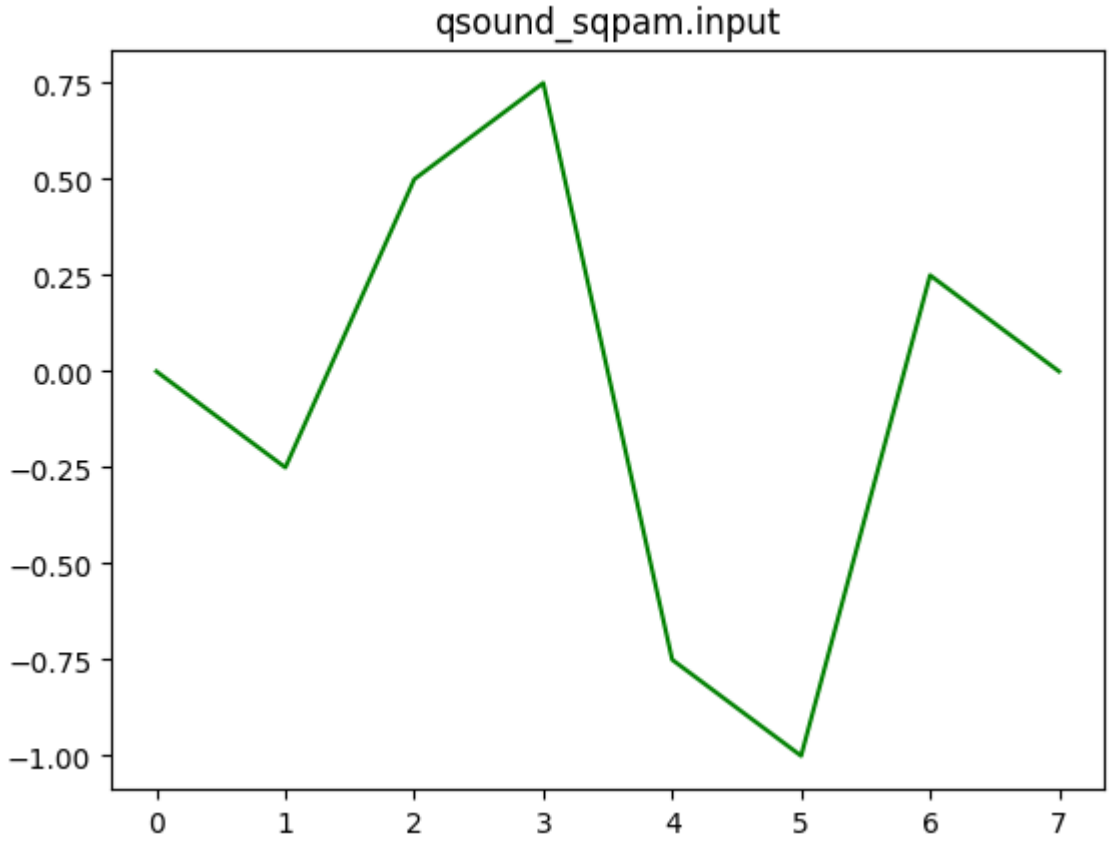
[6]:
```
qsound_qpam = qa.QuantumAudio('qpam')
qsound_qpam.load_input(digital_audio)

qsound_sqpam = qa.QuantumAudio('sqpam')
qsound_sqpam.load_input(digital_audio)
```

```
For this input, the QPAM representation will require:
         3 qubits for encoding time information and
         0 qubits for encoding amplitude information.
For this input, the SQPAM representation will require:
         3 qubits for encoding time information and
         1 qubits for encoding amplitude information.
```

[6]:
```
QuantumAudio
```

The loaded signal is acessible via the `input` attribute.

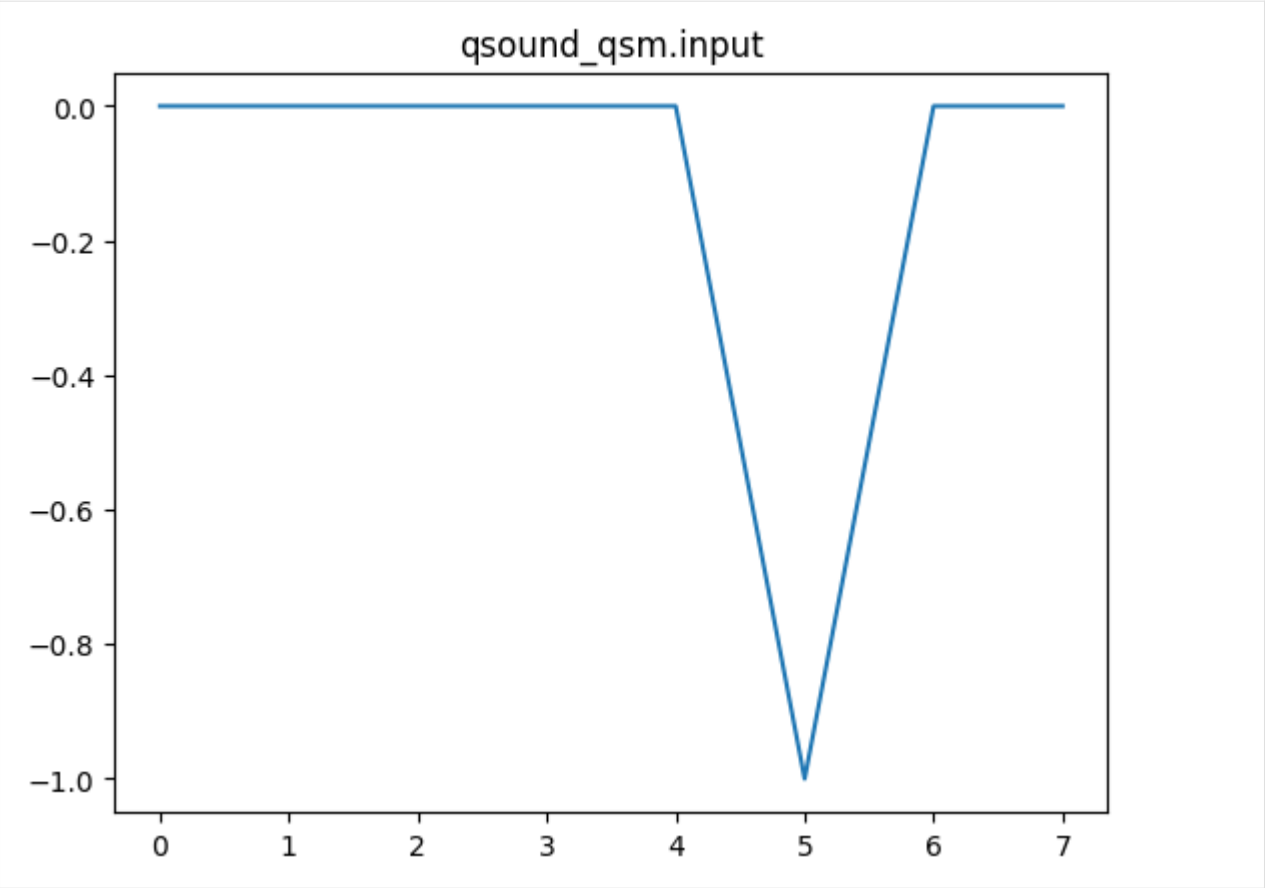
[7]:
```
plt.plot(qsound_qpam.input, 'r')
plt.title('qsound_qpam.input')
plt.show()
plt.plot(qsound_sqpam.input, 'g')
plt.title('qsound_sqpam.input')
plt.show()
```



However, the same *digital_audio* example will NOT work with QSM. The QSM works with integer values only, as it expects a *quantized* signal, so it will round the numbers by default, removing all of the decimals and destroying the input, as shown:

[8]:
```
qsound_qsm = qa.QuantumAudio('qsm')
qsound_qsm.load_input(digital_audio)

plt.plot(qsound_qsm.input)
plt.title('qsound_qsm.input')
plt.show()
```

```
For this input, the QSM representation will require:
         3 qubits for encoding time information and
         1 qubits for encoding amplítude information.
```



To load an input to the QSM encoder, we need to quantize (or re-quantize) the amplitudes of our signal. In this example, we have conveniently built a *digital_audio* simulating a PCM audio with 3-bit depth quantization. So we only need to multiply our signal by $2^{bitDepth-1}$ and retrieve the quantized version of the signal:

[9]:
```
bit_depth=3
quantized_ditial_audio = digital_audio*(2**(bit_depth-1))

print(quantized_ditial_audio)
```

```
[ 0. -1.  2.  3. -3. -4.  1.  0.]
```

(To prove that this quantization also holds for *actual* sound files, uncomment the following block and load a typical 1 second of audio, 44100 Hz, 16-bit depth using and check. We used a CC sweep file found in Freesound.org (Note: For this tutorial, this file would be too large to simulate).

Also Note: this requires the `soundfile` package.

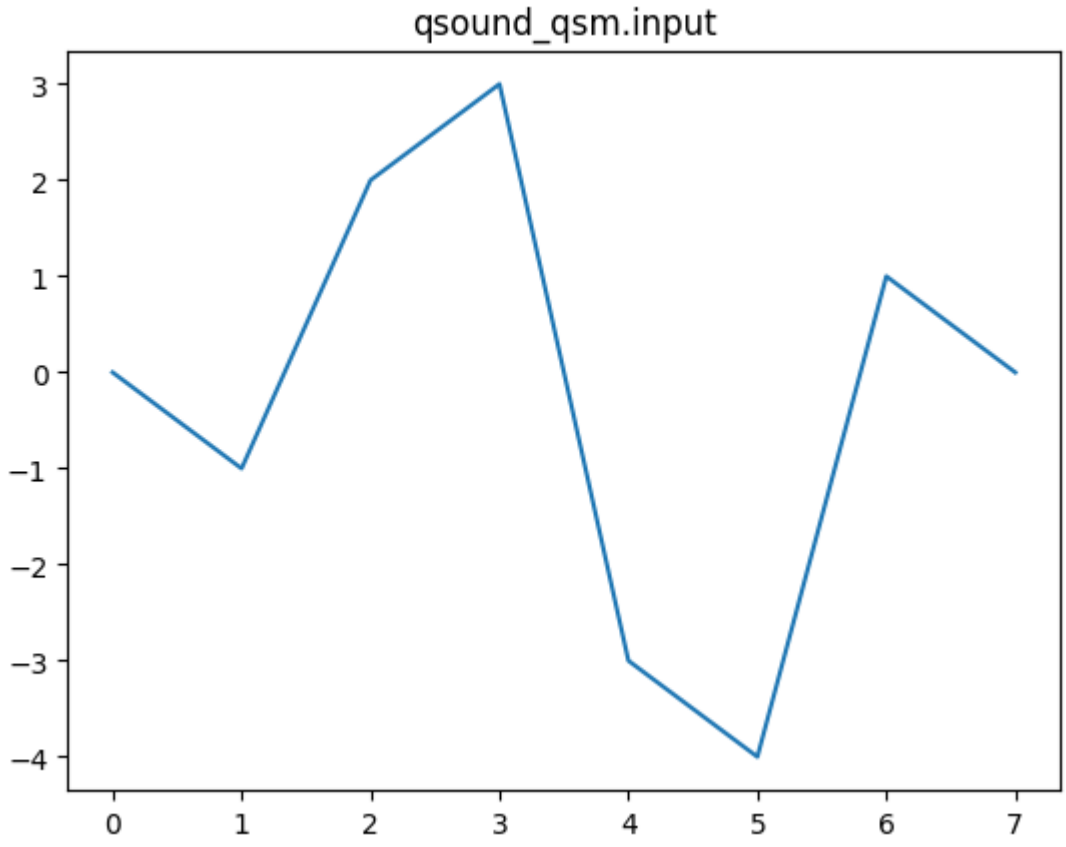
[10]:
```
# import soundfile as sf

# real_audio = sf.read('sweep_2_22000_log.wav')[0]

# bit_depth=16
# quantized_real_audio = real_audio*(2**(bit_depth-1))

# print(quantized_real_audio)
```

```
[  0.   7.  13. ... -74.  38.   0.]
```

Now we can load the quantized version ou our example audio to QSM, by specifying the *bit depth* as an additional argument. Remeber that the bit deph will also dictate the amount of qubits necessary to store the amplitude information.

[11]:
```
qsound_qsm = qa.QuantumAudio('qsm')
qsound_qsm.load_input(quantized_ditial_audio, 3)

plt.plot(qsound_qsm.input)
plt.title('qsound_qsm.input')
plt.show()
```

```
For this input, the QSM representation will require:
         3 qubits for encoding time information and
         3 qubits for encoding amplítude information.
```

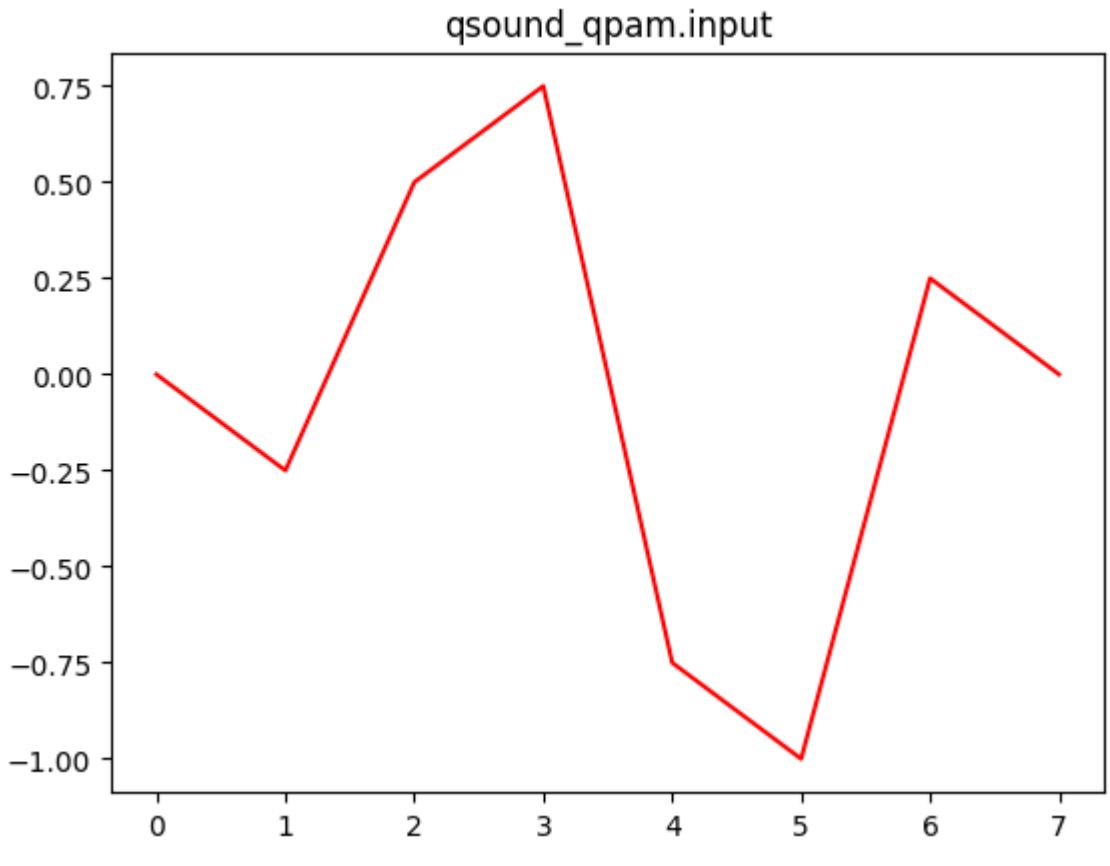


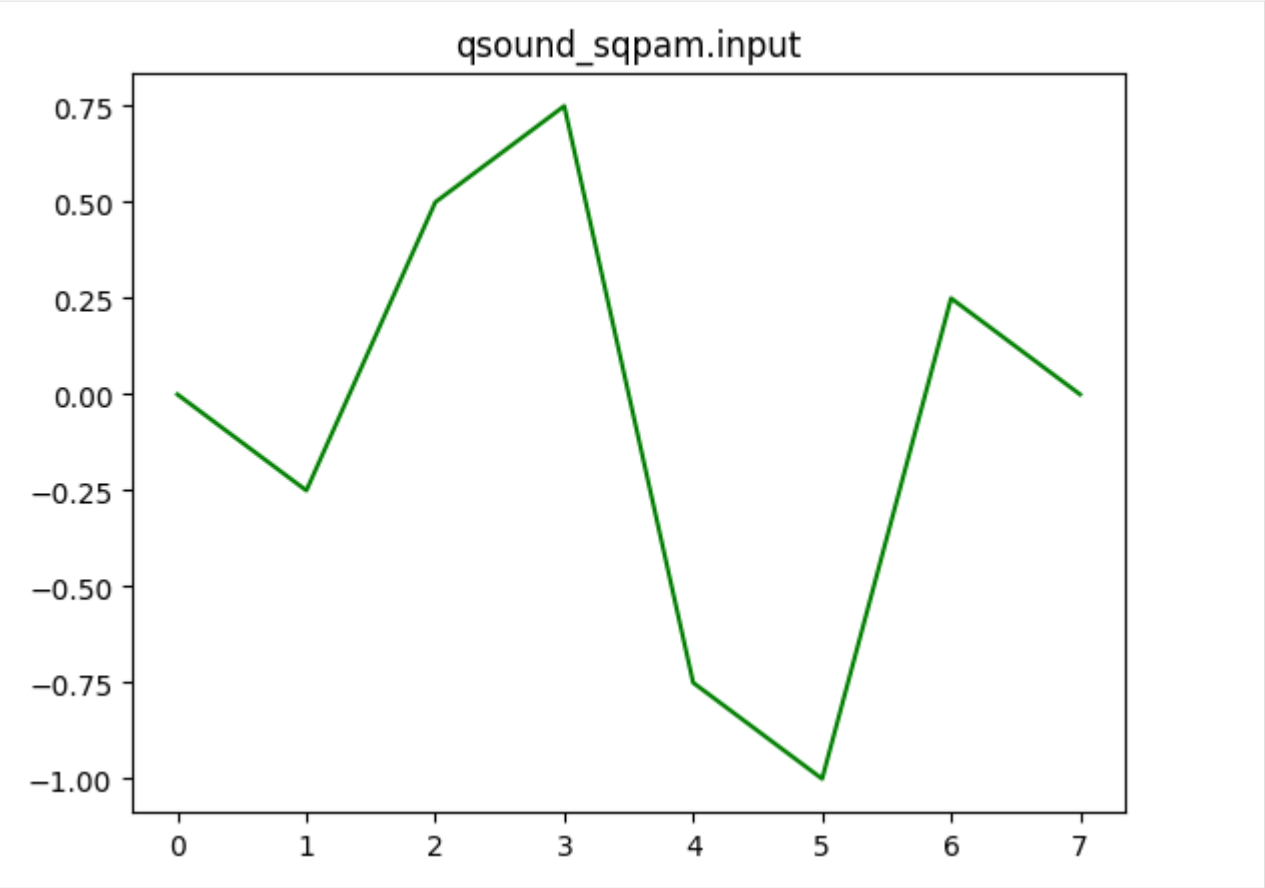
2.2 Note:

For usability reasons, QPAM and SQPAM can also handle quantized signals. The following code does exactly the same thing as before:

[12]:
```
# qsound__qpam = qa.QuantumAudio('qpam')
qsound_qpam.load_input(quantized_ditial_audio, 3)

# qsound_sqpam = qa.QuantumAudio('sqpam')
qsound_sqpam.load_input(quantized_ditial_audio, 3)

plt.plot(qsound_qpam.input, 'r')
plt.title('qsound_qpam.input')
plt.show()
plt.plot(qsound_sqpam.input, 'g')
plt.title('qsound_sqpam.input')
plt.show()
```

```
For this input, the QPAM representation will require:
         3 qubits for encoding time information and
         0 qubits for encoding amplítude information.
For this input, the SQPAM representation will require:
         3 qubits for encoding time information and
         1 qubits for encoding amplítude information.
```





This means that when working with quantized signals, we can easily switch between quantum audio representations - at least for encoding purposes (any additional quantum algorithm will have dramatically different impacts on each representation).

