OQCL

OPEN QUANTUM COMPUTING LAB

Student Manual

Where to start, how to run and submit a project, and a seven-session path through quantum computing with worked examples and experiments.

PART 1 — GETTING STARTED

1. Start here
2. Guide 1 — Your first project, step by step
3. Guide 2 — Writing README.md and REPORT.md
4. Guide 3 — Submitting with GitHub

PART 2 — QUANTUM IN 7 SESSIONS

1. Qubits and amplitudes
2. Measurement, gates, and interference
3. Entanglement
4. Real hardware: compilation and noise
5. Algorithms: search and phase estimation
6. Near-term methods: Hamiltonians and variational circuits
7. Communication, benchmarking, and your own project

Ali M. Saghiri · William Paterson University

Online version: amsaghiri.github.io/open-quantum-computing-lab/start-here.html
November 2026

Start here

New to OQCL? You do not need to read everything. Follow the six steps below in order. Your first hour is: read Chapter 1, then run the Quantum Coin experiment.

What to do, in order

1. **Read Chapter 1.** It gives you the language used everywhere else: amplitudes, probabilities, and phase. Open Chapter 1 →
2. **Run your first experiment, Q01 Quantum Coin.** It takes 15–25 minutes on a simulator and needs no hardware. Follow the step-by-step guide →
3. **Choose your project.** Open the project ladder and pick the smallest project that extends what you already understand. Each project page names the chapter to read first.
4. **Apply by email before graded or QPU work.** Send the project ID, project title, your name, and a short plan to saghiria@wpunj.edu, then wait for instructor approval.
5. **Do the experiment the OQCL way.** Write a prediction, reproduce the baseline on a simulator, change exactly one variable for your extension, and analyze the result with a number, not only a picture.
6. **Submit.** Put your notebook, raw data, figures, and report in one folder and open a Pull Request. GitHub submission guide →

Want a fixed study order? Quantum in 7 Sessions takes you through the textbook one idea at a time, with a small experiment and a project in each session.

Student guides

Four short manuals. Read them when you reach that step; you do not need them all on day one. Start Here, Guides 1–3, and all seven sessions are also available together as one printable file: Student Manual (PDF).

Guide 1 · 30 min

Your first project, step by step

A complete walkthrough of Q01 Quantum Coin: prediction, working code, raw data, a figure, and an extension.

Guide 2 · 15 min

Writing README.md and REPORT.md

Copy-ready templates and what a good answer looks like in each section.

Guide 3 · 20 min

Submitting with GitHub

Fork, create your folder, upload, and open a Pull Request, even if you have never used GitHub.

Guide 4 · when approved

Using real hardware (QUEST Lab path)

For projects with a QPU part: simulator baseline first, then an approved hardware run, then analysis back in OQCL.

Which chapter goes with which project

Read the chapter before starting any project in its row. If you are new to quantum computing, read the chapters in order.

READ	TOPIC	THEN YOU CAN DO
Chapter 1	Mathematical Language of Quantum Computing	Q01 Quantum Coin · Q02 Amplitude and Phase Explorer
Chapter 2	Qubits, Measurement, and Single-Qubit Gates	Q03 X, H, and Interference · Q08 Shot-Count Convergence
Chapter 3	Multiple Qubits, Tensor Products, and Entanglement	Q04 Bell Correlations · Q05 Bell State Under Noise · Q06 GHZ Scaling · Q07 GHZ: Simulator vs QPU
Chapter 4	Circuits, Compilation, and Real Hardware	Q09 Physical Qubit Mapping · Q10 Compilation Tradeoffs · Q23 Cost-Aware Hardware Plan
Chapter 5	Interference and Quantum Search	Q11 Grover Search: Four States · Q12 Grover Under Noise
Chapter 6	Phase, Eigenvalues, and Phase Estimation	Q13 Phase Estimation
Chapter 7	Noise, Error Detection, and Quantum Error Correction	Q14 Syndrome Decoding · Q15 QEC Resource Feasibility

Chapter 8	Hamiltonians, Dynamics, and Trotterization	Q16 Ising Trotter Study · Q17 Trotter Error vs Noise
Chapter 9	Variational and Hybrid Quantum Algorithms	Q18 VQE Energy Estimation · Q19 QAOA Max-Cut
Chapter 10	Quantum Machine Learning	Q20 Variational Quantum Classifier
Chapter 11	Quantum Communication and BB84	Q21 BB84 and Eavesdropping
Chapter 12	Benchmarking and Reproducible Quantum Experiments	Q22 Reproducible Processor Benchmark · Q24 Reproduce, Extend, Publish

Where to find things

PAGE	USE IT WHEN YOU WANT TO
Textbook	Read the 12 chapters, each with a worked example and self-check questions.
Projects	See all 24 projects with level, time, and whether hardware is involved.
QUEST Lab	Plan and run the hardware part of a project after approval.
Submit	Check the submission workflow and the required folder layout.
Use Cases	See what each topic is used for, and what it does not prove.
References	Find the external sources and further reading.

Common questions

Do I need a real quantum computer?

No. Most projects run on a simulator. Hardware is optional unless your instructor explicitly requires it, and any QPU run needs approval first.

Which software should I use?

Any suitable quantum simulator or computing environment. OQCL is platform-neutral. Guide 1 shows one working option so you are not stuck choosing.

Do I have to finish all 12 chapters before a project?

No. Read the prerequisite chapter listed on your project page. Read in order only if the earlier material is new to you.

What if my prediction turns out wrong?

That is acceptable. A failed prediction is fine when the experiment is reproducible and the analysis is sound. Explain what you expected, what happened, and why they differ.

How much time will a project take?

From 15–25 minutes for the first micro-experiment to 5–10 hours for the capstone. The project ladder lists the time for each one.

Who do I ask when I am stuck?

Email your instructor at saghiria@wpunj.edu with the project ID and a short description of the problem.

Guide 1 — Your first project, step by step

About 30 minutes | Simulator only

This guide walks through Q01 — Quantum Coin from an empty folder to a finished experiment. Every later project follows the same routine, so learning it once here saves time on all of them.

About the tool. OQCL is platform-neutral and you may use any suitable simulator. The code below uses Qiskit with the Aer simulator as one working option. It was tested with Qiskit 2.6 and Qiskit Aer 0.18.

Step 1 — Read first

Read Chapter 1. You need one idea from it: a qubit prepared with a Hadamard gate, $H|0\rangle$, has probability ½ of giving 0 and ½ of giving 1 when measured.

Step 2 — Write your prediction before any code

Open a text file and answer these three questions. This is the part students skip, and it is the part that makes the work scientific.

1. **State.** $H|0\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$.
2. **Prediction.** $P(0)$ will be close to 0.5, and it will get closer as the number of shots grows.
3. **Metric.** The distance from the ideal value, $|P(0) - 0.5|$, for 10, 100, 1,000, and 10,000 shots.

