# Assignment 1 — Paper-to-Code: Single-Qubit Foundations & Early Multi-Qubit Circuits

**Submission:** After the deadline, you will no longer have access to this Git repository. See **Section 3.1** of this PDF for the required deliverables to include in this repository.

## 1 — Learning Objectives

By completing this assignment, you will be able to:

1. **Set up a reproducible Python environment** for quantum programming using modern tooling (and run the provided tests locally).
2. **Represent and manipulate** 1–2 qubit states using Dirac notation and matrix/vector representations (including normalization and measurement probabilities).
3. **Prove and simplify small circuit identities** (up to global phase), using gate algebra and rotation identities.
4. **Compare two programming paradigms** for quantum circuits:
   - **circuits-as-objects** (Qiskit), and
   - **circuits-as-functions** (PennyLane), by implementing the same state-preparation task in both.
5. **Implement and verify circuits in Qiskit**, including single-qubit state preparation, early two-qubit circuits (e.g., Bell states), and phase-insensitive equivalence checks.
6. **Build two mini-applications** in Qiskit:
   - the **SWAP test** (overlap estimation), and
   - the **Bernstein–Vazirani algorithm** (hidden bit-string recovery), and connect each implementation to its theoretical guarantee.

***Scope note (important):*** *Some questions introduce multi-qubit circuits earlier than the lecture sequence. For those parts, you are graded primarily on* ***correct construction and verification via simulation****. Don't worry if you don't understand the multi-qubit circuit constructions, focus on understanding the programming frameworks.*

---

## 2 — Development Environment Setup

We use **pixi** (a modern conda-style dependencies manager) for reproducible environments and tests. You may choose to work **remotely** in GitHub Codespaces by following the steps in **2.1**, or choose to set up a **local** development environment on your machine by following the steps in **2.2**.

---

### 2.1 — Remote (GitHub Codespaces)

1. Click **Code → Codespaces → Create codespace on main**.
2. Wait for setup. The image includes `pixi`, and dependencies for the current assignment will install automatically on first run.

*GitHub Codespaces are limited to 60 hours of active usage per month per account (wall-clock time, not CPU hours). If you reach this limit, you can continue the assignment using a local setup or another online development environment, we are trying to get more hours from GitHub for this course.*

Possible issues you may encounter:

- GitHub might get stuck on "Opening codespaces" after you complete step 1. If this happens, try closing and reopening the **Code → Codespaces** menu, or try reloading the page.

- You may encounter the same issue described here: Codespaces opens then fails saying "Oh no, it looks like you're offline!". Follow the instruction described in that link to disable your browser's tracking prevention for your codespace's URL.

---

## 2.2 — Local Setup

Steps 2.2.1 and 2.2.2 only need to be done once to set up your coding environment. Step 2.2.3 needs to be done once per new assignment to gather the resources and download the dependencies necessary for your coding.

### 2.2.1 — Setting up your coding environment on Windows 10/11 via WSL2

1. Install **Windows Subsystem for Linux 2 (WSL2)** by opening PowerShell as Administrator and running: `wsl.exe --install`. Reboot your machine if prompted.
2. Install **Ubuntu 24.04** for WSL from the Microsoft Store: https://apps.microsoft.com/detail/9NZ3KLHXDJP5 (If needed, enable WSL2: `wsl --install` in PowerShell, then reboot.)
3. Launch **Ubuntu 24.04.1 LTS** and create your UNIX username & password.
4. From now on, "terminal/shell" = the **Ubuntu 24.04** terminal (not PowerShell/CMD).

### 2.2.2 — Install and Set Up git and pixi (WSL/Ubuntu/macOS)

#### Create a course folder and open VS Code (or your preferred code editor)

If needed, close your terminal and install VS Code: https://code.visualstudio.com/.

Then, in a terminal/shell, run:

```
mkdir -p ~/ECE484 && cd ~/ECE484
code .
```

#### Update tools & install git

- **WSL/Ubuntu**

  ```
  sudo apt update
  sudo apt install -y git build-essential curl
  ```

*For other Linux distributions (such as Arch or Red Hat–based distributions like Fedora), refer to the documentation of the package manager on your system.*

- **macOS** (requires Homebrew: https://brew.sh/)

  ```
  brew update
  brew install git
  ```

#### Configure GitHub SSH (if not done before):

*Follow the instructions at https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account?platform=linux&tool=webui.*