Now, let's generate quantum circuits with 3 steps:

1. Converting/preprocessing the signal for a specified encoding scheme (for example, qpam converts the signal into probability amplitudes, sqpam creates an array of angles) - This is done internally by the QuantumAudio class when calling the `prepare()` method.
2. Generating a Preparation circuit for the input, which encodes the classical information into the quantum system acording to the representation - this is also done with the `prepare()` method

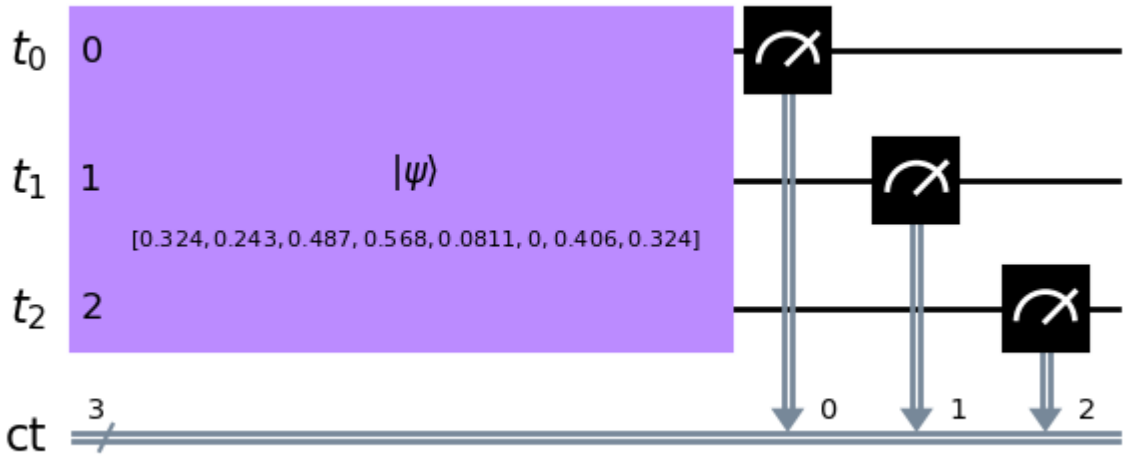
- (any custom quantum circuit, (aka, signal processing) could be applied at this point, by acessing the `circuit` attribute - qsound.circuit)

3. Inserting measurement instructions at the end of the circuit - measure() method

For now, we are only trying to prepare the quantum audio state and then measure it back: a *Quantum Audio Bypass Circuit*

[13]:
```
qsound_qpam.prepare()
qsound_qpam.measure()
qsound_qpam.circuit.draw('mpl')
```

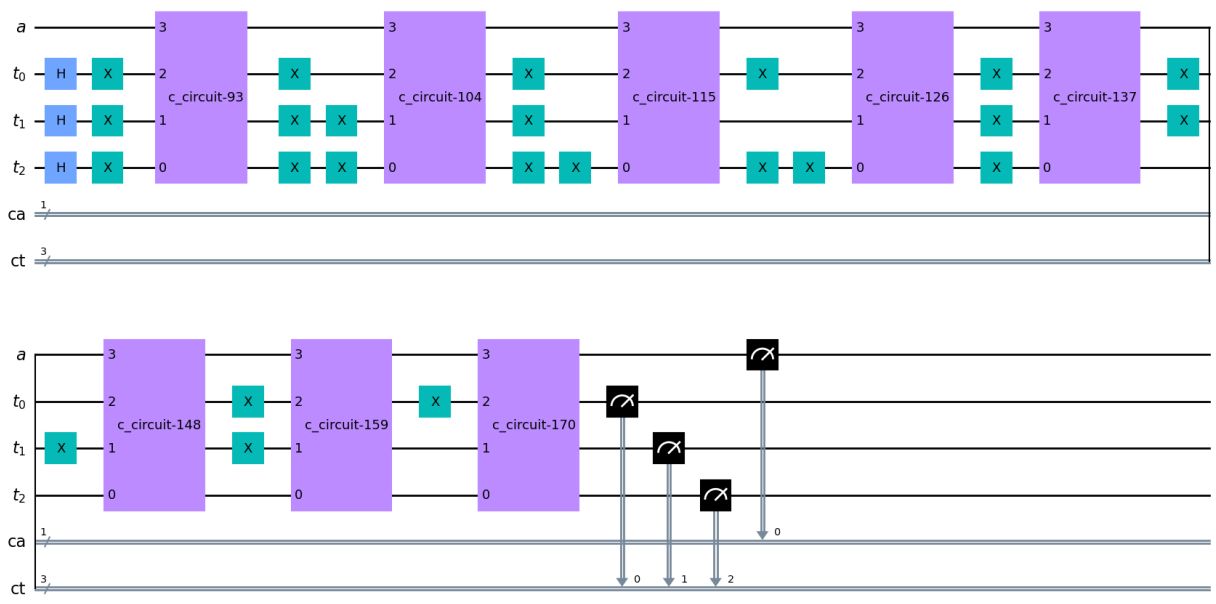
[13]:



If you are a one-liner, fell free to write everything in a single line. This is the advantage of the QuantumAudio class, and may be very useful for live performances:

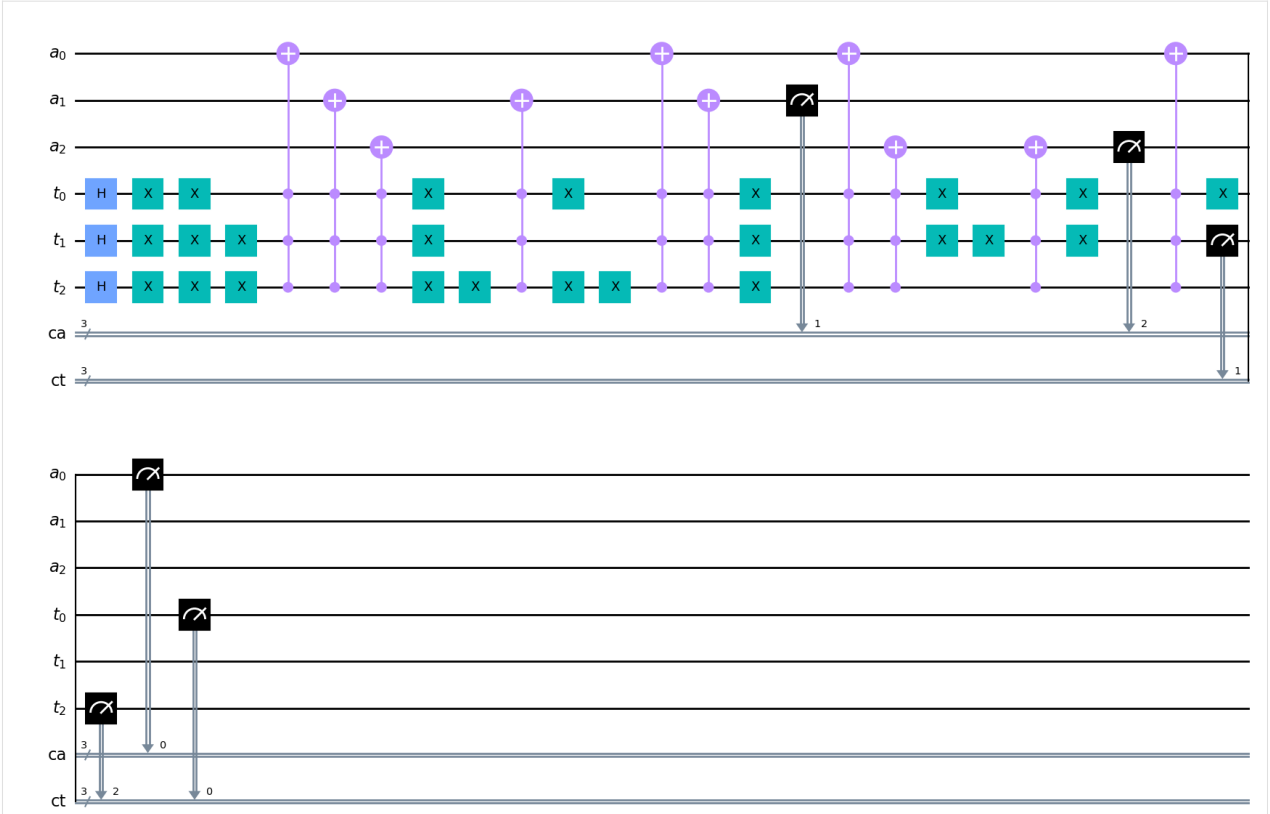
[14]:
```
qsound_sqpam.prepare().measure().circuit.draw('mpl')
```

[14]:



[15]:
```
qsound_qsm.prepare().measure().circuit.draw('mpl')
```

[15]:



Now that we have a quantum circuit, we can run it on *aer_simutator*, or use it elsewhere. There are attributes storing qiskit `result` and `counts`.

QPAM:

[16]:
```
# Default values: QuantumAudio.run(shots=10, backend_name='aer_simulator', provider=Aer)

# Simulating qsound_qpam.circuit in 'aer_simulator' with 1 thousand shots:
shots = 1000
qsound_qpam.run(shots)
print(qsound_qpam.result)
print('------------------------------------')
print(qsound_qpam.counts)
plot_histogram(qsound_qpam.counts)
```

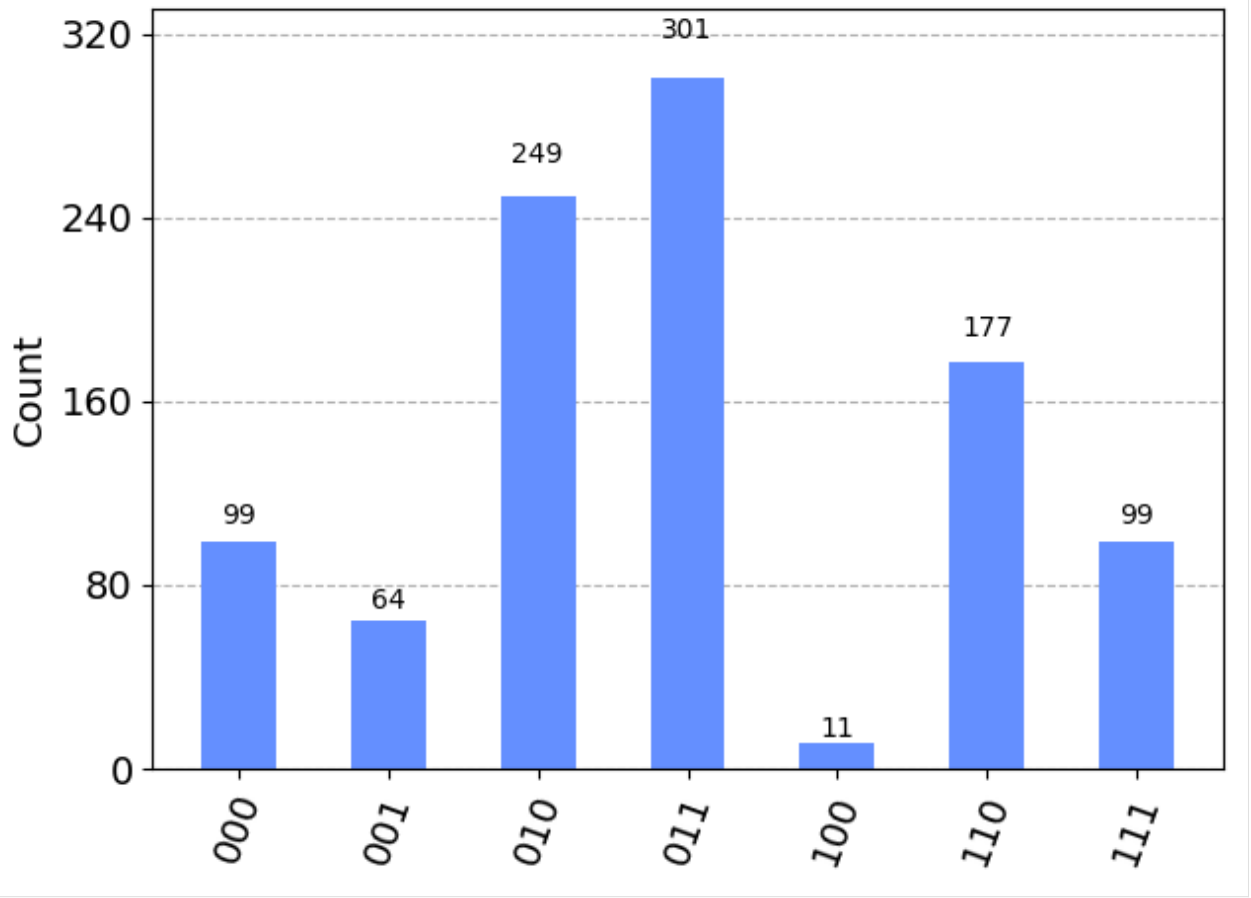
```
Result(backend_name='aer_simulator', backend_version='0.11.1', qobj_id='e3d8679a-195a-
↪42f5-9958-8a631e6459ec', job_id='48957fd5-50b4-49a1-bfbe-b90df251166c', success=True,␣
↪results=[ExperimentResult(shots=1000, success=True, meas_level=2,␣
↪data=ExperimentResultData(counts={'0x4': 11, '0x0': 99, '0x2': 249, '0x7': 99, '0x3':␣
↪301, '0x1': 64, '0x6': 177}), header=QobjExperimentHeader(clbit_labels=[['ct', 0], ['ct
↪', 1], ['ct', 2]], creg_sizes=[['ct', 3]], global_phase=0.0, memory_slots=3, metadata=
↪{}, n_qubits=3, name='circuit-91', qreg_sizes=[['t', 3]], qubit_labels=[['t', 0], ['t',
↪ 1], ['t', 2]]), status=DONE, seed_simulator=2878838266, metadata={'noise': 'ideal',
↪'batched_shots_optimization': False, 'measure_sampling': True, 'parallel_shots': 1,
↪'remapped_qubits': False, 'active_input_qubits': [0, 1, 2], 'num_clbits': 3, 'parallel_
↪state_update': 8, 'sample_measure_time': 0.000369611, 'num_qubits': 3, 'device': 'CPU',
```

(continues on next page)

(continued from previous page)

```
↪ 'input_qubit_map': [[2, 2], [1, 1], [0, 0]], 'method': 'statevector', 'fusion': {
↪'applied': False, 'max_fused_qubits': 5, 'threshold': 14, 'enabled': True}}, time_
↪taken=0.002522784)], date=2022-12-22T17:43:58.241188, status=COMPLETED,␣
↪header=QobjHeader(backend_name='aer_simulator', backend_version='0.11.1'), metadata={
↪'time_taken': 0.0027873, 'time_taken_execute': 0.002587719, 'mpi_rank': 0, 'num_mpi_
↪processes': 1, 'max_gpu_memory_mb': 0, 'max_memory_mb': 15878, 'parallel_experiments':␣
↪1, 'time_taken_load_qobj': 0.000188461, 'num_processes_per_experiments': 1, 'omp_
↪enabled': True}, time_taken=0.0030050277709960938)
----------------------------------
{'100': 11, '000': 99, '010': 249, '111': 99, '011': 301, '001': 64, '110': 177}
```

[16]:



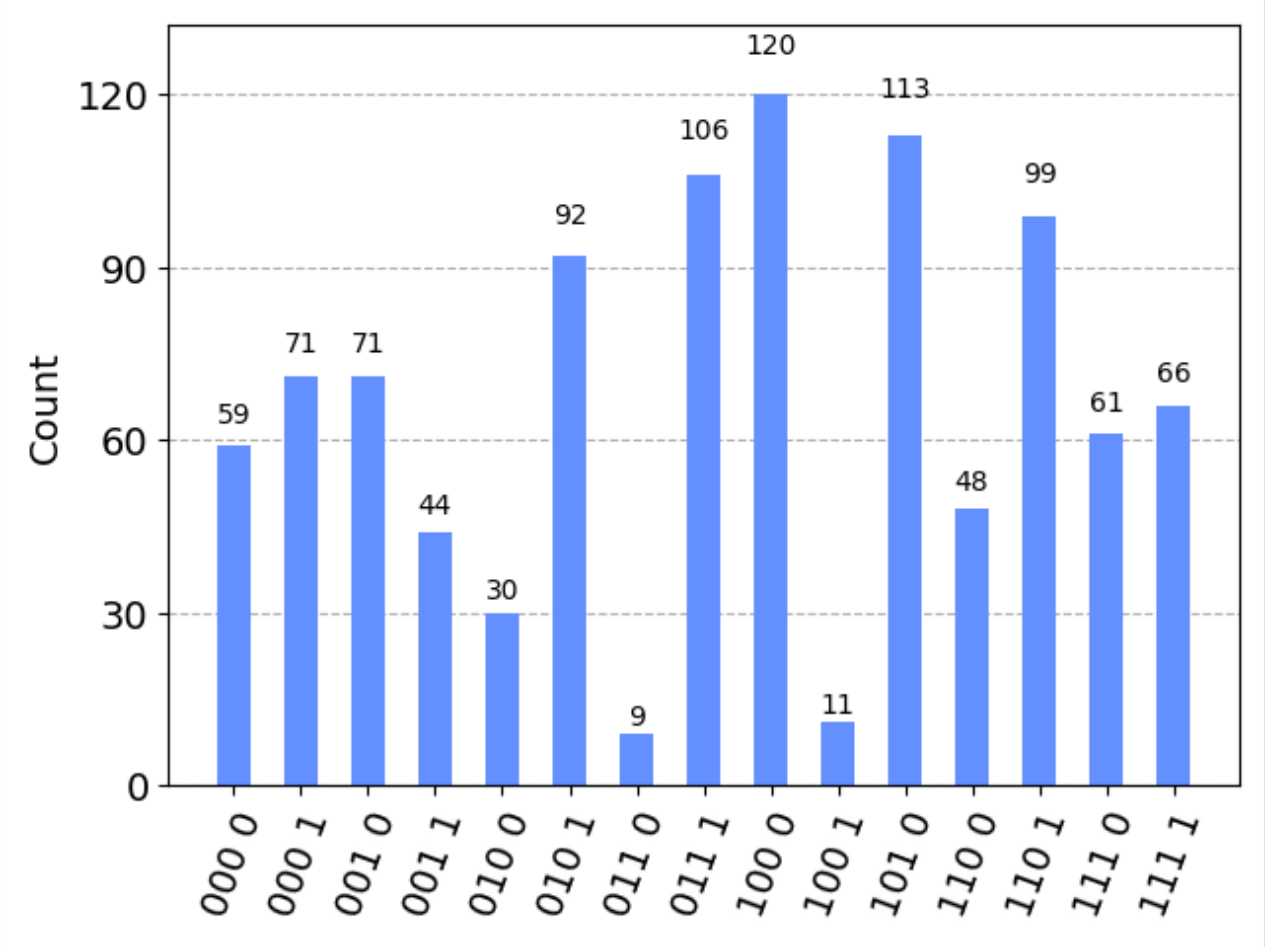
SQPAM:

[17]:

```
qsound_sqpam.run(shots)
print(qsound_sqpam.counts)
plot_histogram(qsound_sqpam.counts)
```

```
{'001 0': 71, '110 1': 99, '101 0': 113, '111 1': 66, '000 1': 71, '111 0': 61, '001 1':␣
↪44, '011 1': 106, '011 0': 9, '100 0': 120, '110 0': 48, '010 1': 92, '100 1': 11,
↪'000 0': 59, '010 0': 30}
```

[17]:



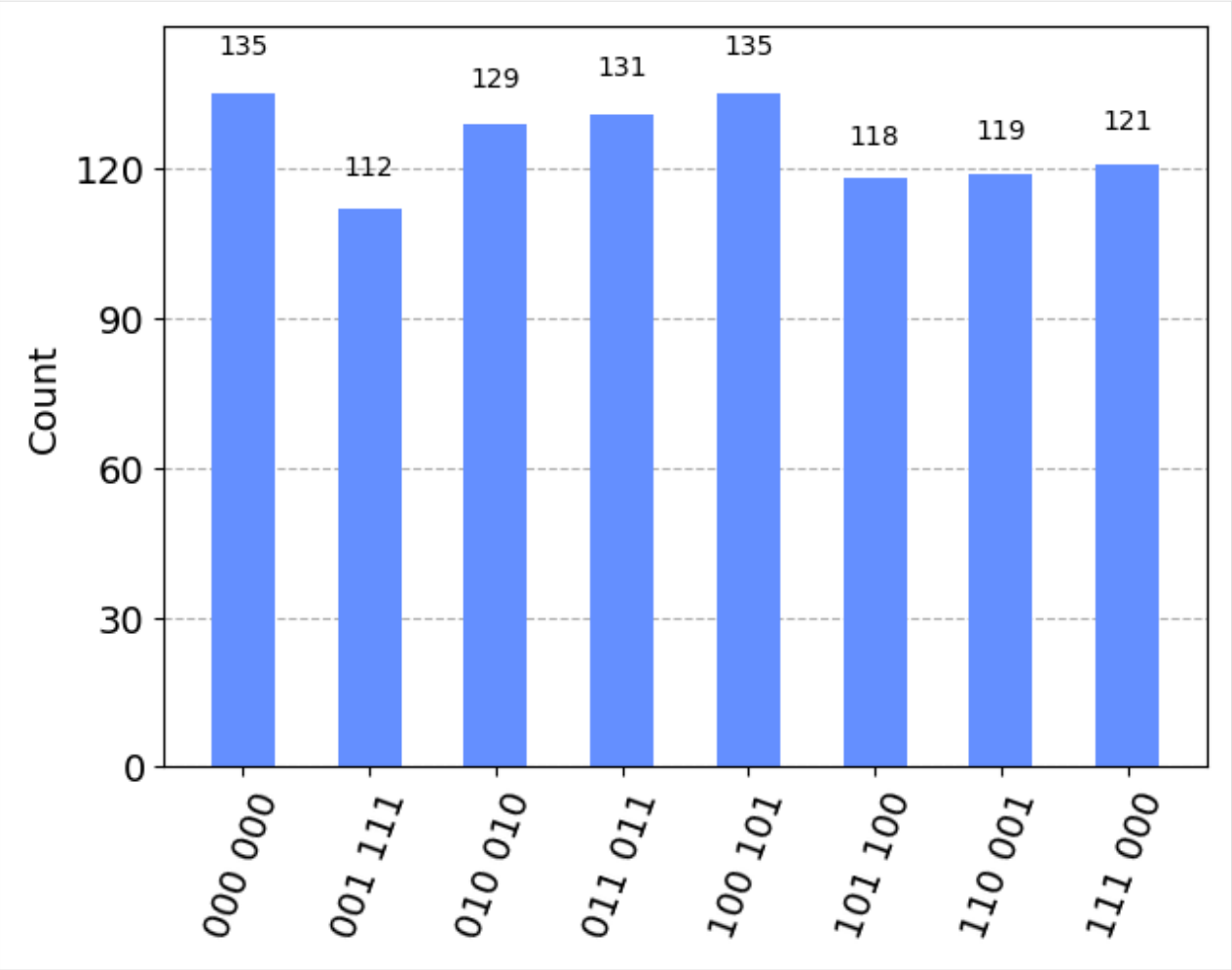
QSM:

[18]:

```
qsound_qsm.run(shots)
print(qsound_qsm.counts)
plot_histogram(qsound_qsm.counts)
```

```
{'010 010': 129, '110 001': 119, '111 000': 121, '100 101': 135, '001 111': 112, '011 011
↪': 131, '000 000': 135, '101 100': 118}
```

[18]:

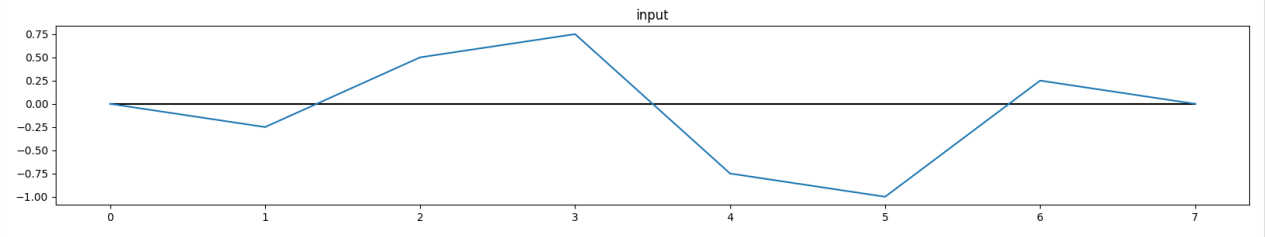


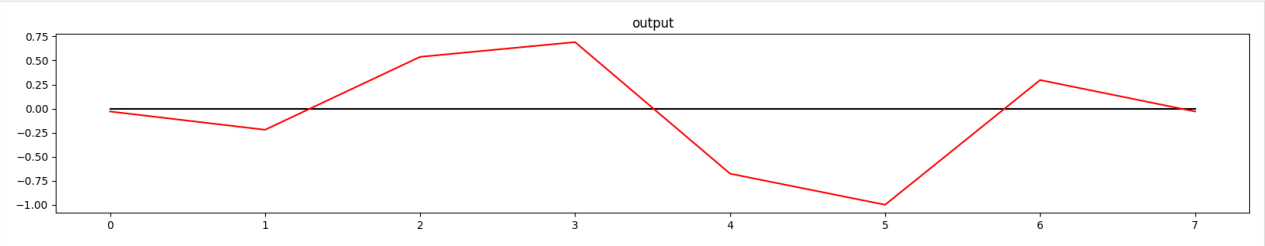
The last step of the process is to decode/reconstruct the histogram output into a digital audio output using the `reconstruct_audio()` method:

QPAM

[19]:

```
qsound_qpam.reconstruct_audio()
qsound_qpam.plot_audio()
```





You might notice that the reconstructed signal is not perfect. This is the case for QPAM and SQPAM, as they have proabilistic retrieval characteristics. This means: The higher the amount of experiments (`shots`), the higher the precision of the reconstructed signal will be:

[20]:
```
qsound_qpam.output
```

[20]:
```
array([-0.03020621, -0.22025645,  0.53801821,  0.69100562, -0.6767354 ,
       -1.        ,  0.29672665, -0.03020621])
```

[21]:
```
qsound_qpam.input
```

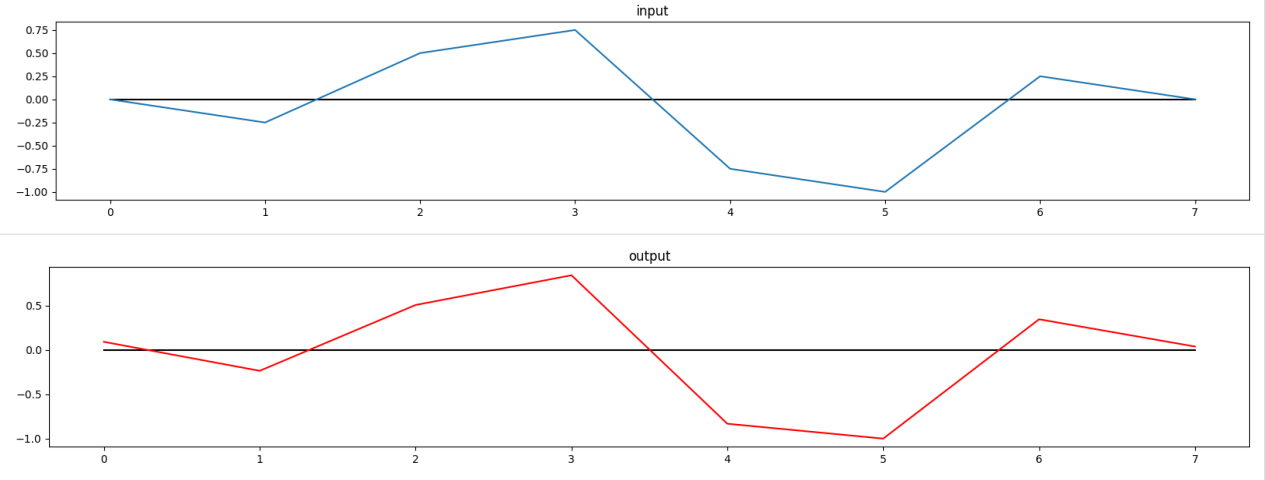
[21]:
```
array([ 0.  , -0.25,  0.5 ,  0.75, -0.75, -1.  ,  0.25,  0.  ])
```

[22]:
```
# Reconstrunction Error
sum(qsound_qpam.output - qsound_qpam.input)
```

[22]:
```
0.0683462015824361
```

SQPAM

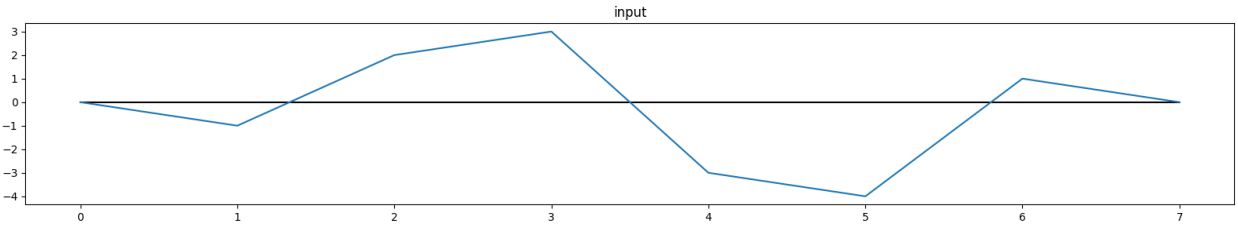
[23]:
```
qsound_sqpam.reconstruct_audio()
qsound_sqpam.plot_audio()
```

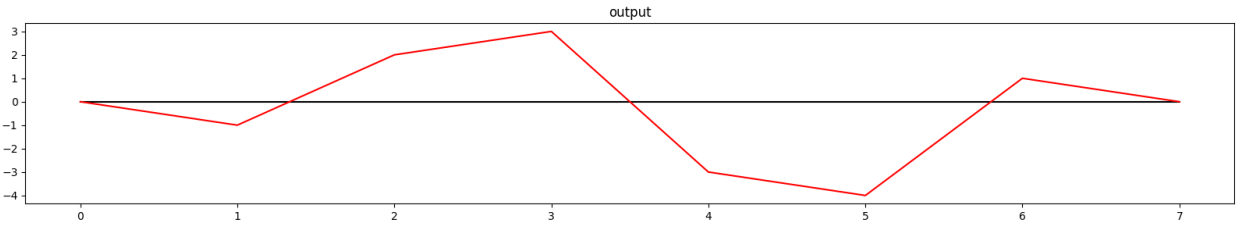


QSM

Note: QSM has a deterministic retrieval procedure, hence, perfect reconstruction

[24]:
```
qsound_qsm.reconstruct_audio()
qsound_qsm.plot_audio()
```





Finally, listen to the output with `listen()` (in this case, the output is too short to be heard):

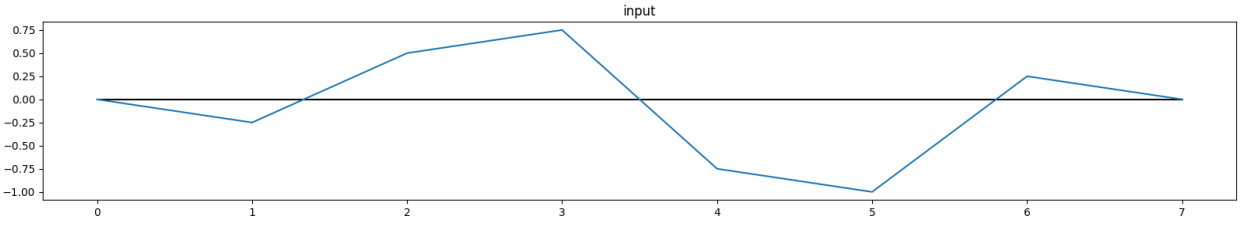
[25]:
```
sample_rate = 3000
qsound_qpam.listen(sample_rate)
```

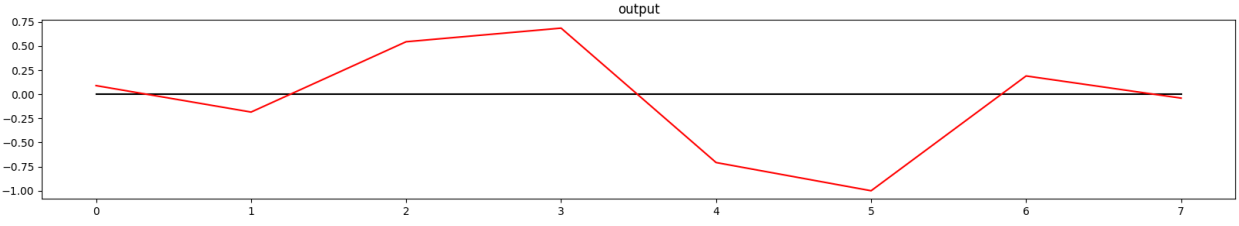
```
<IPython.lib.display.Audio object>
```

For one-liners, this whole process can be written, for any representation, as:

[26]:
```
qsound = qa.QuantumAudio('qpam')
qsound.load_input(quantized_ditial_audio, 3).prepare().measure().run(1000).reconstruct_
↪audio().plot_audio()
```

```
For this input, the QPAM representation will require:
        3 qubits for encoding time information and
        0 qubits for encoding amplítude information.
```





This summarizes the introduction to the *quantumaudio* module. For more functionalities and potential applications, refer to the documentation and to the Github Repository Readme file.

.

.

Download this notebook from the latest Github release.

Itaborala @ ICCMR Quantum https://github.com/iccmr-quantum/quantumaudio

CHAPTER

THREE

USING QUANTUM CIRCUITS TO GENERATE AND MANIPULATE WAVETABLES ON SUPERCOLLIDER

3.1 Part 1 : The “Dithering” and the “Geiger Counter” Effects

This example is a simple application of a quantum audio representation. Wavetable synthesis is a good option for the near-term quantum technology, since it requires a small audio sample, containing just one period of a desired oscillattor.

For more information regarding quantum representations of audio and these applications, you can refer to this book chapter, or its abridged pre-release draft in ArXiv

The main idea of this notebook is to: + Create a wavetable in python + Create a quantum circuit that prepares a quantum audio state from the table using a quantum audio encoding scheme - and measure it back. + Use a SuperCollider (SC) server as a wavetable synthesis engine + Load the wavetable into SuperCollider and start listening to the synthesizer + Simulate the quantum circuit several times and update the wavetable with the results in real time

For creating the quantum audio state we will use the IBM’s *Qiskit* language/framework and the *quantumaudio* module for building the quantum audio circuits. For more information on how to use the *quantumaudio.py* module, refer to the Quantumaudio module Tutorial.

3.1.1 This Notebook is suposed to be run along with a SuperCollider server (scsynth or supernova), contatining a pre-defined or a pre-stored wavetable SynthDef called “”, like the one declared in the example file *Wavetables.scd* found here.

3.1.2 If you have a SuperCollider client (like SCIDE, SCApp, etc), you can open the Wavetables.scd script. Boot the server by positioning the cursor at the “s.boot;” line and then pressing “ Ctrl(Cmd)+Enter “. Then store (and load) the :nbsphinx-math:`qtable `SynthDef, by positioning the cursor at any line inside the definition and pressing” Ctrl(Cmd)+Enter “.

Python dependencies:

First, make sure you have all of the following dependencies installed: - quantumaudio - numpy - matplotlib - IPython.display - bitstring - qiskit - liblo - python-supercollider client

Import everything:

[1]:
```
import numpy as np
from numpy import pi
import time
```

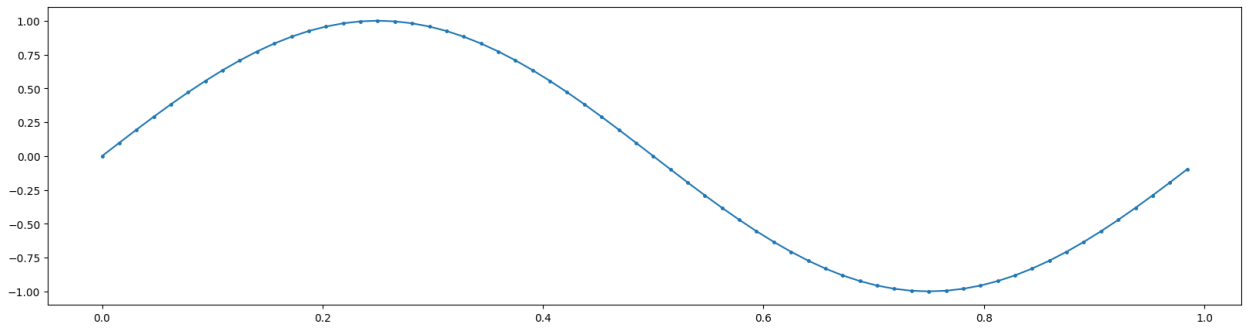
(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, Aer, execute
```

Now we will create a wavetable with one period of a sine function. The table will have 64 samples, which will produce a 6-qubit QPAM audio state:

[2]:
```
# Audio Sample / Table
qubit_size = 6
table_size = 2**qubit_size
t = np.linspace(0, 1-1/table_size, table_size)
sinewave = np.sin(2*np.pi*t)
plt.figure(figsize=(20,5))
plt.plot(t, sinewave, '.-', ms=5)
plt.show()
```



3.2 Stablishing a connection with SuperCollider

SuperCollider is a musical platform with a server-client structure. The server side is a powerful synthesis engine, *scsynth*, similar to what can be found in game development environments like *Unreal Engine*. The client side is an interpreted programming language with a javascript-like sintax, *sclang*. The language is used to send real-time messages to the synthesis engine using the OSC (Open Sound Control) communication protocol. Client messages could be instructions to create/instantiate synth nodes according to some definition, allocate buffers, change synth parameters, run a complex synthesis routine, etc.

Due to this structure, many projects have been able to wrap *sclang* messages in other programming languages, like Python. One very successful python-based Supercollider client app & IDE is called FoxDot. But since we wish to communicate with SC from a Jupyter Notebook, we will use a more minimalistic approach, with the python-supercollider module. It has a minimal set of objects from *sclang* (Sever, Buffer, Synth, Group, Bus), that can act as a SC client, building OSC messages and sending them directly to *scsynth*.

[3]:
```
from supercollider import Server, Buffer, Synth
```

Let's connect to a running *scsynth*: ##### (Note: *you need to boot your SuperCollider server before this step*)

[4]:
```
server = Server()
```

SuperCollider has a special way of dealing with wavetables. The wavetable buffers are written in a specific supercollider wavetable format, which are optimized in a way that requires less runtime operations. Any UGen that deals with wavetables (like the 'Osc' UGen in the *Wavetables.scd* file) will expect to read a buffer in this format. This buffer is twice the size of the original table and its content stores a pre-processed version of the wavetable. The following function handles this buffer pre-processing, according to specification:

[5]:
```
def toWavetable(signal):
    wavetable = np.zeros(2*len(signal))
    wavetable[0::2] = 2*signal - np.roll(signal, -1)
    wavetable[1::2] = np.roll(signal, -1) - signal
    return wavetable
```

The next function will be used to update an allocated buffer in SuperCollider, using the SC 'set' message:

[6]:
```
def updateBuffer (buffer, signal):
    wavetableformat = toWavetable(signal)
    buffer.set(wavetableformat)
```

Let's try to listen to 1 second of our sinewave on SC. First we need to allocate a buffer, and update it with our wavetable:

[7]:
```
b = Buffer.alloc(server, 2*len(sinewave))
```

[8]:
```
wavetable = sinewave
updateBuffer(b, wavetable)
```

Let's synth!

The following code will instantiate a Synth node in SC and play it. Then it waits for 1 second, and sends a 'free' message to the server, releasing the instance. Notice how you can set SynthDef keyword arguments using a dictionary in the third positional argument of the Synth function:

Be Careful with your sound speaker/headphone volume! The default gain is set to -25dB in the line below, which is audible in many systems. Tune up the gain as necessary.

[9]:
```
synth = Synth(server, "qTable", {"buf" : b, "gain" : -25, "freq" : 350})
time.sleep(1)
synth.free()
```

So far, so good! You should have listened to a sinusoidal oscillator created in python and numpy sine function! Now, let's use quantum computing to manipulate this table.

3.3 Quantum ""Dithering""

First, let's explore a simple quantum circuit example. We will just prepare a QPAM quantum audio circuit from the sine table using the QuantumAudio class inside the *quantumaudio.py* module (refer to the *Quantum Audio Module Demo* to learn more), then apply measurements imediatelly after the preparation - a *"quantum audio loopback"*.

[10]:
```
import quantumaudio as qa
```

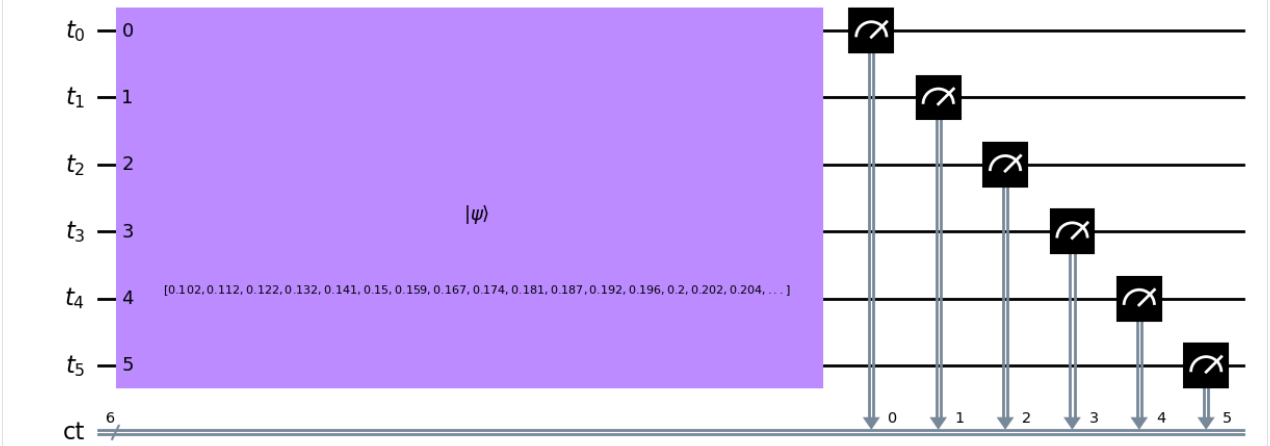
[11]:
```
qsine = qa.QuantumAudio('qpam').load_input(sinewave)
```