Step 3 — Create your project folder

Use your own GitHub username in the folder name.

```
projects/q01-yourgithubname/
  README.md
  notebook.ipynb
  results/raw/
  results/figures/
  environment/requirements.txt
  REPORT.md
```

Step 4 — Set up the software

You need Python 3. In a terminal, or in the first cell of a notebook with `!` in front, run:

```
pip install qiskit qiskit-aer matplotlib
```

Step 5 — Run the baseline

Put this in `notebook.ipynb` and run it. It builds the circuit, runs it for each shot count, saves the raw counts, and draws one figure.

```
import json, os
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator
import matplotlib.pyplot as plt

# 1. Build the circuit: H on |0>, then measure
qc = QuantumCircuit(1, 1)
qc.h(0)
qc.measure(0, 0)

# 2. Run the baseline for each shot count
sim = AerSimulator(seed_simulator=7)
results = []
for shots in [10, 100, 1000, 10000]:
    counts = sim.run(qc, shots=shots).result().get_counts()
    p0 = counts.get("0", 0) / shots
    p1 = counts.get("1", 0) / shots
    results.append({"shots": shots, "counts": counts,
                    "P0": p0, "P1": p1, "abs_error": abs(p0 - 0.5)})
    print(shots, counts, round(p0, 4), round(abs(p0 - 0.5), 4))

# 3. Save the raw data (never only a screenshot)
os.makedirs("results/raw", exist_ok=True)
os.makedirs("results/figures", exist_ok=True)
with open("results/raw/q01_counts.json", "w") as f:
    json.dump(results, f, indent=2)

# 4. One labeled figure
plt.plot([r["shots"] for r in results], [r["abs_error"] for r in results], "o-")
plt.xscale("log")
plt.xlabel("Number of shots")
plt.ylabel("|P(0) - 0.5|")
plt.title("Q01: sampling error vs. shots")
plt.savefig("results/figures/q01_error_vs_shots.png", dpi=150, bbox_inches="tight")
```

Step 6 — Check what you got

You should see four lines, one per shot count. Your numbers will differ from these because measurement is random, but the pattern should be the same: the error gets smaller as shots increase.

SHOTS	EXAMPLE COUNTS	P(0)	|P(0) − 0.5|
10	0: 4, 1: 6	0.40	0.10
100	0: 54, 1: 46	0.54	0.04
1,000	0: 514, 1: 486	0.514	0.014
10,000	0: 5012, 1: 4988	0.5012	0.0012

Two files should now exist: `results/raw/q01_counts.json` and `results/figures/q01_error_vs_shots.png`. If they do, your baseline is done.

Step 7 — Add one controlled extension

Change exactly one thing and keep everything else fixed. Write a prediction first. Two ideas that fit Q01:

- **Repeat the experiment.** Run each shot count 20 times and look at how much P(0) varies between runs. Prediction to test: the spread shrinks as shots grow.
- **Change the state.** Replace `qc.h(0)` with `qc.x(0)` and predict the counts before you run it.

Step 8 — Record your environment

So that someone else can reproduce your result, save the exact package versions:

```
pip freeze > environment/requirements.txt
```

Step 9 — Write it up

Fill in `README.md` and `REPORT.md`. Guide 2 gives you templates for both.

Before you move on, check

- The prediction was written before the code was run.
- Raw counts are saved as a file, not only shown on screen.
- The metric is a number.
- The figure has axis labels and a title.
- No passwords, tokens, or API keys are anywhere in the folder.

Guide 2 — Writing README.md and REPORT.md

About 15 minutes to read | Templates included

Every project needs two short text files. The README lets someone else rerun your work. The REPORT explains what the result means. Copy the templates below and fill them in.

Where to get the files. Both templates are in the templates folder of the repository, ready to copy into your project folder.

README.md — so others can reproduce your work

Write the Question, Hypothesis, and Metric sections before you run anything. Fill in the rest when you finish.

```
# Qxx — Project title

**Student / team:**
**GitHub username(s):**
**Date:**

## Question
What scientific question does this experiment answer? (1–2 sentences)

## Hypothesis
What do you predict will happen, and why? Write this BEFORE you run anything.

## Metric
What number will you compute to decide whether the prediction was supported?

## How to reproduce
1. Install the dependencies: `pip install -r environment/requirements.txt`
2. Open `notebook.ipynb` and run all cells from top to bottom.
3. Raw data is written to `results/raw/` and figures to `results/figures/`.

## Runtime and backend
- Simulator / backend used:
- Shots:
- Approximate run time:
- QPU used? No / Yes (device, date, jobs, credits or cost):
```

REPORT.md — what the result means

```
# Qxx — Project title: Report

## 1. Baseline
What did you reproduce? State the prediction you wrote before running, then the measured result
(give the metric as a number, not only a figure).

## 2. Extension
Which single primary variable did you change? What stayed fixed? What did you predict?

## 3. Results
Tables and labeled figures (stored in `results/figures/`). Report the metric for the baseline and
for the extension.

## 4. Interpretation
Did the results support the prediction? Separate three possible sources of any difference:
1. the ideal mathematical prediction,
2. finite-shot sampling,
3. implementation effects such as noise or compilation.

## 5. Uncertainty
How large is the sampling uncertainty? How do you know (repeated runs, shot count, error estimate)?

## 6. Limitations
What can this experiment NOT tell us? What would you change with more time or resources?

## 7. Conclusion
Two or three sentences: what you learned and what evidence supports it.

## Sources
List every third-party source, tutorial, or code you used.
```

What a good answer looks like

SECTION	WEAK	STRONG
Hypothesis	“I think it will work.”	“P(0) will be within 0.05 of 0.5 at 1,000 shots, because the ideal probability is ½.”
Results	A screenshot of a histogram.	A table of counts, the metric as a number, and a labeled figure made from the saved data.
Extension	Several things changed at once.	One variable changed, the others listed as fixed.

Interpretation	“The result was close.”	“The remaining difference of 0.014 is consistent with sampling error at 1,000 shots.”
Limitations	Left empty.	“A simulator has no hardware noise, so this says nothing about a real device.”

Explaining a difference between prediction and result

When your measurement does not match the ideal value, decide which of three layers can explain it:

1. **The ideal mathematical prediction** — was the calculation itself right?
2. **Finite-shot sampling** — is the difference the size you expect from randomness?
3. **Implementation effects** — noise or compilation, when you use a noisy simulator or real hardware.

Final checklist

This is the same checklist you confirm when you open your Pull Request.

- README reproduces the work
- Notebook is executable, or its limitations are documented
- Raw data included
- Figures labeled
- REPORT interprets uncertainty and limitations
- Dependencies documented
- No secrets or private information
- Third-party sources and code cited

Guide 3 — Submitting with GitHub

About 20 minutes | No GitHub experience needed

You submit a project by adding one folder to your own copy of the OQCL repository and asking for it to be reviewed. That request is called a Pull Request. This guide shows the whole path in the browser, with a command-line option at the end.