```
For this input, the QPAM representation will require:
        6 qubits for encoding time information and
        0 qubits for encoding amplitude information.
```

This simple loopback circuit can already lead to some interesting artirtic sounding reults - if we use QPAM's inherently *imperfect, probabilistic retrieval* characteristics wisely.

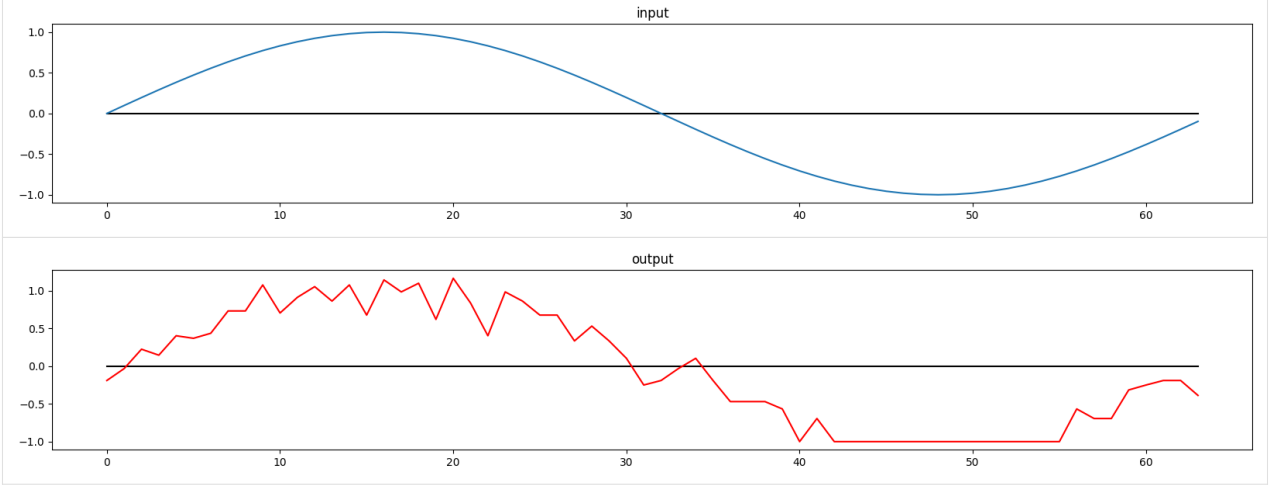
[12]:
```
qsine.prepare().measure()
qsine.circuit.draw('mpl')
```

[12]:



Now that everything is set up, and we can run (or simulate in this case) the circuit and then use the results to update the wavetable:

[13]:
```
shots = 1024
wavetable = qsine.run(shots).reconstruct_audio().output
qsine.plot_audio()
```



Start the synth again,

[14]:
```
synth = Synth(server, "qTable", {"buf" : b, "gain" : -35, "freq" : 350})
```

and update the buffer:

[15]:
```
updateBuffer(b, wavetable)
```

Voila!! You can notice that the sound is not perfectly reconstructed, due to the small number of shots. The result is the introduction of some noise on the signal. This is referred as a *Quantum Dithering Effect*. The higher the number of shots, the lower is the noise amplitude.

For stopping the synth, run:

[16]:
```
synth.free()
```

3.4 Wavetable Update Loops and the Geiger Counter Effect

An immediate extension of this buffer updating idea is to simulate the circuit seversal times, changing the number of shots on each run while listening to the variations in real time.

Let's start our synth again, with our original signal:

[17]:
```
updateBuffer(b, sinewave)
synth = Synth(server, "qTable", {"buf" : b, "gain" : -35, "freq" : 250})
```

Then we will make a loop and run the circuit several times. The effect will be a 'colour' change in the sound timbre. (fell free to experiment with different parameters):

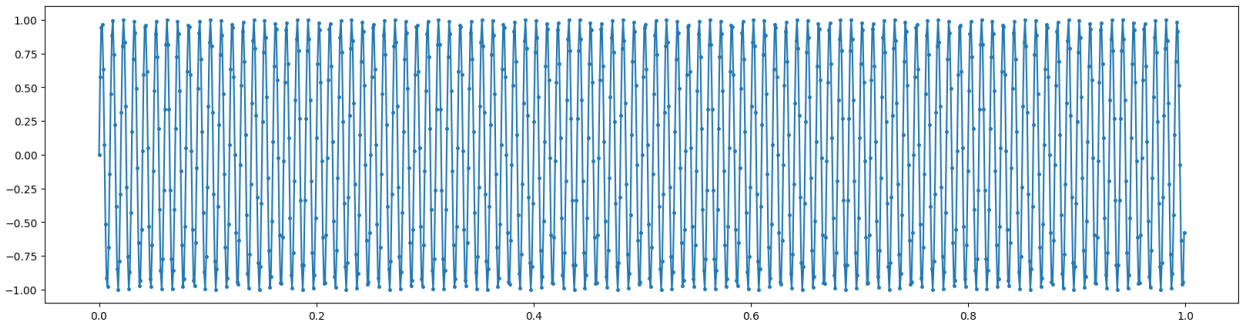
[18]:
```
SHOTS =[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300,␣
↪400, 500, 1000, 2000, 3000, 4000, 5000, 10000, 20000, 100000]
```

[19]:
```
time_interval = 0.1
for i in SHOTS:
    wavetable = qsine.run(i).reconstruct_audio().output
    updateBuffer(b, wavetable)
    time.sleep(time_interval)
```

[20]:
```
synth.free()
```

But depending on the choice of wavetable & parameters, with a slow reading frequency (~1Hz), the auditory result reminds the sound of a *Geiger Counter*, a physics measuring device used to detect radiation particles. Let's build a geiger counter example with a 10-qubit (1024 samples) wavetable with a 100Hz sinewave:

[21]:
```
# Audio Sample / Table
qubit_size = 10
size = 2**qubit_size
t = np.linspace(0, 1-1/size, size)
geiger = np.sin(2*np.pi*100*t)
plt.figure(figsize=(20,5))
plt.plot(t, geiger, '.-', ms=5)
plt.show()
```



[22]:
```
qgeiger = qa.QuantumAudio('qpam').load_input(geiger).prepare().measure()
```

```
For this input, the QPAM representation will require:
         10 qubits for encoding time information and
         0 qubits for encoding amplítude information.
```

[23]:
```
g = Buffer.alloc(server, 2*len(geiger))
```

[24]:
```
updateBuffer(g, geiger)
```

[25]:
```
synth = Synth(server, "qTable", {"buf" : g, "gain" : -22, "freq" : 1})
```

[26]:
```
SHOTS =[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300,␣
↪400, 500, 1000, 2000, 3000, 4000, 5000, 10000, 20000, 50000, 100000]
time_interval = 1
for i in SHOTS:
    wavetable = qgeiger.run(i).reconstruct_audio().output
    updateBuffer(g, wavetable)
    time.sleep(time_interval)
```

[27]:
```
synth.free()
```

At first, the circuit has too litle shots, and the distribution will peak in a few states, resulting on the sharp attacks typical of a geiger counter. As the amount of shots increases, the sound turns into noise. Then the statistical distribution slowly starts to take the form of the original table, and the original sound arises from the noise.

.

.

Download this notebook from the latest Github release.

Itaborala @ ICCMR Quantum https://github.com/iccmr-quantum/quantumaudio

CHAPTER

FOUR

USING QUANTUM CIRCUITS TO GENERATE AND MANIPULATE WAVETABLES ON SUPERCOLLIDER

4.1 Part 2 : SQPAM Y-Rotation Effect

4.1.1 This Notebook is the Part 2 continuation of the Examples with SuperCollider notebook (part 1 here) notebook. If you haven't learned how to connect Jupyter with SuperCollider please refer to the first part. To learn about the quantumaudio module, refer to the tutorial.

After exploring some purely "quantum loopback" effects, consisting of preparations directly succeded by measurements only, let's try to manipulate our audio in the quantum domain.

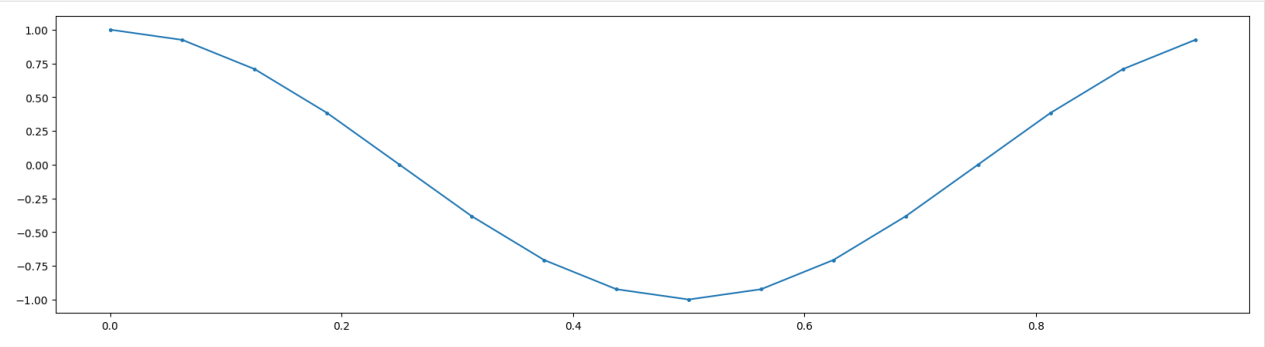
For this example, we are going to use another quantum audio scheme: SQPAM

Let's first just import the necessary libraries:

[1]:
```
import numpy as np
from numpy import pi
import time
import quantumaudio as qa
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, Aer, execute
from supercollider import Server, Buffer, Synth
```

Similarly to part 1, we are going to use a sinusoidal table (one period of a cosine function in this case), but we won't need as many qubits for now (the simulations will take too long otherwise):

[2]:
```
# Audio Sample / Table
qubit_size = 4
table_size = 2**qubit_size
t = np.linspace(0, 1-1/table_size, table_size)
sinewave = np.cos(2*np.pi*t)
plt.figure(figsize=(20,5))
plt.plot(t, sinewave, '.-', ms=5)
plt.show()
```



Then let's load this table in a 5-qubit SQPAM QuantumAudio object:

[3]:
```
qsine = qa.QuantumAudio('sqpam').load_input(sinewave)
```

```
For this input, the SQPAM representation will require:
         4 qubits for encoding time information and
         1 qubits for encoding amplítude information.
```

We will take advantage of our quantum audio scheme to explore some effects. The amplitudes are stored as Bloch sphere rotation angles, using controlled **:math:`R_y`** gates. An interesting and simple idea to test would be to just *shift* all the angles (and therefore, all the amplitudes) by a certain amount:

[4]:
```
def add_ry_sqpam(qa, angle, shots=10000):
    qa.prepare()
    qa.circuit.ry(angle, 0)
    qa.measure()
    qa.run(shots).reconstruct_audio()
    return qa.output
```

[5]:
```
# This is just a convenient function to visualise what we are doing.
def plot_out(out, color='darkblue'):
    plt.figure()
    plt.plot(out, color)
    plt.show()
    plt.close()
```

Then, we import the same SuperCollider functions used before to listen to the result.

[6]:
```
def toWavetable(signal):
    wavetable = np.zeros(2*len(signal))
    wavetable[0::2] = 2*signal - np.roll(signal, -1)
    wavetable[1::2] = np.roll(signal, -1) - signal
    return wavetable
```

[7]:
```
def updateBuffer (buffer, signal):
    wavetable = toWavetable(signal)
    buffer.set(wavetable)
```

[8]:
```
server = Server()
```

[9]:
```
b = Buffer.alloc(server, 2*len(sinewave))
```

[10]:
```
wavetable = sinewave
updateBuffer(b, wavetable)
```