Before you start. For graded or QPU work, apply by email first and wait for approval: saghiria@wpunj.edu with the project ID, project title, your name, and a short plan.

Step 1 — Create a GitHub account

If you do not have one, sign up at github.com. Your username becomes part of your project folder name.

Step 2 — Fork the repository

Open the OQCL repository and click **Fork** in the top-right corner, then **Create fork**. You now have your own copy at `github.com/YOUR-USERNAME/open-quantum-computing-lab`. You can change it freely without affecting anyone else.

Step 3 — Prepare your project folder on your computer

Name it with the project ID and your GitHub username, and use this layout:

```
q01-yourgithubname/
  README.md
  notebook.ipynb
  src/                 # optional
  results/raw/
  results/figures/
  environment/requirements.txt
  REPORT.md
```

FILE	PURPOSE
README.md	Question, hypothesis, reproduction instructions, runtime/backend.
notebook.ipynb	Executable experiment and analysis.

results/raw/	Machine-readable counts/data; never credentials.
results/figures/	Labeled plots/diagrams generated from results.
REPORT.md	Baseline, extension, interpretation, uncertainty, limitations, conclusion.

Step 4 — Upload the folder to your fork

1. In **your fork**, open the `projects` folder.
2. Click **Add file → Upload files**.
3. Drag your whole `q01-yourgithubname` folder into the page. Dragging the folder keeps its structure.
4. Under **Commit changes**, write a short message such as `Q01 submission: yourgithubname`.
5. Choose **Create a new branch for this commit** and click **Propose changes**.

Step 5 — Open the Pull Request

1. GitHub shows a page titled **Open a pull request**. Check that the base repository is `amsaghiri/open-quantum-computing-lab`.
2. The description box is pre-filled with the OQCL submission form. Fill in every line: project ID, baseline, extension, result with the metric, and the execution metadata (backend, shots, working time, and cost or credits if any).
3. Tick each box in the checklist only if it is true.
4. Click **Create pull request**.

Step 6 — Respond to review

Your instructor may ask for changes. Edit or upload files on the same branch in your fork and the Pull Request updates automatically. You do not need to open a new one.

> **Never upload these**
>
> API keys, tokens, passwords, private keys, billing or account identifiers, or private student information. Check your notebook outputs too: a token printed in a cell is still a leaked token.

Command-line option

If you already use Git, this does the same as Steps 4 and 5. Replace `YOUR-USERNAME` and the folder name.

```
git clone https://github.com/YOUR-USERNAME/open-quantum-computing-lab.git
cd open-quantum-computing-lab
git checkout -b q01-yourgithubname

# copy your finished folder into projects/
cp -r /path/to/q01-yourgithubname projects/

git add projects/q01-yourgithubname
git commit -m "Q01 submission: yourgithubname"
git push -u origin q01-yourgithubname
```

Then open your fork on GitHub and click **Compare & pull request**.

Course submission and public publication are separate

Submitting for the course does not publish your work in the public book. Student work enters the curated public book only through an explicit opt-in and attribution process.

PART 2

Quantum in 7 Sessions

One idea per session, with a worked example, two experiments, common mistakes, and self-check questions with answers.

Quantum in 7 Sessions

A guided path through OQCL for students who want a clear order to follow. Each session has one main idea, a short reading, a worked example, two small experiments you can run, self-check questions with answers, and a project.

How to use it. Do the sessions in order, one per week or at your own pace. Each takes roughly one study sitting for the reading and experiment, plus the time of the project you choose. You need Python with Qiskit installed; Guide 1 shows the setup. All code on these pages was run with Qiskit 2.6 and Qiskit Aer 0.18.

The whole path at a glance

SESSION	TOPIC	READ	PROJECTS
Session 1	**Qubits and amplitudes** What a qubit is, and why an amplitude is not a probability.	Ch. 1	Q01, Q02
Session 2	**Measurement, gates, and interference** How gates change a state, and how amplitudes can cancel.	Ch. 2	Q03, Q08
Session 3	**Entanglement** Two qubits whose results are linked in a way no single qubit explains.	Ch. 3	Q04, Q06, Q05
Session 4	**Real hardware: compilation and noise** Why the circuit you draw is not the circuit the machine runs.	Ch. 4, Ch. 7	Q09, Q10, Q23, Q14
Session 5	**Algorithms: search and phase estimation** Using interference on purpose to make the right answer likely.	Ch. 5, Ch. 6	Q11, Q13, Q12
Session 6	**Near-term methods: Hamiltonians and variational circuits** A quantum circuit with adjustable angles, tuned by a classical loop.	Ch. 8, Ch. 9, Ch. 10	Q16, Q18, Q19, Q20

Session 7	**Communication, benchmarking, and your own project** A key-distribution protocol, honest measurement, and what to do next.	Ch. 11, Ch. 12	Q21, Q22, Q24

Built for self-study

Each session page contains a worked example to do on paper, two experiments with their real output, a list of common mistakes and tips, and self-check questions with answers you can reveal. Instructors will find matching teacher notes for every session.

What these sessions are, and are not

They are a study order and a set of small experiments built on the OQCL textbook. They do not replace the chapters, which carry the full explanations and worked examples, and seven sessions will not make anyone an expert. The aim is that after Session 7 you can read a simple circuit, predict what it does, test the prediction, and report the result honestly.

Session 1 — Qubits and amplitudes

Session 1 of 7 | Self-study | Simulator only

By the end of this session you can describe a qubit with two amplitudes, turn amplitudes into measurement probabilities, and say why phase matters even when you cannot see it in one measurement.

1. Read

- Chapter 1 — Mathematical Language of Quantum Computing

Read the chapter first. The summary below is a reminder, not a replacement.

2. Key ideas

- A qubit state is written $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. The numbers α and β are complex **amplitudes**.
- The Born rule turns amplitudes into probabilities: $P(0) = |\alpha|^2$ and $P(1) = |\beta|^2$. A valid state has $|\alpha|^2 + |\beta|^2 = 1$.
- Multiplying the whole state by a **global phase** changes nothing you can observe. A **relative phase** between α and β is different: it can change what later gates do.
- Two states can give identical probabilities when measured right away and still be different states.

3. Worked example

Work through this with pen and paper before touching the code.

Normalize $3|0\rangle + 4i|1\rangle$ and find the probabilities.

1. Squared length: $|3|^2 + |4i|^2 = 9 + 16 = 25$, so the length is 5.
2. Divide by the length: $|\psi\rangle = (3/5)|0\rangle + (4i/5)|1\rangle$.
3. Born rule: $P(0) = (3/5)^2 = 0.36$ and $P(1) = |4i/5|^2 = 0.64$.
4. Check: $0.36 + 0.64 = 1$.

The factor i did not affect either probability. It is a relative phase, and it will matter as soon as another gate is applied.