Be Careful with your sound speaker/headphone volume! The default gain is set to -25dB in the line below, which is audible in many systems. Tune up the gain as necessary.

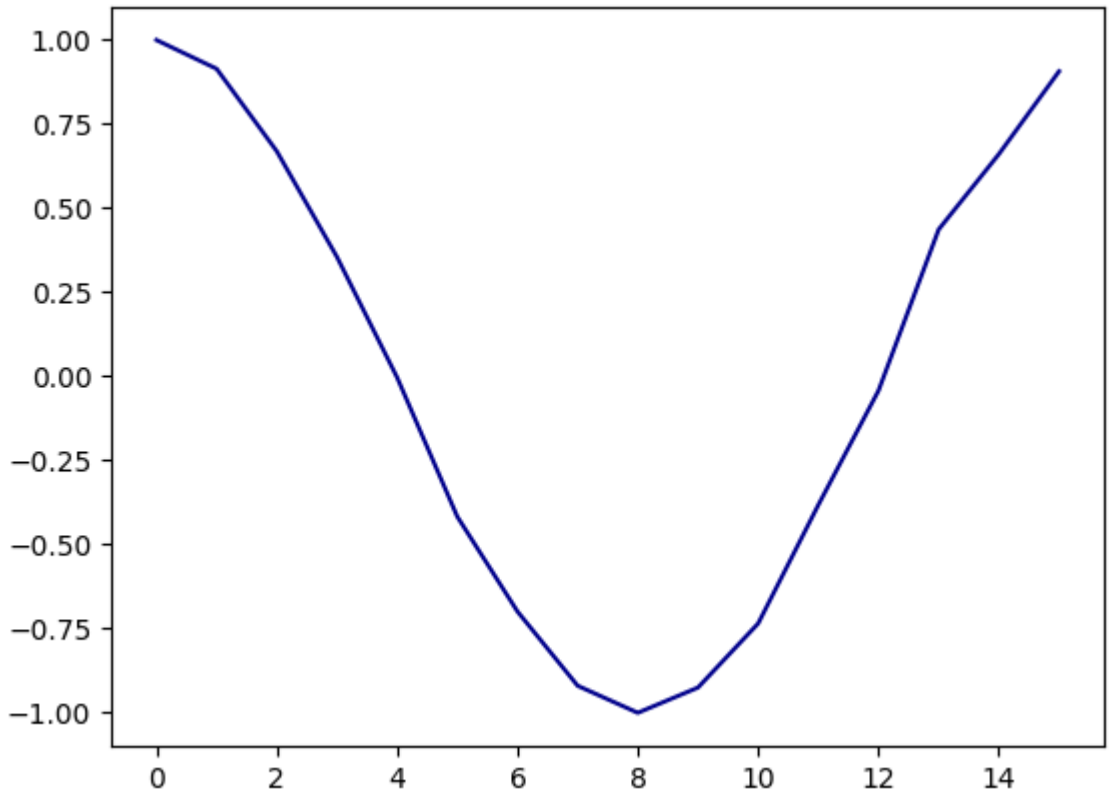
[11]:
```
synth = Synth(server, "qTable", {"buf" : b, "gain" : -25, "freq" : 250})
time.sleep(1)
synth.free()
```

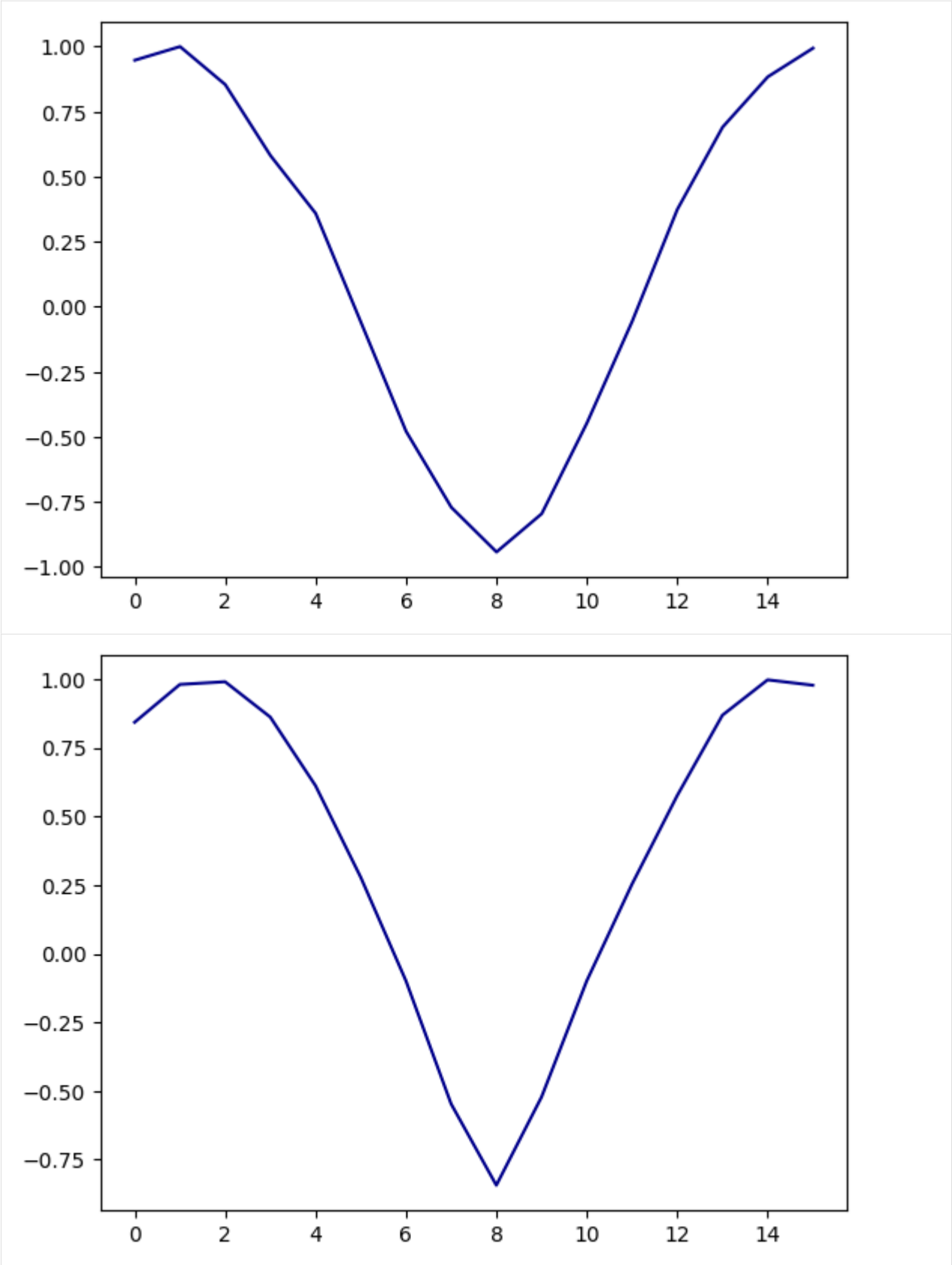
Now, let's apply incremental **:math:`R_y`** rotations to our audio to see how the signal changes. If you restart the `synth` again, you will be able to listen to the changes.

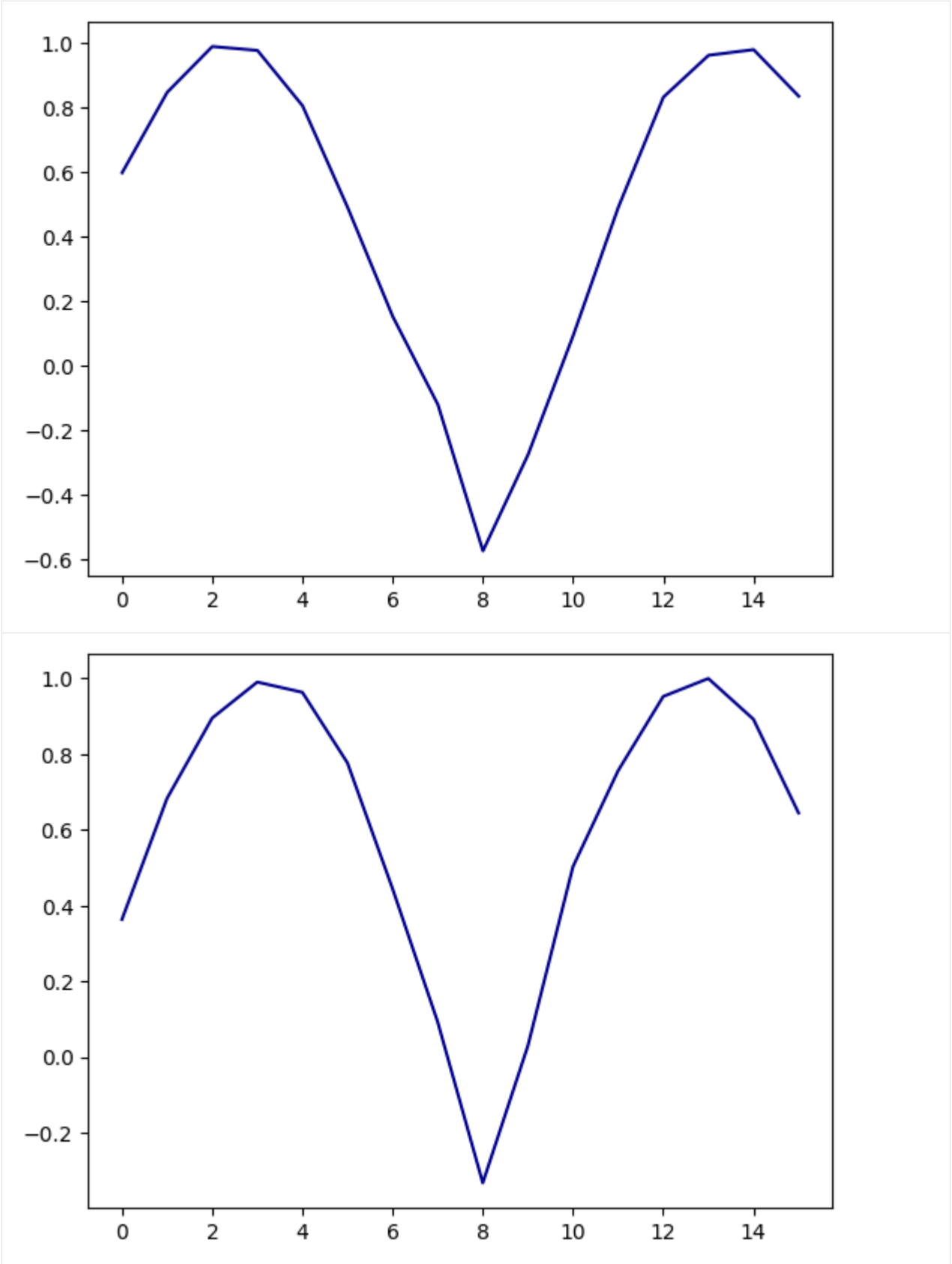
In this case, we will be performing 10 simulations, rotating the SQPAM audio by $\pi/10$ each time.

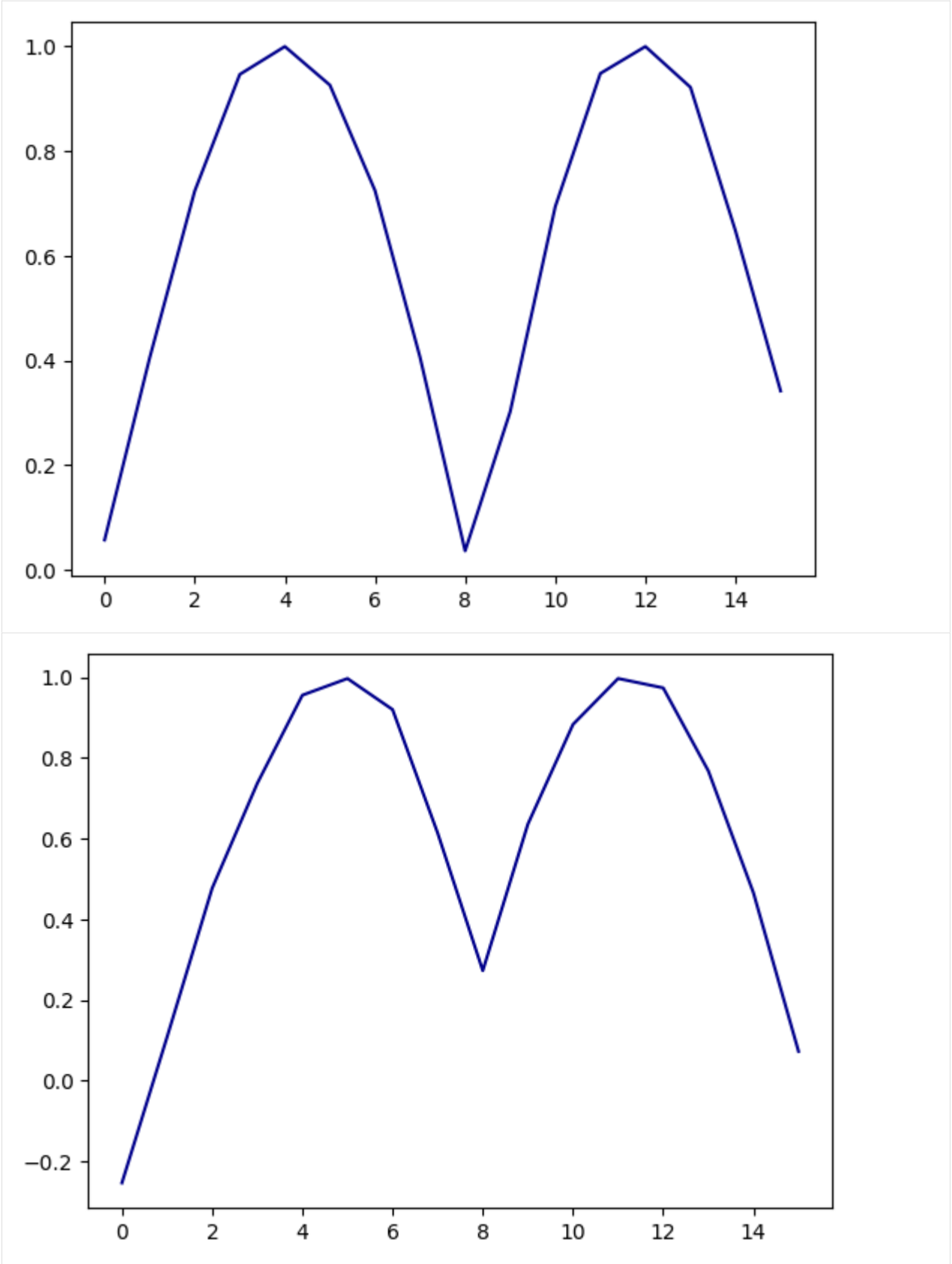
[12]:
```
synth = Synth(server, "qTable", {"buf" : b, "gain" : -25, "freq" : 250})
```

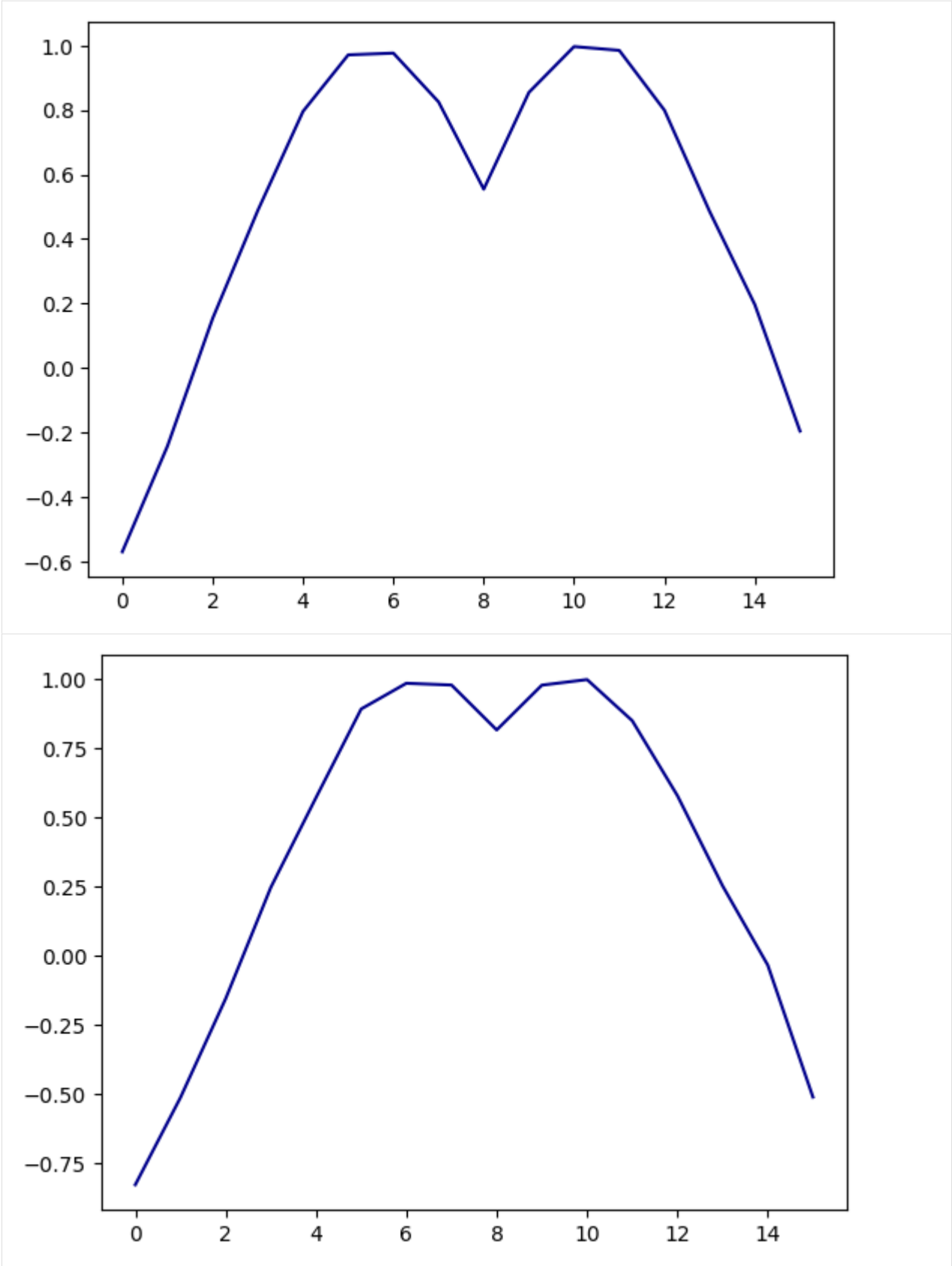
[13]:
```
for i in [x*pi/10 for x in range(11)]:
    out = add_ry_sqpam(qsine, i)
    updateBuffer(b, out)
    plot_out(out)
```

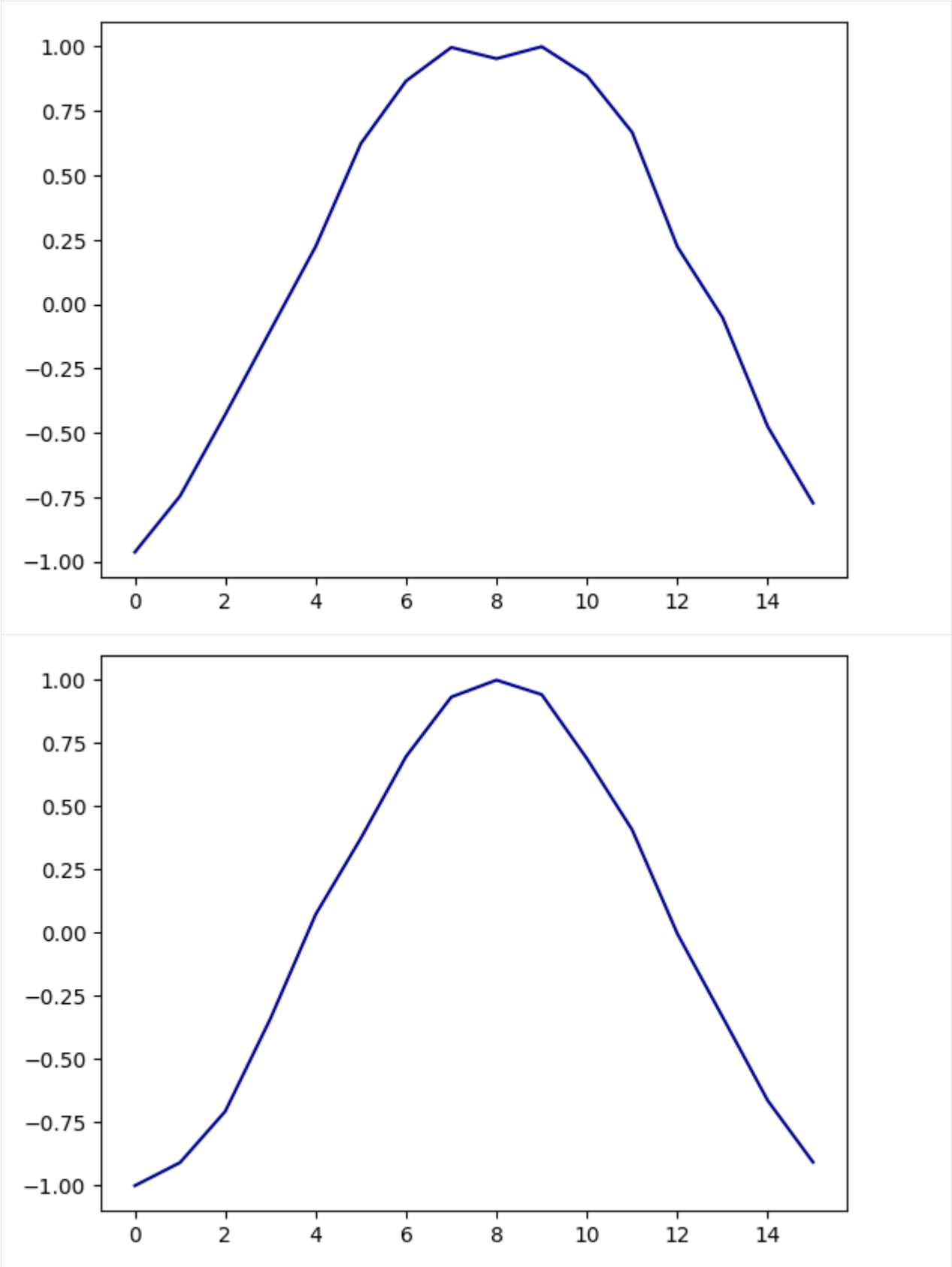



1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
2
4
6
8
10
12
14


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0
2
4
6
8
10
12
14
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
2
4
6
8
10
12
14


1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
12
14
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
2
4
6
8
10
12
14


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
2
4
6
8
10
12
14


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14

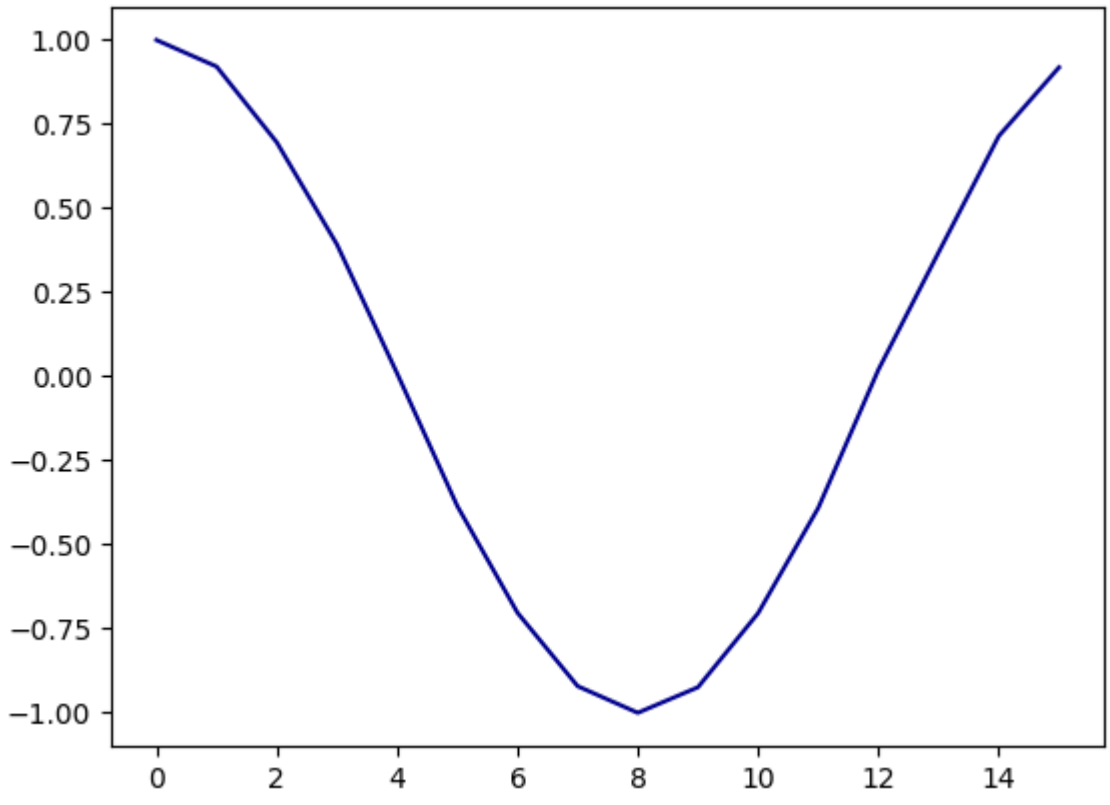
[14]:
```
synth.free()
```

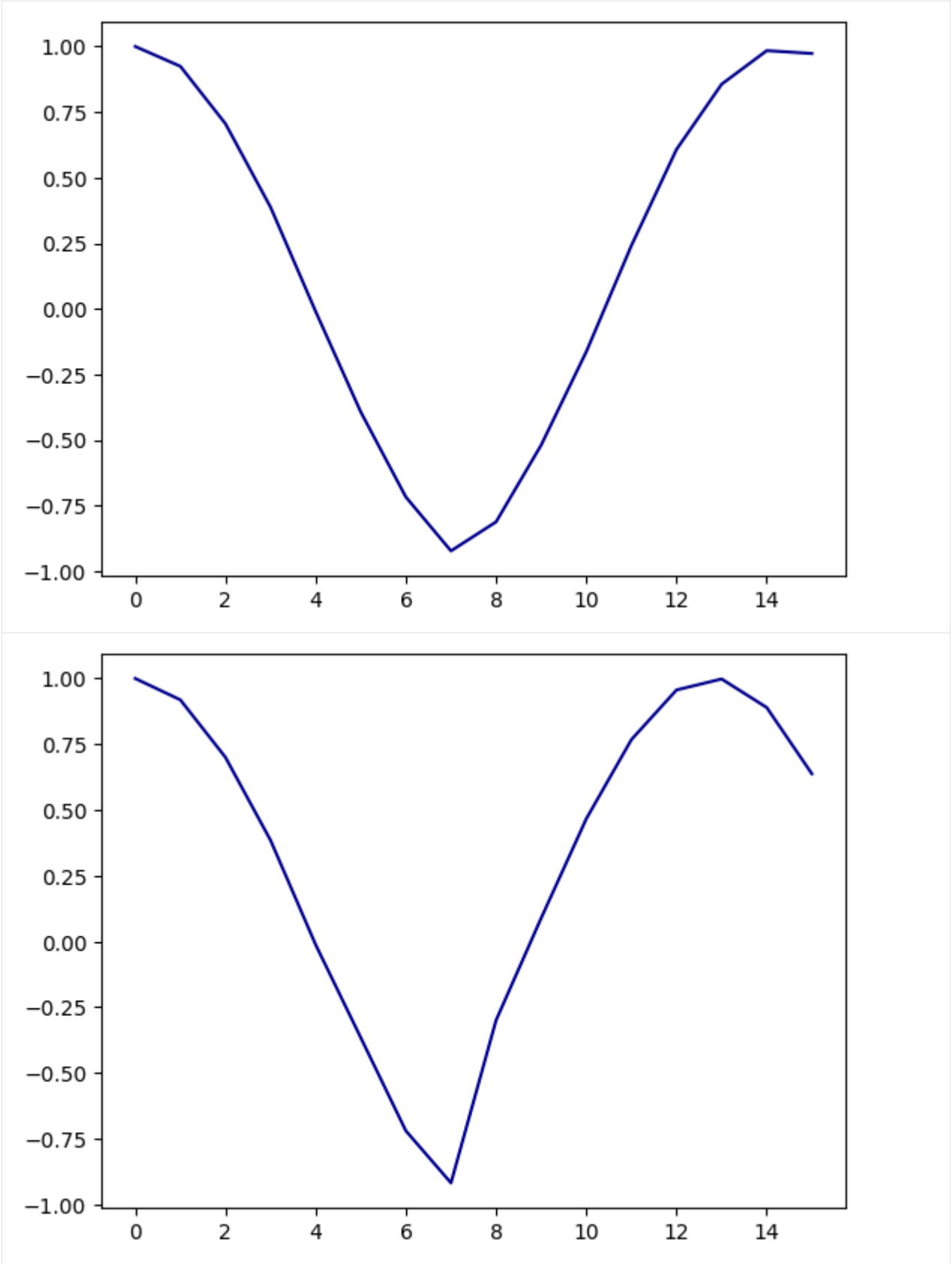
By using controlled rotations, it is possible to apply these distortions only to a segment of the audio file. In this example, we are rotating only the second half of the signal by using the *Most Significant Qubit* as control condition

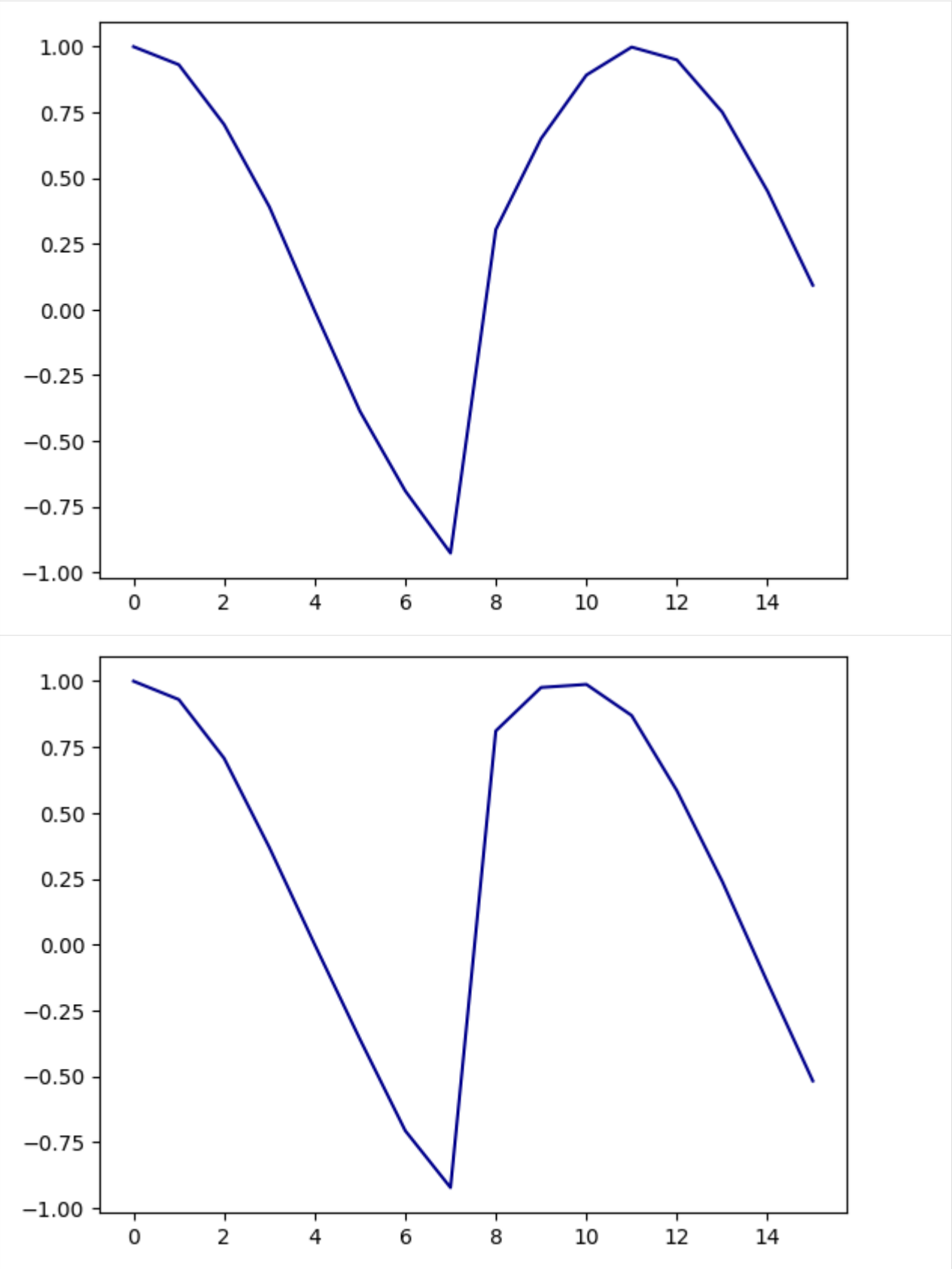
[15]:
```
def add_cry_sqpam(qa, angle, control_qubit, shots=100000):
    qa.prepare()
    cry=QuantumCircuit(1)
    cry.ry(angle,0)
    qa.circuit.append(cry.to_gate().control(1), [control_qubit, 0])
    qa.measure()
    qa.circuit.draw()
    qa.run(100000).reconstruct_audio()
    return qa.output
```

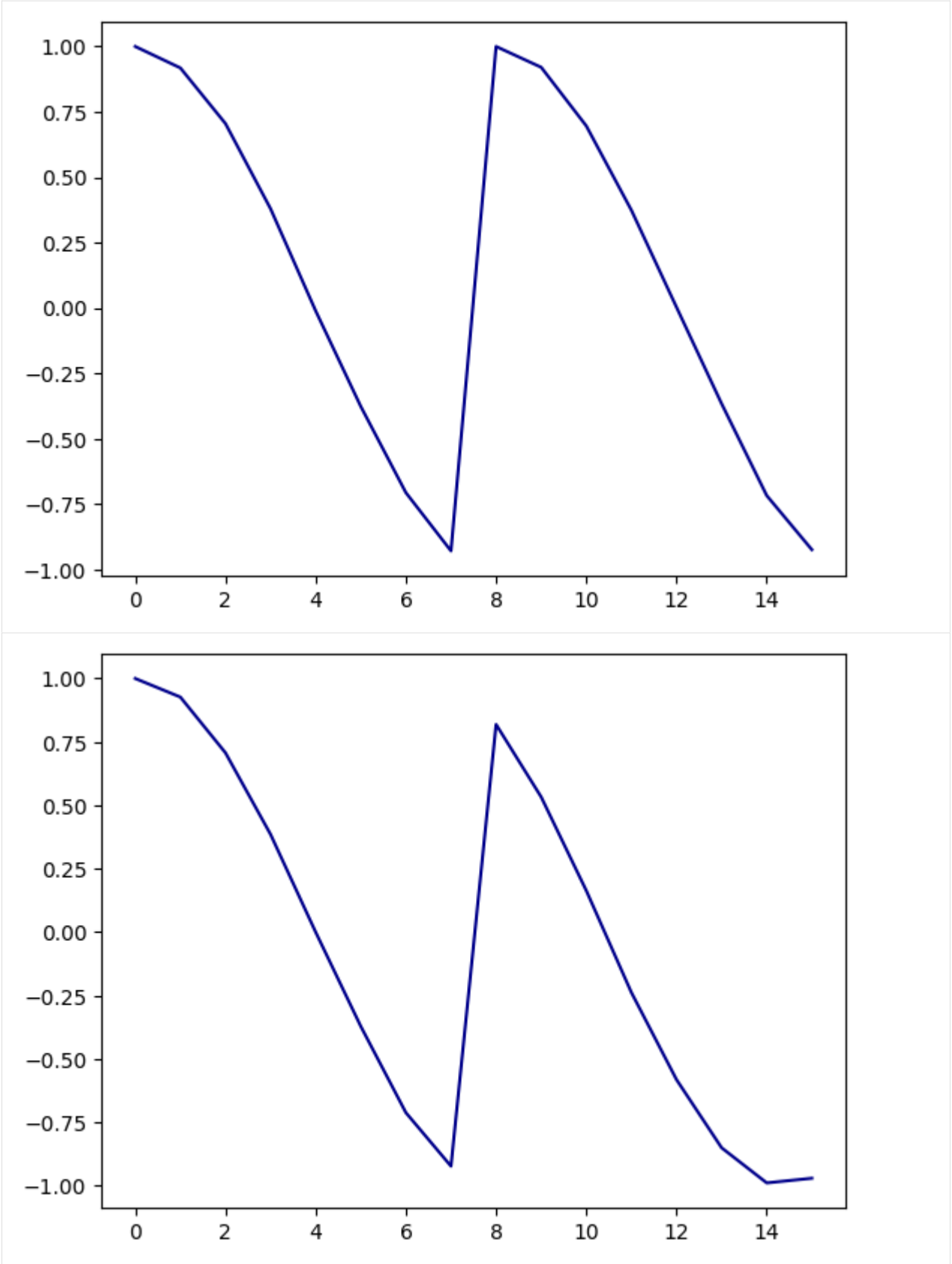
[16]:
```
synth = Synth(server, "qTable", {"buf" : b, "gain" : -25, "freq" : 250})
```

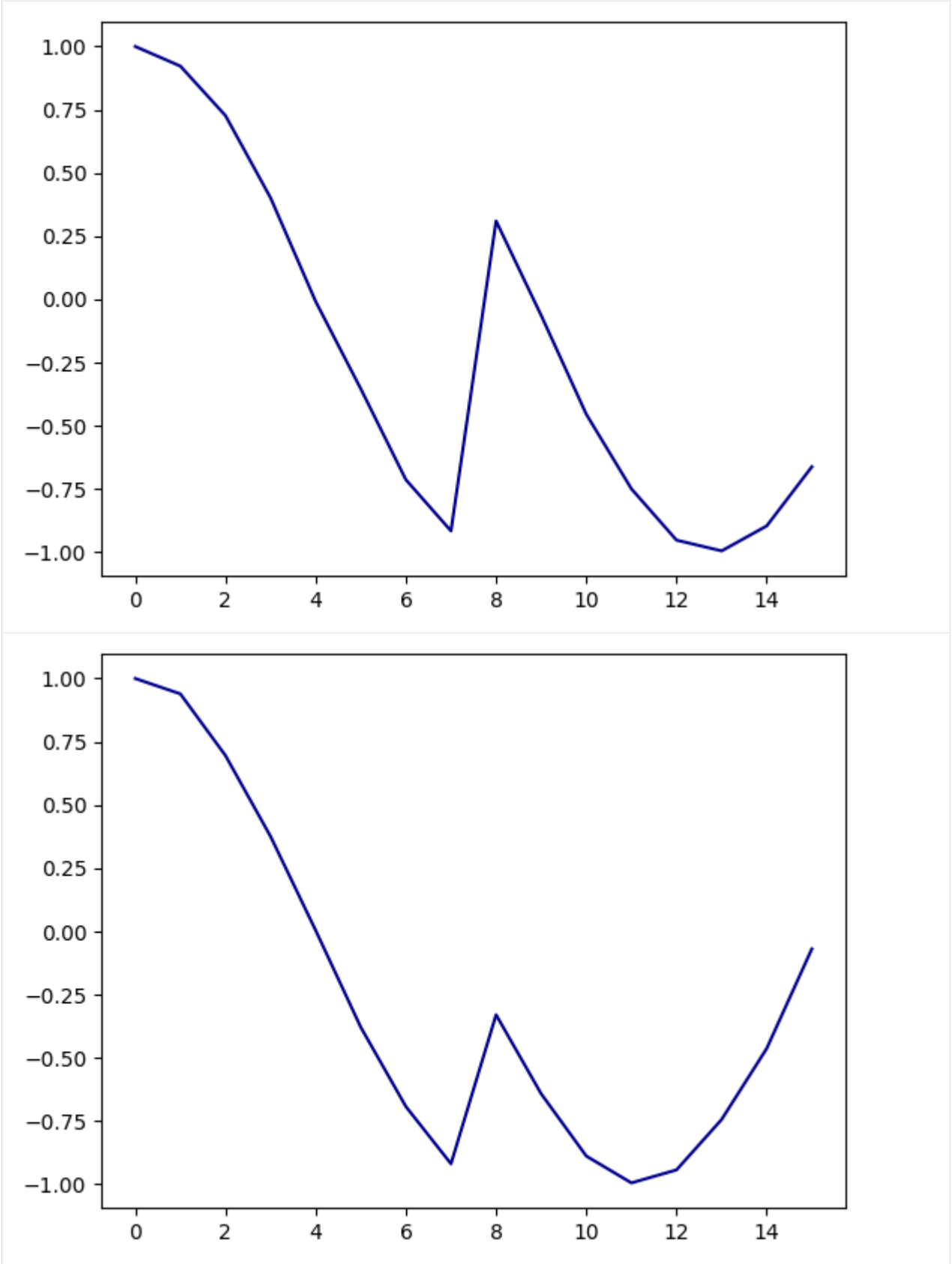
[17]:
```
for i in [x*pi/5 for x in range(11)]:
    out = add_cry_sqpam(qsine, i, 4)
    updateBuffer(b, out)
    plot_out(out)
```

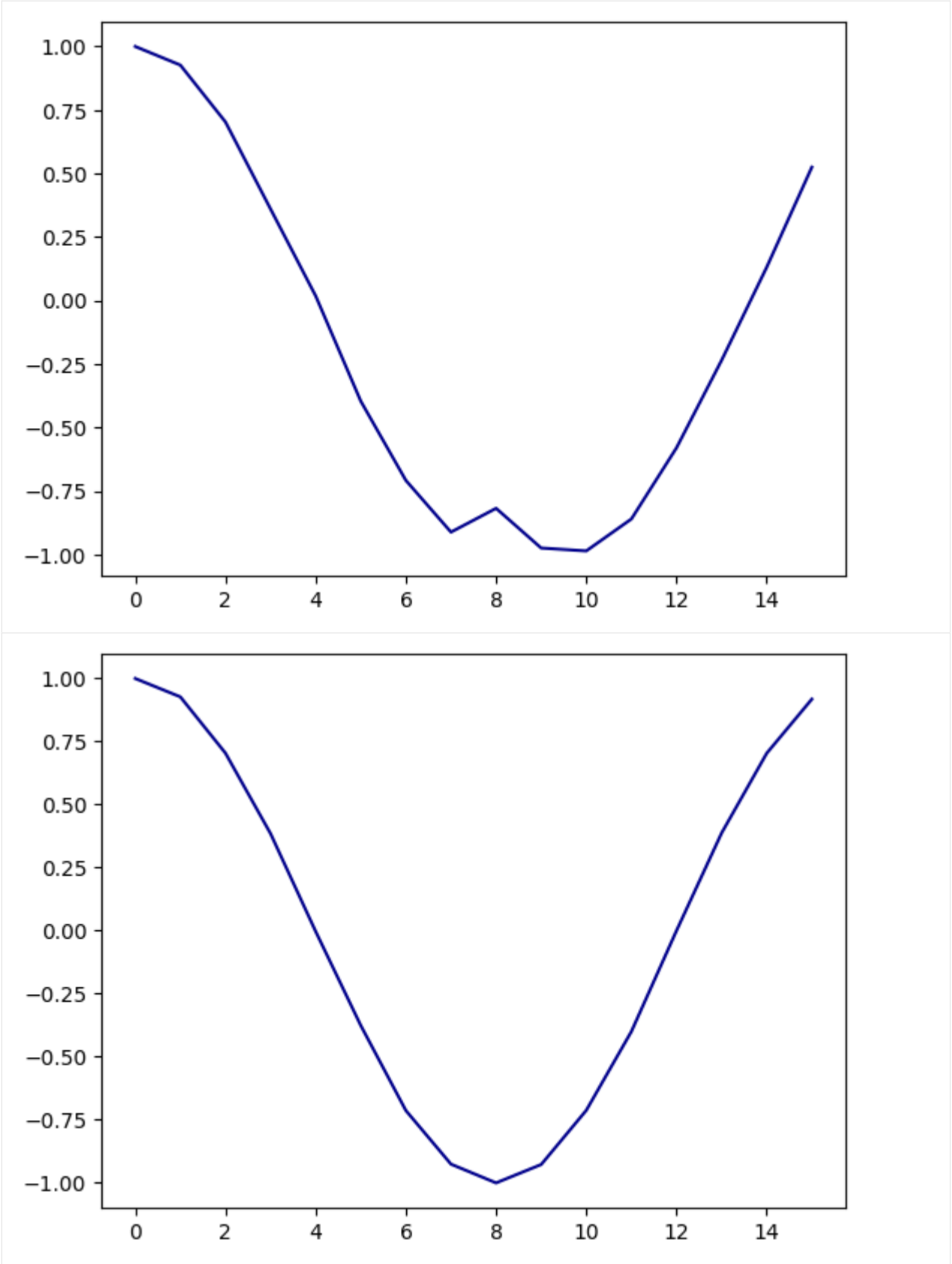



1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14

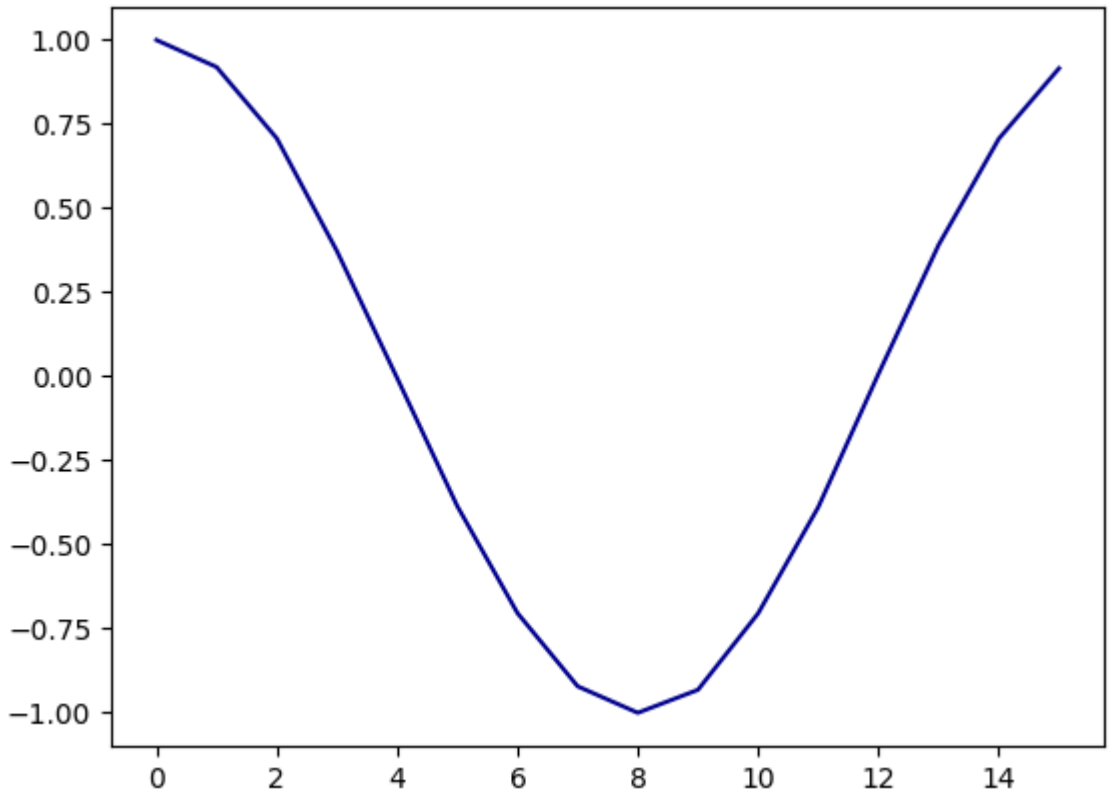

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14

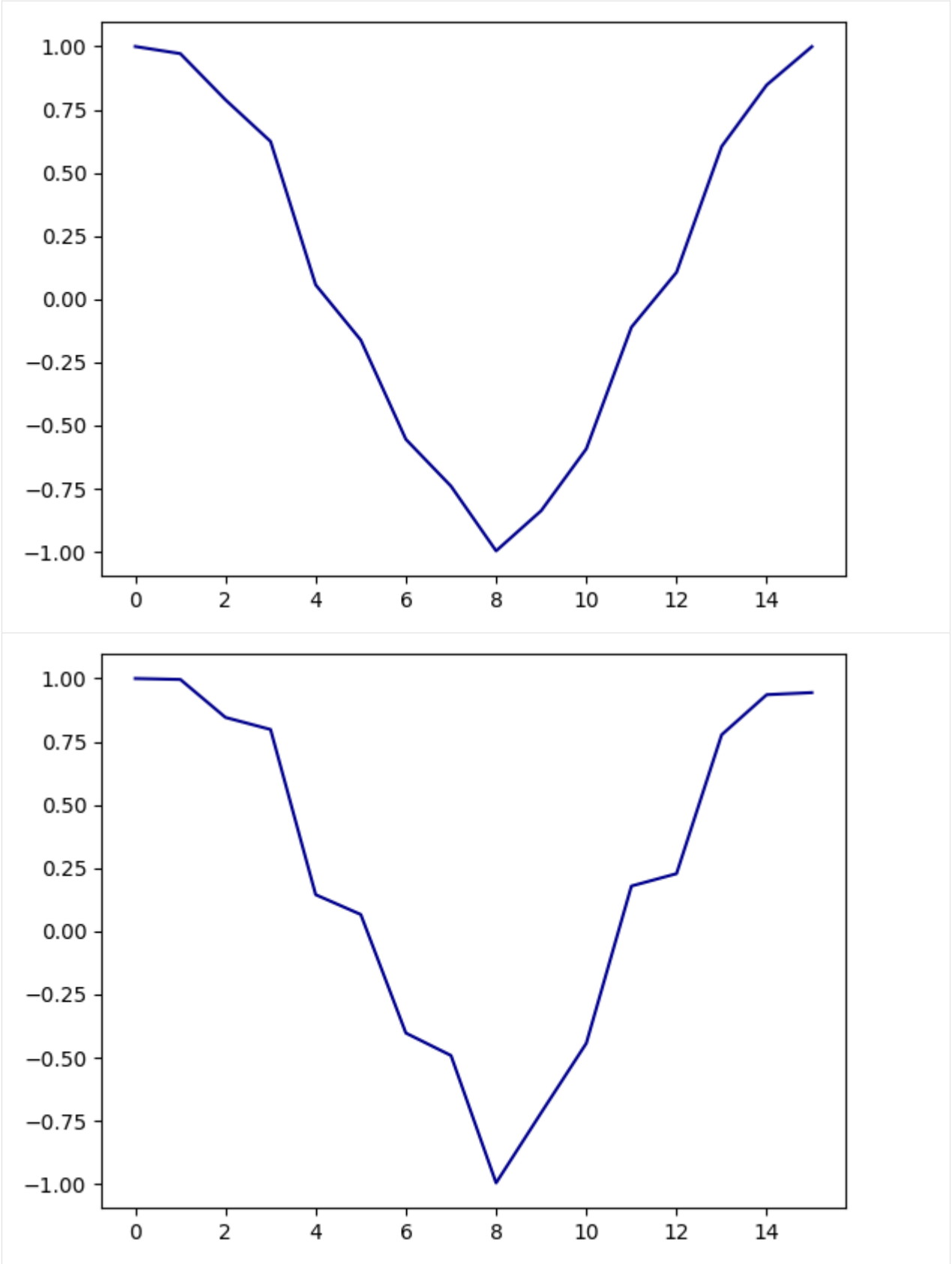

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14

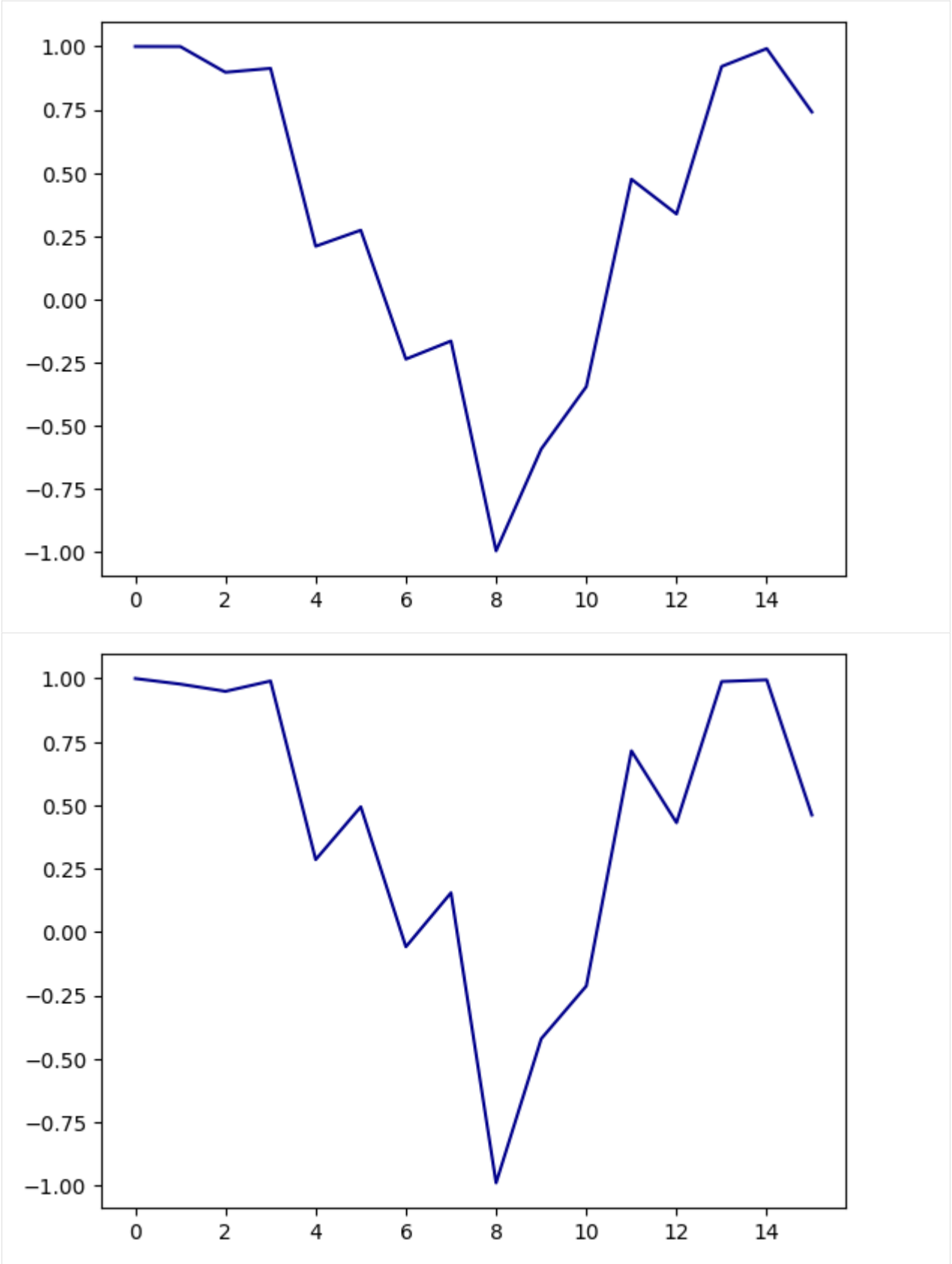
Feel free to further extrapolate and explore this idea!

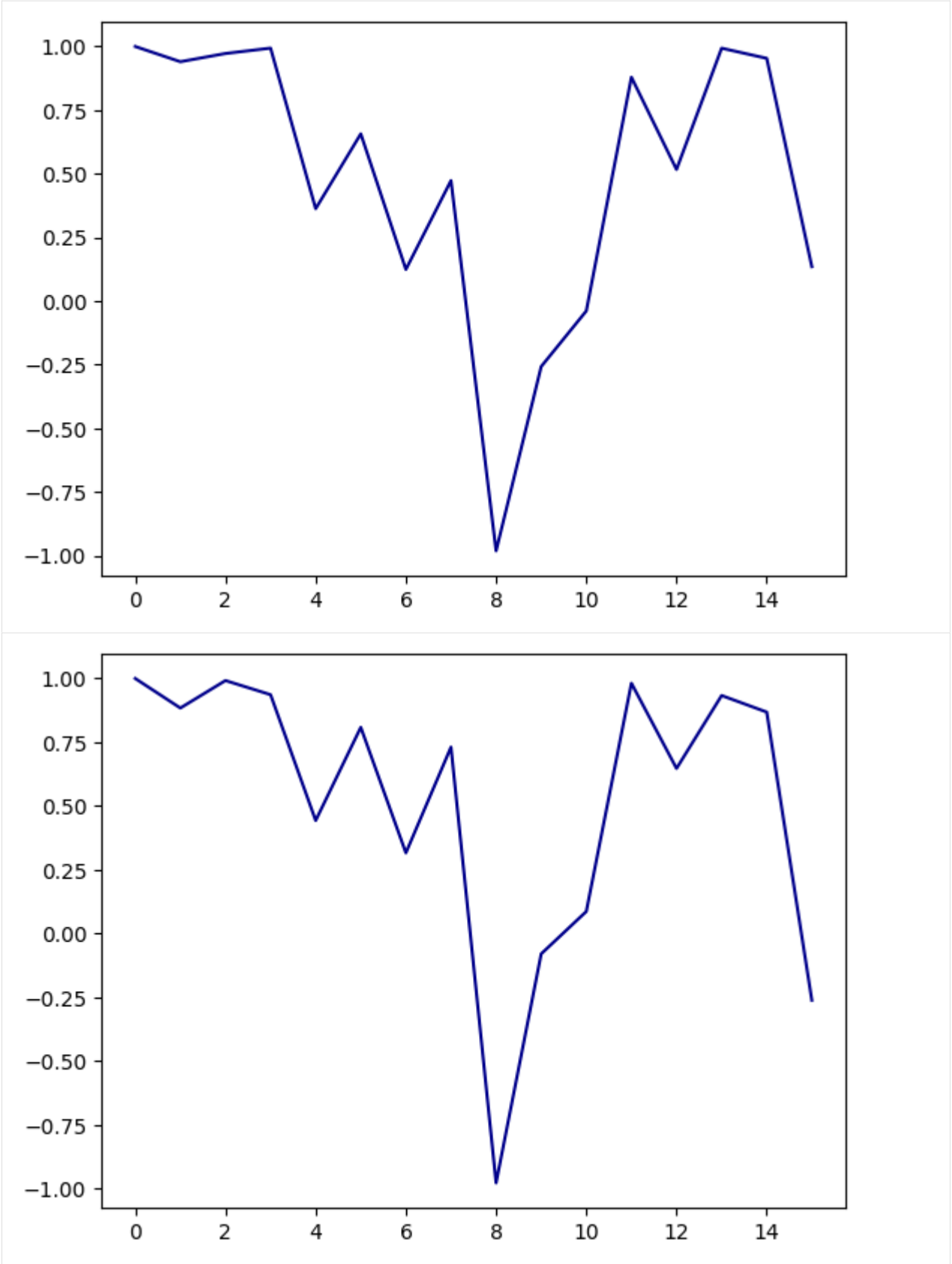
[18]:
```
def add_mcry_sqpam(qa, angle, shots=100000):
    qa.prepare()
    cry=QuantumCircuit(1)