4. Try it

Build two states that have the same measurement probabilities but a different relative phase, and check how similar they really are. Write down what you expect before you run the code.

```
import numpy as np
from qiskit.quantum_info import Statevector

# A qubit state is two complex amplitudes: alpha for |0> and beta for |1>
psi = Statevector([1, 1j]) / np.sqrt(2)

print("valid state:", psi.is_valid())          # is it normalized?
print("probabilities:", psi.probabilities())   # [P(0), P(1)]

# The same probabilities, a different relative phase
phi = Statevector([1, -1j]) / np.sqrt(2)
print("probabilities:", phi.probabilities())
print("overlap |<phi|psi>|^2:", abs(phi.inner(psi)) ** 2)
```

Output (counts will vary slightly from run to run):

```
valid state: True
probabilities: [0.5 0.5]
probabilities: [0.5 0.5]
overlap |<phi|psi>|^2: 0.0
```

What to notice

Both states give 50/50, so a single measurement in the computational basis cannot tell them apart. Yet their overlap is 0: they are orthogonal, as different as two quantum states can be. The difference lives entirely in the relative phase.

5. Second experiment

Check the worked example by computer, and confirm that a global phase changes nothing. Predict first.

```
import numpy as np
from qiskit.quantum_info import Statevector

v = np.array([3, 4j])                 # not normalized: 3|0> + 4i|1>
psi = Statevector(v / np.linalg.norm(v))
print("norm of v:", np.linalg.norm(v))
print("probabilities:", psi.probabilities())

# A global phase changes nothing observable
same = Statevector(-1 * psi.data)
print("overlap with -psi:", round(abs(same.inner(psi)) ** 2, 6))
```

Output:

```
norm of v: 5.0
probabilities: [0.36 0.64]
overlap with -psi: 1.0
```

What it shows

The computer agrees with the hand calculation: 0.36 and 0.64. Multiplying the whole state by −1 gives overlap 1, meaning it is physically the same state.

6. Common mistakes and tips

Common mistakes

- Adding amplitudes and expecting 1. It is the **squared magnitudes** that add to 1.
- Squaring a complex amplitude instead of taking its magnitude first: $|i|^2 = 1$, but $i^2 = -1$.
- Deciding that phase is irrelevant because it does not show up in one measurement.

Tips

- Always check normalization first. It catches most arithmetic slips.
- Keep amplitudes as long as possible and convert to probabilities only at the very end.
- Draw the two amplitudes as arrows in the complex plane. A relative phase is the angle between them.

7. Check yourself

Answer each question on paper, then open the answer to compare.

1. Why can $\alpha = 1$ and $\beta = 1$ not describe a qubit without rescaling?

   ANSWER

   $|1|^2 + |1|^2 = 2$, not 1. Divide both amplitudes by $\sqrt{2}$.

2. What information do you lose if you replace amplitudes by probabilities too early?

   ANSWER

   The relative phase. Without it you cannot predict what happens after another gate, because interference depends on it.

3. Change the second state to [1, 1j] multiplied by −1. Predict the overlap with the first state before you run it.

   ANSWER

   The overlap is 1. Multiplying the whole state by −1 is a global phase, so it is the same physical state. The second experiment confirms it.

8. Do a project

Pick one. Each follows the routine from Guide 1: predict, run the baseline, change one variable, analyze.

- Q01 Quantum Coin
- Q02 Amplitude and Phase Explorer

Session 2 — Measurement, gates, and interference

Session 2 of 7 | Self-study | Simulator only

By the end of this session you can predict the result of the X, Z, and H gates on a single qubit, explain measurement as sampling, and show interference with a three-gate circuit.

1. Read

- Chapter 2 — Qubits, Measurement, and Single-Qubit Gates

Read the chapter first. The summary below is a reminder, not a replacement.

2. Key ideas

- A gate is a reversible (unitary) operation on the state. **X** swaps $|0\rangle$ and $|1\rangle$. **Z** flips the sign of the $|1\rangle$ amplitude. **H** turns $|0\rangle$ into an equal superposition.
- Measurement is **sampling**: each shot gives 0 or 1, and you estimate probabilities from many shots. More shots give a better estimate, not a different state.
- Applying H twice returns the qubit to $|0\rangle$. The two paths to $|1\rangle$ have opposite signs and cancel. That cancellation is **interference**.
- A Z gate between the two H gates changes only a phase, yet it flips the final answer from 0 to 1. Phase is invisible until interference makes it visible.

3. Worked example

Work through this with pen and paper before touching the code.

Follow the amplitudes through H, Z, H starting from $|0\rangle$.

1. $H|0\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$.
2. Z flips the sign of $|1\rangle$: $(|0\rangle - |1\rangle)/\sqrt{2}$. The probabilities are still 50/50.
3. Apply H to each part: $H|0\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ and $H|1\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$.
4. Combine: $\frac{1}{2}[(|0\rangle + |1\rangle) - (|0\rangle - |1\rangle)] = \frac{1}{2}[2|1\rangle] = |1\rangle$.

The two $|0\rangle$ contributions cancel and the two $|1\rangle$ contributions add. Without the Z, the signs are the other way round and the result is $|0\rangle$.

4. Try it

Run three small circuits and compare them. Predict each result before you look. Write down what you expect before you run the code.

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator

sim = AerSimulator(seed_simulator=7)

def run(build, label):
    qc = QuantumCircuit(1, 1)
    build(qc)
    qc.measure(0, 0)
    print(label, sim.run(qc, shots=1000).result().get_counts())

run(lambda qc: qc.h(0), "H        ")                          # about half 0, half 1
run(lambda qc: (qc.h(0), qc.h(0)), "H, H     ")              # always 0
run(lambda qc: (qc.h(0), qc.z(0), qc.h(0)), "H, Z, H  ")     # always 1
```

Output (counts will vary slightly from run to run):

```
H         {'1': 486, '0': 514}
H, H      {'0': 1000}
H, Z, H   {'1': 1000}
```

> **What to notice**
>
> One H gives a random-looking result. Two H gates give 0 every time, so the randomness was not noise: it was a superposition that could be undone. Inserting Z, which changes no probability on its own, turns the answer into 1 every time.

5. Second experiment

Replace the Z in the middle with X and predict the result first. Predict first.

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator

qc = QuantumCircuit(1, 1)
qc.h(0)
qc.x(0)          # X instead of Z in the middle
qc.h(0)
qc.measure(0, 0)
print(AerSimulator(seed_simulator=7).run(qc, shots=1000).result().get_counts())
```

Output:

```
{'0': 1000}
```

What it shows

The answer is 0 every time. X swaps the two amplitudes of H|0⟩, but they are equal, so nothing changes and the second H returns the qubit to |0⟩. Equivalently, HXH = Z, and Z leaves |0⟩ alone.

6. Common mistakes and tips

Common mistakes

- Reading H as “makes the qubit random”. A random bit stays random after a second H; a superposition does not.
- Thinking more shots change the state. Shots only improve your estimate of the same probabilities.
- Assuming Z “does nothing” because it leaves the probabilities of |0⟩ and |1⟩ unchanged.

Tips

- When unsure, track amplitudes line by line as in the worked example. Three lines of algebra beat guessing.
- Sampling error shrinks like 1/√shots. To halve the error you need four times the shots.
- Useful identities: HH = I, HZH = X, HXH = Z.

7. Check yourself

Answer each question on paper, then open the answer to compare.

1. If H made the qubit “randomly 0 or 1”, what would H followed by H give? Why is that not what happens?

   ANSWER

   A random bit would still be random after the second H, giving about 50/50. The experiment gives 0 every time, so the state after one H was a definite superposition, not a random bit.

2. What does X do between the two H gates instead of Z? Predict, then test.

   ANSWER

   Always 0. See the second experiment: X swaps two equal amplitudes, which changes nothing.

3. How many shots do you need before P(0) for a single H is reliably within 0.05 of 0.5?

ANSWER

The standard error of P(0) is √(0.25/N). Two standard errors within 0.05 needs N of about 400. With 100 shots you will often miss by more than 0.05.

8. Do a project

Pick one. Each follows the routine from Guide 1: predict, run the baseline, change one variable, analyze.

- Q03 X, H, and Interference
- Q08 Shot-Count Convergence

Session 3 — Entanglement

Session 3 of 7 | Self-study | Simulator only

By the end of this session you can write a two-qubit state in the computational basis, build a Bell state with H and CNOT, and say what the measurement results do and do not show.

1. Read

- Chapter 3 — Multiple Qubits, Tensor Products, and Entanglement

Read the chapter first. The summary below is a reminder, not a replacement.

2. Key ideas

- Two qubits have four basis states: $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$. In general n qubits need 2^n amplitudes.
- A **CNOT** flips the target qubit only when the control qubit is 1.
- H on the first qubit followed by CNOT produces the Bell state $(|00\rangle + |11\rangle)/\sqrt{2}$. It cannot be written as "a state of qubit A" times "a state of qubit B". That is **entanglement**.
- Each qubit alone looks random, but the two results always agree. Agreement in one basis is not yet proof of entanglement; you need a second measurement basis to rule out an ordinary shared coin.

3. Worked example

Work through this with pen and paper before touching the code.

Build the Bell state step by step, writing states as $|$first qubit, second qubit$\rangle$.

1. Start: $|00\rangle$.
2. H on the first qubit: $(|00\rangle + |10\rangle)/\sqrt{2}$.
3. CNOT with the first qubit as control flips the second qubit only in the $|10\rangle$ term: $(|00\rangle + |11\rangle)/\sqrt{2}$.

Why it is not two separate qubits. A product state looks like $(a|0\rangle + b|1\rangle)(c|0\rangle + d|1\rangle) = ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|11\rangle$. To match the Bell state we need $ad = 0$ and $bc = 0$, but also $ac \neq 0$ and $bd \neq 0$. If $ad = 0$ then $a = 0$ or $d = 0$, which makes ac or bd zero. No choice works, so the state cannot be split.

4. Try it

Build the Bell state and look at which outcomes appear. Write down what you expect before you run the code.

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator

qc = QuantumCircuit(2, 2)
qc.h(0)          # put qubit 0 in superposition
qc.cx(0, 1)      # entangle qubit 1 with qubit 0
qc.measure([0, 1], [0, 1])

counts = AerSimulator(seed_simulator=7).run(qc, shots=1000).result().get_counts()
print(counts)    # only '00' and '11' appear
```

Output (counts will vary slightly from run to run):

```
{'11': 511, '00': 489}
```

What to notice

Only 00 and 11 appear, each about half the time; 01 and 10 never do. Each qubit on its own is 50/50, but together they are perfectly correlated. Note that Qiskit prints the bit string with qubit 0 on the right.

5. Second experiment

Measure the same Bell state in a different basis by adding an H on each qubit just before measurement. Predict first.

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator

qc = QuantumCircuit(2, 2)
qc.h(0)
qc.cx(0, 1)      # Bell state
qc.h([0, 1])     # rotate both qubits: measure in the X basis instead of Z
qc.measure([0, 1], [0, 1])
print(AerSimulator(seed_simulator=7).run(qc, shots=1000).result().get_counts())
```

Output:

```
{'11': 511, '00': 489}
```

What it shows

The results are still only 00 and 11. A pair of classical coins that merely agree would give all four outcomes here. Perfect correlation in two different bases is the signature that separates a Bell state from a shared classical coin.

6. Common mistakes and tips

Common mistakes

- Saying “the qubits are both 0 or both 1, we just do not know which”. That describes a classical shared coin, not a Bell state.
- Believing that measuring one qubit sends a signal to the other. The correlations cannot be used to communicate.
- Reading Qiskit bit strings left to right. Qiskit prints qubit 0 on the **right**.

Tips

- To test for entanglement, measure in a second basis. A classical mixture loses its correlation there; a Bell state keeps it.
- Count amplitudes: n qubits need 2^n of them. This is why classical simulation gets hard quickly.
- Draw the circuit before writing code. Most errors are a control and target swapped.

7. Check yourself

Answer each question on paper, then open the answer to compare.

1. A classical pair of coins glued together also gives only 00 and 11. What extra measurement separates that from a Bell state?

   ANSWER

   Measure both qubits in the X basis (an H on each before measuring). A Bell state still gives only 00 and 11. A classical mixture of 00 and 11 gives all four outcomes about equally.

2. Add a third qubit and one more CNOT to make a GHZ state. Which outcomes do you expect?

   ANSWER

   Only 000 and 111, each about half the time.

3. How many amplitudes describe 10 qubits? 30 qubits?

ANSWER

2^{10} = 1,024 and 2^{30} = 1,073,741,824, about a billion.

8. Do a project

Pick one. Each follows the routine from Guide 1: predict, run the baseline, change one variable, analyze.

- Q04 Bell Correlations
- Q06 GHZ Scaling
- Q05 Bell State Under Noise

Session 4 — Real hardware: compilation and noise

Session 4 of 7 | Self-study | Simulator only

By the end of this session you can tell a logical circuit from the compiled circuit a device runs, explain why limited connectivity adds gates, and name the main kinds of error.

1. Read

- Chapter 4 — Circuits, Compilation, and Real Hardware
- Chapter 7 — Noise, Error Detection, and Quantum Error Correction

Read the chapters first. The summary below is a reminder, not a replacement.

2. Key ideas

- A real device supports only a small **native gate set**, and only certain pairs of qubits can interact directly (its **connectivity**).
- **Compilation** (transpilation) rewrites your circuit into native gates and inserts extra operations to move information between distant qubits.
- Every extra gate, and especially every two-qubit gate, is another chance for error. Depth and two-qubit gate count are the first numbers to check before a hardware run.
- Errors come in two basic kinds, bit flips and phase flips. Error correction spreads information over several qubits so errors can be detected without copying the unknown state, which is forbidden.

3. Worked example

Work through this with pen and paper before touching the code.

Estimate how much routing costs. Suppose every two-qubit gate succeeds with probability 0.99, and ignore every other error.