    # Each qubit controls a rotation that is faster compared to the previous one
    cry.ry(angle,0)
    qa.circuit.append(cry.to_gate().control(1), [4, 0])
    cry.ry(angle,0)
    qa.circuit.append(cry.to_gate().control(1), [3, 0])
    cry.ry(angle,0)
    qa.circuit.append(cry.to_gate().control(1), [2, 0])
    cry.ry(angle,0)
    qa.circuit.append(cry.to_gate().control(1), [1, 0])


    qa.measure()
    qa.circuit.draw()
    qa.run(100000).reconstruct_audio()
    return qa.output
```

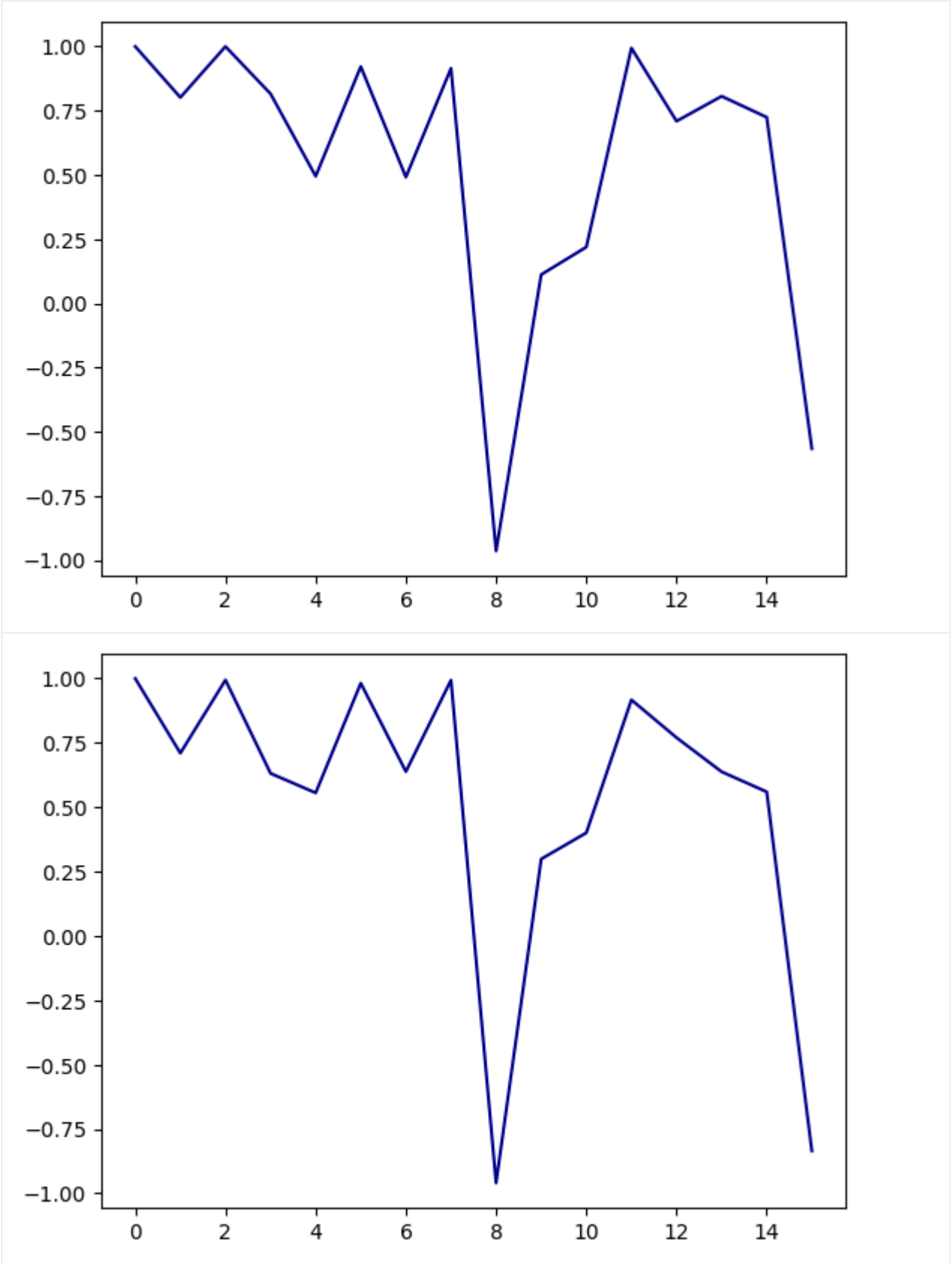
[19]:
```
for i in [x*pi/85 for x in range(20)]:
    out = add_mcry_sqpam(qsine, i)
    updateBuffer(b, out)
    plot_out(out)
```

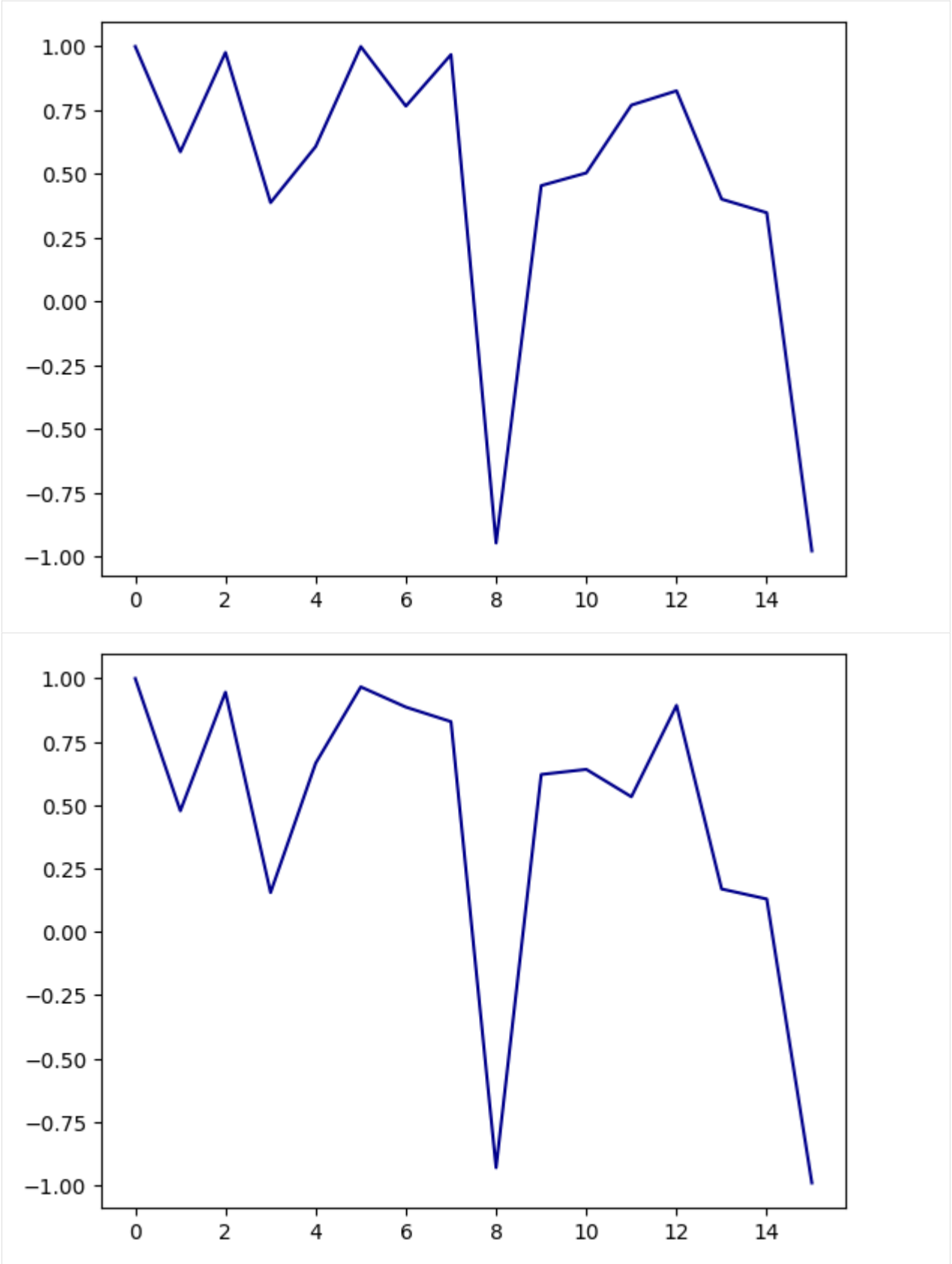


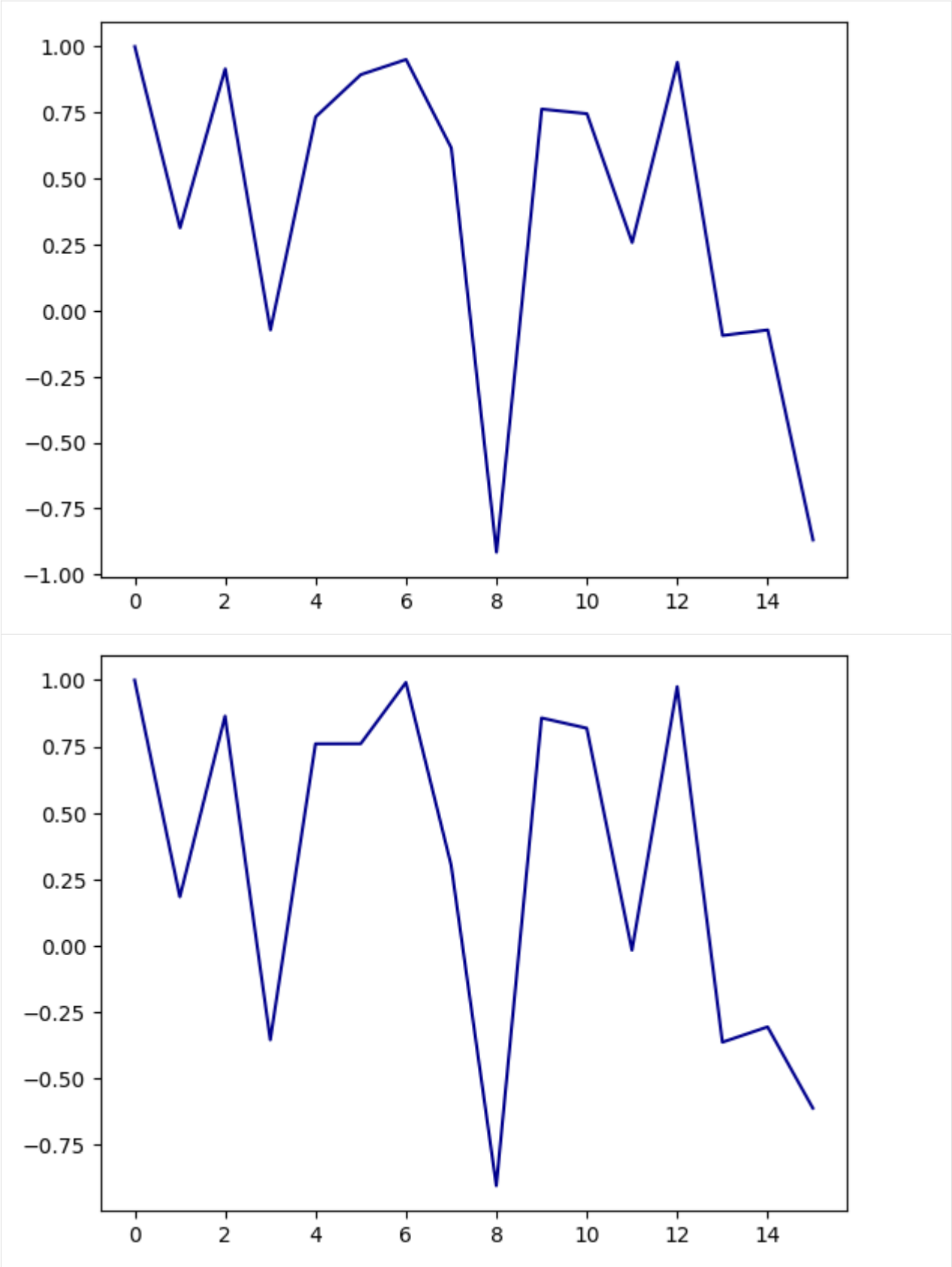

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14

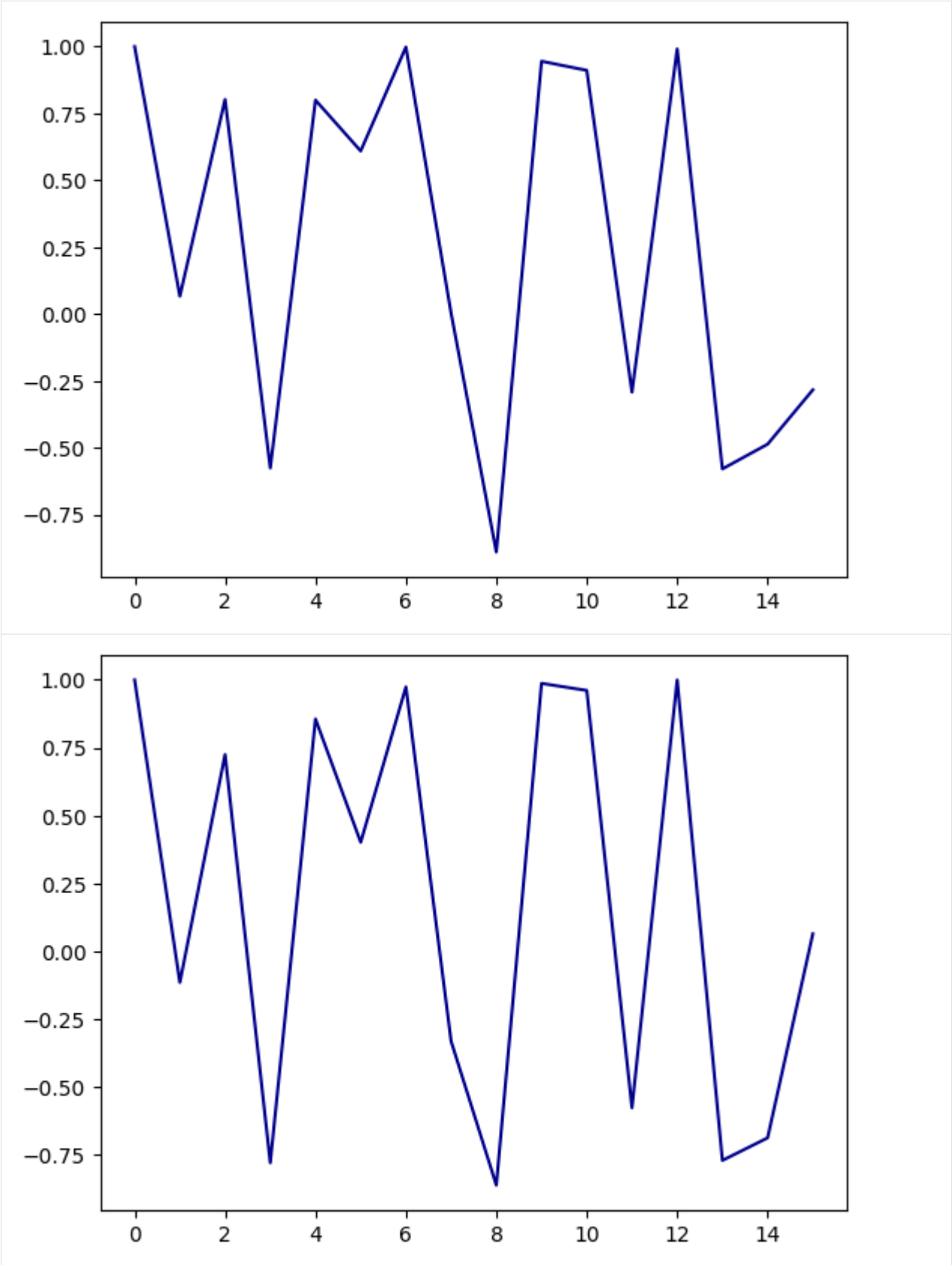

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14

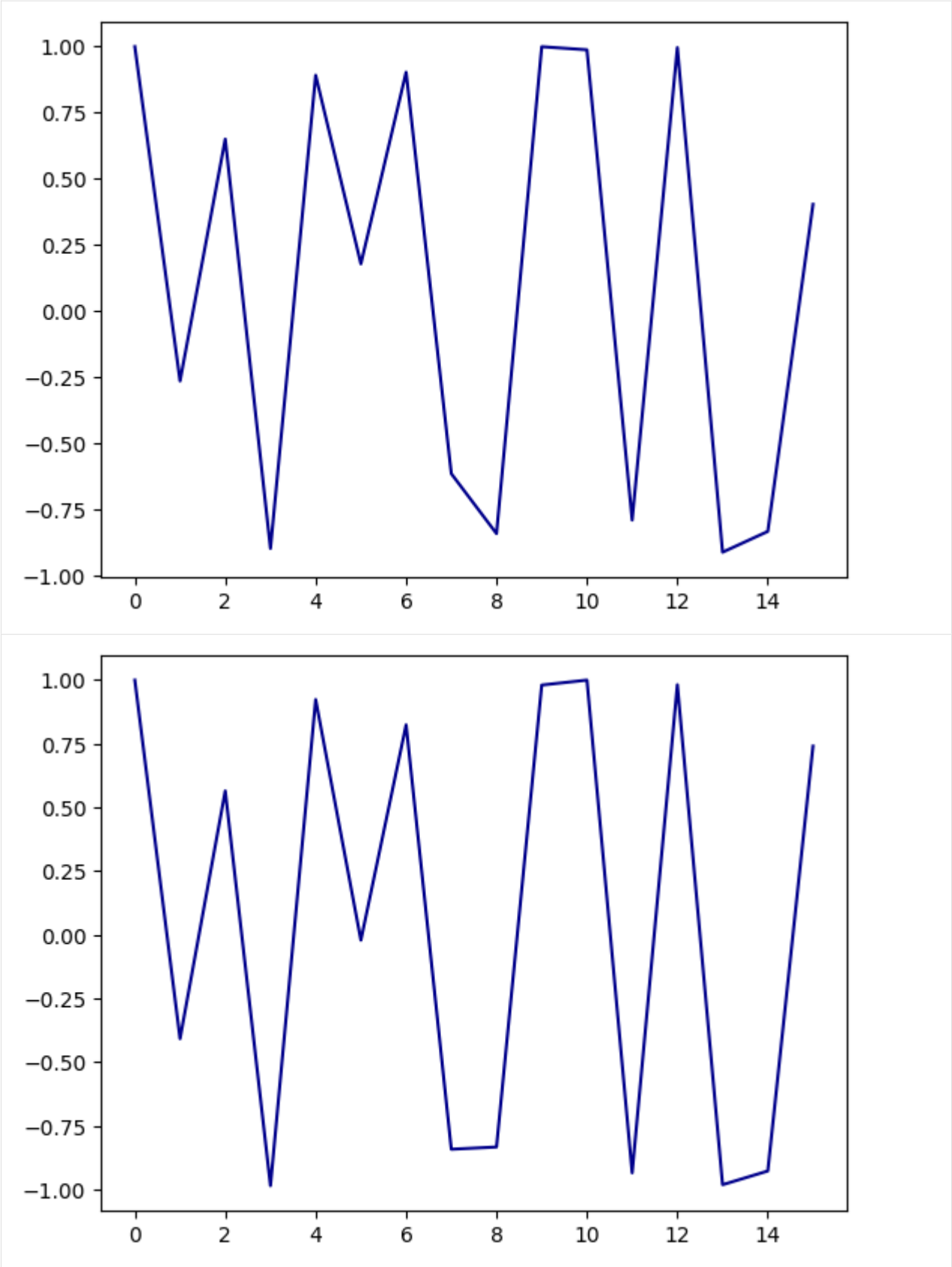

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14

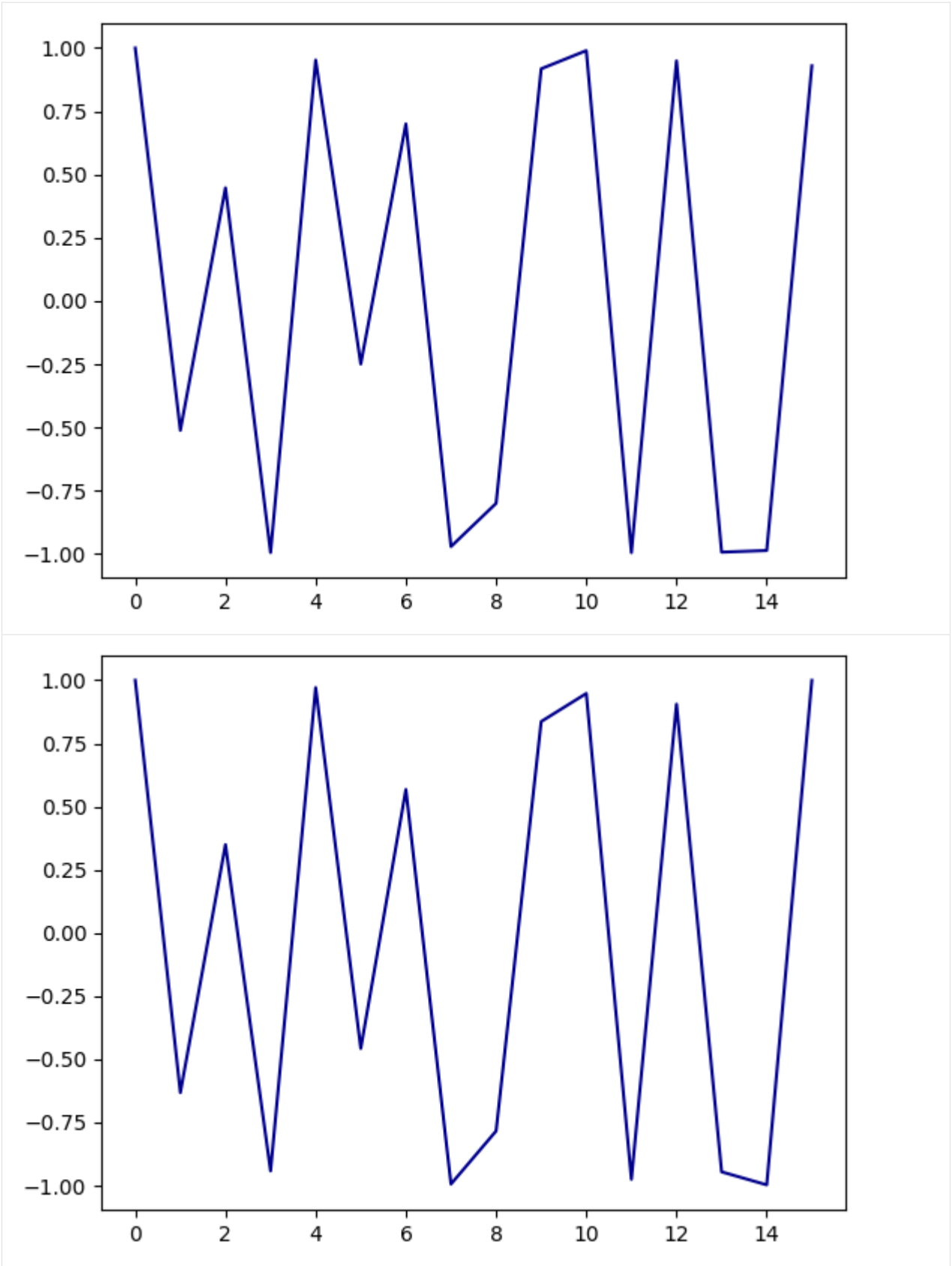

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14

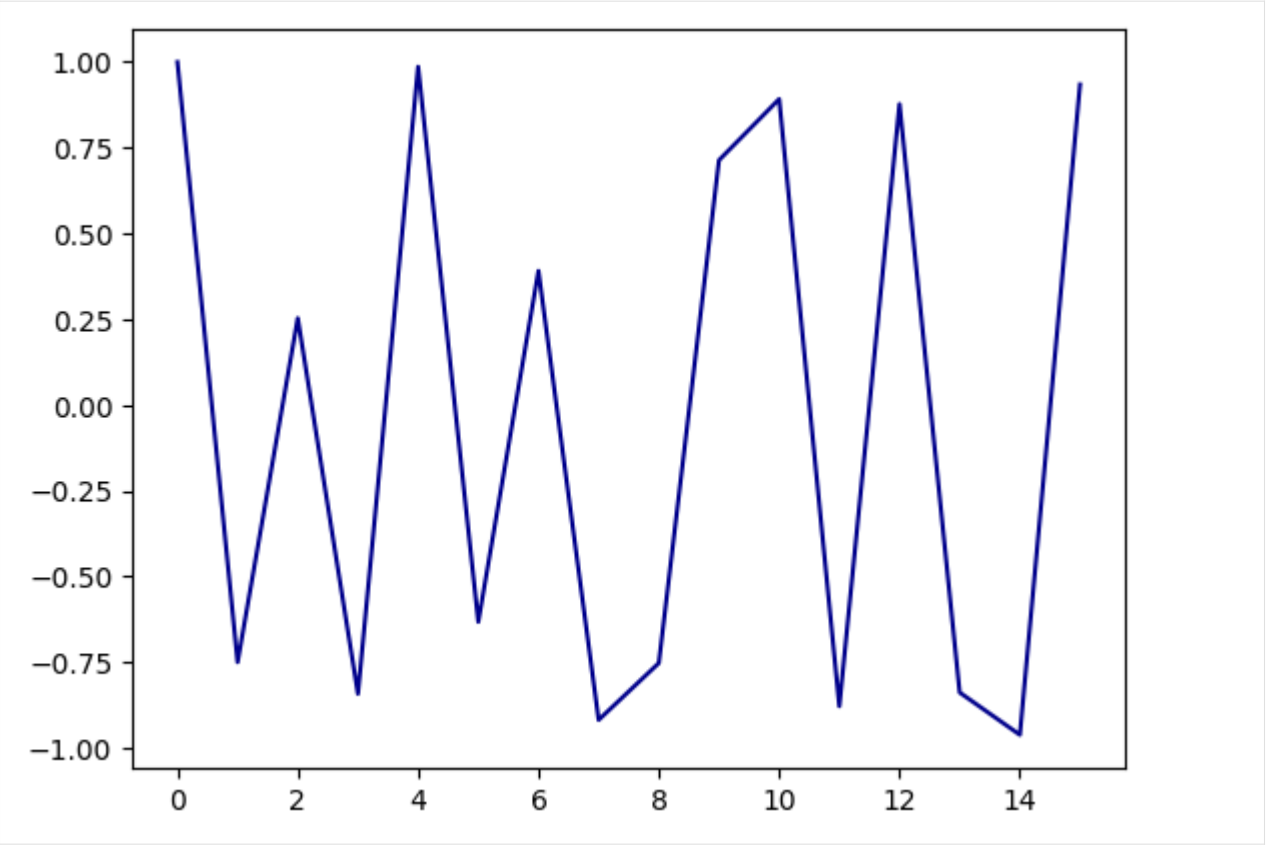

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
2
4
6
8
10
12
14


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
2
4
6
8
10
12
14


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14



[20]:
```
synth.free()
```

.

.

Download this notebook from the latest Github release.

Itaborala @ ICCMR Quantum https://github.com/iccmr-quantum/quantumaudio

CHAPTER

FIVE

LICENSE

PYTHON MODULE INDEX

q

INDEX