1. Logical circuit, 3 two-qubit gates: $0.99^3 \approx 0.970$.
2. Compiled circuit, 6 two-qubit gates: $0.99^6 \approx 0.941$.
3. Routing alone moved the estimate from about 97% to about 94%, before counting single-qubit gates, readout, or idle time.

This is a rough model, not a prediction for any particular device. Its use is to compare two versions of the same circuit before you spend hardware time.

4. Try it

Take a 4-qubit GHZ circuit in which one qubit controls all the others, and compile it for a device whose qubits sit in a line. Write down what you expect before you run the code.

```
from qiskit import QuantumCircuit, transpile

# Logical circuit: a 4-qubit GHZ state with qubit 0 controlling all the others
qc = QuantumCircuit(4)
qc.h(0)
qc.cx(0, 1)
qc.cx(0, 2)
qc.cx(0, 3)

# A device whose qubits sit in a line: 0-1-2-3
line = [[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2]]
hw = transpile(qc, basis_gates=["rz", "sx", "x", "cx"], coupling_map=line,
               optimization_level=1, seed_transpiler=7)

print("logical : depth", qc.depth(), " two-qubit gates", qc.count_ops().get("cx", 0))
print("hardware: depth", hw.depth(), " two-qubit gates", hw.count_ops().get("cx", 0))
```

Output (counts will vary slightly from run to run):

```
logical : depth 4  two-qubit gates 3
hardware: depth 9  two-qubit gates 6
```

What to notice

The logical circuit needs 3 two-qubit gates. On a line, qubit 0 is not next to qubits 2 and 3, so the compiler adds routing and the count roughly doubles, with a much larger depth. Nothing about the algorithm changed; only the hardware layout did. Exact numbers can differ between Qiskit versions.

5. Second experiment

Build the same GHZ state as a chain, so that every CNOT acts on neighbouring qubits. Predict first.

```
from qiskit import QuantumCircuit, transpile

# The same GHZ state, built as a chain: each CNOT acts on neighbours
qc = QuantumCircuit(4)
qc.h(0)
qc.cx(0, 1)
qc.cx(1, 2)
qc.cx(2, 3)

line = [[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2]]
hw = transpile(qc, basis_gates=["rz", "sx", "x", "cx"], coupling_map=line,
               optimization_level=1, seed_transpiler=7)
print("hardware: depth", hw.depth(), " two-qubit gates", hw.count_ops().get("cx", 0))
```

Output:

```
hardware: depth 6  two-qubit gates 3
```

What it shows

Now the compiled circuit needs only 3 two-qubit gates, the same as the logical one. The state prepared is identical; only the order of the CNOTs changed. Designing with the hardware layout in mind removed all the routing overhead.

6. Common mistakes and tips

Common mistakes

- Reporting the depth of the circuit you drew instead of the circuit that ran.
- Assuming all qubits on a device are equally good and all pairs are connected.
- Blaming every difference from the ideal result on “noise” without separating sampling error, compilation, and device error.

Tips

- Always save both the logical and the compiled circuit, and record depth and two-qubit gate count for each.
- Rewrite the circuit to respect the device layout when you can. It is often cheaper than letting the compiler route.
- Fix the transpiler seed so that your compiled circuit is reproducible.

7. Check yourself

Answer each question on paper, then open the answer to compare.

1. Rewrite the logical circuit as a chain (0→1, 1→2, 2→3). It prepares the same GHZ state. What happens to the compiled gate count?

   ANSWER

   > It stays at 3, because every CNOT already acts on neighbours. See the second experiment.

2. Why can a quantum error-correcting code not simply copy the qubit three times?

   ANSWER

   > An unknown quantum state cannot be copied (the no-cloning principle). Error correction instead spreads the information over several qubits using entanglement.

3. Before spending hardware time, which three numbers would you record for a circuit?

   ANSWER

   > Circuit depth, number of two-qubit gates, and the number of jobs and shots (with the estimated cost or credits).

8. Do a project

Pick one. Each follows the routine from Guide 1: predict, run the baseline, change one variable, analyze.

- Q09 Physical Qubit Mapping
- Q10 Compilation Tradeoffs
- Q23 Cost-Aware Hardware Plan
- Q14 Syndrome Decoding

Session 5 — Algorithms: search and phase estimation

Session 5 of 7 | Self-study | Simulator only

By the end of this session you can explain Grover's search as a sign flip followed by a reflection, run it on four states, and describe what phase estimation measures.

1. Read

- Chapter 5 — Interference and Quantum Search
- Chapter 6 — Phase, Eigenvalues, and Phase Estimation

Read the chapters first. The summary below is a reminder, not a replacement.

2. Key ideas

- An **oracle** marks the answer by flipping the sign of its amplitude. Probabilities do not change yet.
- The **diffusion** step reflects all amplitudes about their average. The marked amplitude grows and the others shrink.
- For four states, one round is enough to make the marked state certain. In general Grover needs about √N rounds, a quadratic saving in oracle calls for unstructured search. It does not "read all answers at once".
- **Phase estimation** uses the same idea in a different way: a controlled operation writes the phase of an eigenvalue onto helper qubits, where it can be read out.

3. Worked example

Work through this with pen and paper before touching the code.

Grover on four states, by hand. Suppose the marked state is |11⟩.

1. After the first H gates, all four amplitudes are ½.
2. Oracle: the marked amplitude becomes −½. The amplitudes are (½, ½, ½, −½).
3. Average: (½ + ½ + ½ − ½)/4 = ¼.
4. Diffusion replaces each amplitude a by 2 × average − a. Unmarked: 2 × ¼ − ½ = 0. Marked: 2 × ¼ − (−½) = 1.

The final amplitudes are (0, 0, 0, 1), so the marked state is measured with certainty after one round.

4. Try it

Search four states for the marked one, |11⟩, with a single Grover round. Write down what you expect before you run the code.

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator

qc = QuantumCircuit(2, 2)
qc.h([0, 1])        # equal superposition of 00, 01, 10, 11

qc.cz(0, 1)         # oracle: flip the sign of the marked state |11>

qc.h([0, 1])        # diffusion: reflect about the average amplitude
qc.x([0, 1])
qc.cz(0, 1)
qc.x([0, 1])
qc.h([0, 1])

qc.measure([0, 1], [0, 1])
print(AerSimulator(seed_simulator=7).run(qc, shots=1000).result().get_counts())
```

Output (counts will vary slightly from run to run):

```
{'11': 1000}
```

What to notice

All 1,000 shots return 11. After the first H gates every state had probability ¼. The oracle changed only a sign, and the diffusion step converted that hidden sign into a certain answer. This is the interference from Session 2, used deliberately.

5. Second experiment

Apply the Grover round twice instead of once. Predict first.

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator

def grover_round(qc):
    qc.cz(0, 1)                       # oracle for |11>
    qc.h([0, 1]); qc.x([0, 1])
    qc.cz(0, 1)                       # diffusion
    qc.x([0, 1]); qc.h([0, 1])

qc = QuantumCircuit(2, 2)
qc.h([0, 1])
grover_round(qc)
grover_round(qc)                      # one round too many
qc.measure([0, 1], [0, 1])
print(AerSimulator(seed_simulator=7).run(qc, shots=1000).result().get_counts())
```

Output:

```
{'01': 267, '11': 244, '00': 260, '10': 229}
```

What it shows

All four outcomes now appear about equally: the advantage is gone. The second round rotated the state past the answer. More rounds are not better; there is a correct number, and for four states it is one.

6. Common mistakes and tips

Common mistakes

- Saying Grover “tries all answers at once”. It rotates amplitude toward the answer step by step.
- Running more rounds to be safe. Too many rounds rotate the state past the answer and the success probability falls.
- Forgetting that the oracle changes only a sign. On its own it changes no probability.

Tips

- For N items use about (π/4)√N rounds. For N = 4 that is one.
- Check your oracle separately: prepare the marked state directly and confirm that only its sign flips.
- Do the amplitude arithmetic for N = 4 by hand once. It makes every larger case easier to believe.

7. Check yourself

Answer each question on paper, then open the answer to compare.

1. Change the oracle so that it marks $|01\rangle$ instead (hint: an X gate on one qubit before and after the CZ). Predict the result.

   ANSWER

   Put an X on one qubit before and after the oracle's CZ. With the X on qubit 1 the output is 01; with the X on qubit 0 it is 10 (remember Qiskit prints qubit 0 on the right).

2. What happens if you apply the Grover round twice on four states? Predict, then test.

   ANSWER

   The success probability drops back to about 25%, as the second experiment shows.

3. Why does the oracle alone not help, even though it "knows" the answer?

   ANSWER

   The oracle only flips a sign, which leaves all four probabilities at ¼. Interference in the diffusion step is what turns the sign into a measurable difference.

8. Do a project

Pick one. Each follows the routine from Guide 1: predict, run the baseline, change one variable, analyze.

- Q11 Grover Search: Four States
- Q13 Phase Estimation
- Q12 Grover Under Noise

Session 6 — Near-term methods: Hamiltonians and variational circuits

Session 6 of 7 | Self-study | Simulator only

By the end of this session you can say what an expectation value is, describe the hybrid loop used by VQE and QAOA, and explain why a classical baseline is always required.

1. Read

- Chapter 8 — Hamiltonians, Dynamics, and Trotterization
- Chapter 9 — Variational and Hybrid Quantum Algorithms
- Chapter 10 — Quantum Machine Learning

Read the chapters first. The summary below is a reminder, not a replacement.

2. Key ideas

- A **Hamiltonian** describes the energy of a system. Measuring an operator many times gives its **expectation value**, the average result.
- A **parameterized circuit** has gates with adjustable angles. Different angles prepare different trial states.
- The **hybrid loop**: the quantum circuit prepares a trial state and estimates the energy; a classical optimizer chooses the next angles. VQE, QAOA, and variational classifiers all follow this pattern.
- At teaching sizes a classical computer solves these problems easily. A small demonstration is never evidence of quantum advantage, so always compare with an exact or classical answer.

3. Worked example

Work through this with pen and paper before touching the code.

From counts to an expectation value. The state $Ry(\theta)|0\rangle$ is $\cos(\theta/2)|0\rangle + \sin(\theta/2)|1\rangle$.

1. $P(0) = \cos^2(\theta/2)$ and $P(1) = \sin^2(\theta/2)$.
2. Z gives +1 for outcome 0 and −1 for outcome 1, so $\langle Z\rangle = P(0) - P(1) = \cos^2(\theta/2) - \sin^2(\theta/2) = \cos\theta$.
3. Suppose 1,000 shots give 250 zeros and 750 ones. Then $\langle Z\rangle \approx 0.25 - 0.75 = -0.5$.
4. $\cos\theta = -0.5$ means $\theta = 2\pi/3$, or 120°.

This is how an energy is estimated on real hardware: from counts, not from the state itself.

4. Try it

A one-qubit version of the variational idea: find the angle θ that minimizes the expectation value of Z for the state $Ry(\theta)|0\rangle$. Write down what you expect before you run the code.

```
import numpy as np
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector, SparsePauliOp

Z = SparsePauliOp("Z")

def energy(theta):
    qc = QuantumCircuit(1)
    qc.ry(theta, 0)                                    # the trial state
    return Statevector(qc).expectation_value(Z).real  # <Z> = cos(theta)

# The "classical optimizer": try many angles and keep the best one
thetas = np.linspace(0, 2 * np.pi, 41)
values = [energy(t) for t in thetas]
best = int(np.argmin(values))

print("lowest <Z> found:", round(values[best], 3))
print("at theta =", round(thetas[best], 3), "(pi =", round(np.pi, 3), ")")
```

Output (counts will vary slightly from run to run):

```
lowest <Z> found: -1.0
at theta = 3.142 (pi = 3.142 )
```

What to notice

The lowest value is −1 at $\theta = \pi$, where $Ry(\theta)$ has rotated $|0\rangle$ into $|1\rangle$. For this state $\langle Z\rangle = \cos\theta$, so you can check the answer by hand. That hand calculation is the classical baseline; in a real problem you would compare against exact diagonalization or brute force.

5. Second experiment

Repeat the scan with the operator X instead of Z. Predict first.

```
import numpy as np
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector, SparsePauliOp

X = SparsePauliOp("X")
thetas = np.linspace(0, 2 * np.pi, 41)
values = []
for t in thetas:
    qc = QuantumCircuit(1)
    qc.ry(t, 0)
    values.append(Statevector(qc).expectation_value(X).real)   # <X> = sin(theta)

best = int(np.argmin(values))
print("lowest <X>:", round(values[best], 3), "at theta =", round(thetas[best], 3),
      "(3*pi/2 =", round(1.5 * np.pi, 3), ")")
```

Output:

```
lowest <X>: -1.0 at theta = 4.712 (3*pi/2 = 4.712 )
```

What it shows

The minimum is −1 at θ = 3π/2. For this state ⟨X⟩ = sin θ, so the result can be checked by hand. A different operator has its minimum at a different angle: the “best” circuit depends on the question you ask.

6. Common mistakes and tips

Common mistakes

- Treating a variational result as exact. It is an estimate with sampling error, and the optimizer may stop in a local minimum.
- Skipping the classical baseline. At teaching sizes the exact answer is always available, so compare against it.
- Calling a small successful demonstration “quantum advantage”.

Tips

- Plot the whole energy curve for a one-parameter problem before trusting any optimizer.
- Estimate ⟨Z⟩ as P(0) − P(1). Its sampling error is at most about 1/√shots.
- Change one thing at a time: the ansatz, the optimizer, or the number of shots.

7. Check yourself

Answer each question on paper, then open the answer to compare.

1. Replace Z by X in the code. At which angle is ⟨X⟩ lowest? Work it out first.

   ANSWER

   At θ = 3π/2, where ⟨X⟩ = sin θ = −1. See the second experiment.

2. On a real device you would estimate ⟨Z⟩ from counts as P(0) − P(1). How does the number of shots affect the optimizer?

   ANSWER

   Each energy estimate carries random error of roughly 1/√shots. If two angles differ in energy by less than that, the optimizer cannot tell which is better and may wander.

3. Why is “my VQE found the right energy for a tiny molecule” not a claim of advantage?

   ANSWER

   A classical computer solves a tiny instance exactly and instantly. Getting the right answer shows the method works, not that it beats classical computing.

8. Do a project

Pick one. Each follows the routine from Guide 1: predict, run the baseline, change one variable, analyze.

- Q16 Ising Trotter Study
- Q18 VQE Energy Estimation
- Q19 QAOA Max-Cut
- Q20 Variational Quantum Classifier

Session 7 — Communication, benchmarking, and your own project

Session 7 of 7 | Self-study | Simulator only

By the end of this session you can explain how BB84 uses two measurement bases to share a key, state what a benchmark can and cannot claim, and plan a reproducible project of your own.

1. Read

- Chapter 11 — Quantum Communication and BB84
- Chapter 12 — Benchmarking and Reproducible Quantum Experiments

Read the chapters first. The summary below is a reminder, not a replacement.

2. Key ideas

- In **BB84** Alice encodes each bit in one of two bases, chosen at random. Bob measures in a basis he chooses at random.
- When their bases match, Bob's result equals Alice's bit. When they differ, his result is random, so those rounds are discarded (**sifting**).
- An eavesdropper who measures in the wrong basis disturbs the state and causes errors in the sifted key, which Alice and Bob can detect by comparing a sample.
- A **benchmark** describes one device, one circuit, and one day. Keep raw data, settings, and controls so that someone else can repeat it, and claim only what the evidence supports.

3. Worked example

Work through this with pen and paper before touching the code.

How much error does an eavesdropper cause? Look only at rounds where Alice and Bob used the same basis.

1. Eve guesses the basis. She is right half the time, and then causes no error.
2. The other half, she measures in the wrong basis and sends on a state in that wrong basis. Bob, measuring in Alice's basis, now gets a random result and is wrong half the time.
3. Error rate in the sifted key: ½ × ½ = ¼, or 25%.
4. If Alice and Bob compare k bits, Eve escapes detection on each with probability ¾. For k = 20 that is $(¾)^{20} \approx 0.003$, about 0.3%.

4. Try it

Simulate 32 rounds of BB84 with no eavesdropper and compare the sifted keys. Write down what you expect before you run the code.

```
import random
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator

random.seed(7)
sim = AerSimulator(seed_simulator=7)
n = 32
key_alice, key_bob = [], []

for _ in range(n):
    bit, basis_a, basis_b = random.randint(0, 1), random.choice("ZX"), random.choice("ZX")

    qc = QuantumCircuit(1, 1)
    if bit:
        qc.x(0)                 # Alice encodes her bit
    if basis_a == "X":
        qc.h(0)                 # ... in her chosen basis
    if basis_b == "X":
        qc.h(0)                 # Bob measures in his chosen basis
    qc.measure(0, 0)
    result = int(list(sim.run(qc, shots=1).result().get_counts())[0])

    if basis_a == basis_b:      # sifting: keep only the rounds where bases match
        key_alice.append(bit)
        key_bob.append(result)

print("rounds kept:", len(key_alice), "of", n)
print("Alice:", "".join(map(str, key_alice)))
print("Bob  :", "".join(map(str, key_bob)))
print("keys identical:", key_alice == key_bob)
```

Output (counts will vary slightly from run to run):

```
rounds kept: 19 of 32
Alice: 0110111011111011010
Bob  : 0110111011111011010
keys identical: True
```

> **What to notice**
>
> About half of the rounds survive sifting, and the two keys are identical. This is a protocol demonstration, not a security proof. The interesting experiment is the next one: add an eavesdropper and measure how often the keys now disagree.

5. Second experiment

Add an eavesdropper who measures each qubit in a random basis and sends on what she saw. Predict first.

```
import random
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator

random.seed(7)
sim = AerSimulator(seed_simulator=7)

def measure(qc):
    qc.measure(0, 0)
    return int(list(sim.run(qc, shots=1).result().get_counts())[0])

def prepare(bit, basis):
    qc = QuantumCircuit(1, 1)
    if bit:
        qc.x(0)
    if basis == "X":
        qc.h(0)
    return qc

kept = errors = 0
for _ in range(400):
    bit, basis_a, basis_e, basis_b = (random.randint(0, 1), random.choice("ZX"),
                                      random.choice("ZX"), random.choice("ZX"))
    # Eve intercepts: measures in her own random basis ...
    qc = prepare(bit, basis_a)
    if basis_e == "X":
        qc.h(0)
    eve_bit = measure(qc)
    # ... and sends Bob a fresh qubit prepared from what she saw
    qc = prepare(eve_bit, basis_e)
    if basis_b == "X":
        qc.h(0)
    bob_bit = measure(qc)

    if basis_a == basis_b:
        kept += 1
        errors += (bob_bit != bit)

print("rounds kept:", kept, " errors:", errors, " error rate:", round(errors / kept, 3))
```

Output:

```
rounds kept: 207  errors: 52  error rate: 0.251
```

What it shows

About a quarter of the sifted bits are now wrong, matching the hand calculation of 25%. Without Eve the error rate was zero. That jump is what Alice and Bob look for.

6. Common mistakes and tips

Common mistakes

- Thinking the basis announcement leaks the key. Bases are revealed only after the measurements, and the bit values never are.
- Keeping the bits used for the error check. They were announced publicly and must be discarded.
- Treating a simulated protocol as a security proof, or a single benchmark run as a general statement about a device.

Tips

- Write a small table by hand first: Alice's bit and basis, Bob's basis and result, keep or discard.
- With real hardware some errors appear even without an eavesdropper. The protocol must set a threshold.
- For any benchmark, record device, date, shots, circuit, and seed. Without them the number means little.

7. Check yourself

Answer each question on paper, then open the answer to compare.

1. Add Eve: before Bob's gates, measure the qubit in a random basis and prepare it again from her result. What error rate do you predict in the sifted key?

   ANSWER

   About 25%, as the worked example and the second experiment show.

2. Why must Alice and Bob sacrifice part of the key to check for errors?

   ANSWER

   Comparing bits is the only way to estimate the error rate, and bits that have been announced are no longer secret, so they are thrown away.

3. Pick one result from an earlier session. What would another student need from you to reproduce it exactly?

   ANSWER

   The code, the package versions, the random seed, the number of shots, the backend, and the raw counts.

8. Do a project

Pick one. Each follows the routine from Guide 1: predict, run the baseline, change one variable, analyze.

- Q21 BB84 and Eavesdropping
- Q22 Reproducible Processor Benchmark
- Q24 Reproduce, Extend, Publish