#### Setup your name and email on git

Get your Github spam-free email from https://github.com/settings/emails (it looks like `<numbers>+<username>@users.noreply.github.com`).

```
git config --global user.name <Github-USERNAME>
git config --global user.email <Github-EMAIL>
```

Install **pixi**:

```
curl -fsSL https://pixi.sh/install.sh | sh
# or:
# wget -qO- https://pixi.sh/install.sh | sh
```

> *Then close and open your shell so the PATH variable updates. Check that the installation was successful by running `pixi --version`.*

### 2.2.3 — Clone your assignment repository and install dependencies (pixi)

**Clone your private assignment repo**

```
git clone <SSH-URL-FROM-GITHUB-CLASSROOM>
cd <your-repository>
```

**This repo includes `assign1.toml` (environment specifications) and `assign1.lock` (exact download links). Install them using:**

```
pixi install
```

> *Here's a corrected and clearer version, keeping the same style but fixing grammar, flow, and a couple of awkward phrases:*

> *The dependencies file was created for a Linux x86_64 platform. On macOS, you may need to re-resolve the dependencies for your platform by running `pixi update --platform osx-64` or `pixi update --platform osx-arm64` (for Apple Silicon Macs). You may also need to manually change the platform in `pixi.toml` to osx-64 or osx-arm64 before running the install command again.*
> *If that does not work, try recreating the environment from scratch by deleting `pixi.toml` and `pixi.lock`, then running `pixi init .` followed by `pixi add python` and `pixi add --pypi pennylane qiskit-aer qiskit qiskit-ibm-runtime numpy pytest black ruff pyyaml pytest-xdist qutip qpic`, and in the future run the tests using `pixi run pytest -v -s tests`.*
> *If the issue persists, please try an online development environment such as GitHub Codespaces or contact the TAs.*

---

### 2.3 — Verification of the Environment

Run the following to verify that pixi and Python are working correctly:

```
pixi run python -V
# or run provided tests, e.g.:
pixi run tests
```

---

## 3 — Questions (Total = 100)

The first three questions **Q1–Q3** are **on-paper only** (PDF), then **Q4** introduces Qiskit & PennyLane, and **Q5–Q6** are Qiskit-only.

---

### 3.1 — Deliverables

**Create:**

- `paper/answers.pdf` containing answers for **Q1–Q3** and your short discussions for **Q4.4**, **Q6a.2**, and **Q6b.2**.

**Complete the coding TODOs in:**

- `src/Q4_frameworks.py` (Q4)
- `src/Q5_synthesis.py` (Q5)
- `src/Q6a_SwapTest.py` (Q6a)
- `src/Q6b_BernsteinVazirani.py` (Q6b)

Expected new files structure:

```
<your-repository>/
 paper/
      answers.pdf
 src/
       Q4_frameworks.py
       Q5_synthesis.py
       Q6a_SwapTest.py
       Q6b_BernsteinVazirani.py
```

Do no touch the files in `tests/`, `.devcontainer/`, `random_seed.yaml` or any other files/folders not mentioned above. We will automatically detect the submission that have modified forbidden files and may penalize them.

During your work, you can run the provided tests to verify your implementations using:

```
pixi run tests
```

*To help with debugging, the flag -s was added to the pytest command to show print statements.*

---

### 3.2 — Prerequisites

You will need the following:

- **State definitions:** Single-qubit computational basis states:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

  Two-qubit computational basis states:

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

- **Pauli matrices, Hadamard, and rotation gates:**

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$R_X(\alpha) = e^{-i\alpha X/2} = \begin{pmatrix} \cos(\alpha/2) & -i\sin(\alpha/2) \\ -i\sin(\alpha/2) & \cos(\alpha/2) \end{pmatrix}$$

$$R_Y(\alpha) = e^{-i\alpha Y/2} = \begin{pmatrix} \cos(\alpha/2) & -\sin(\alpha/2) \\ \sin(\alpha/2) & \cos(\alpha/2) \end{pmatrix}$$

$$R_Z(\alpha) = e^{-i\alpha Z/2} = \begin{pmatrix} e^{-i\alpha/2} & 0 \\ 0 & e^{i\alpha/2} \end{pmatrix}$$

- **Bra-ket notation, normalization, and applying gate operators:**

  The bra corresponding to ket $|\psi\rangle$ is $\langle\psi| = |\psi\rangle^\dagger$ (conjugate transpose).

  A state is normalized if $\langle\psi|\psi\rangle = 1$.

  Applying a gate $U$ to a state $|\psi\rangle$ is done via matrix-vector multiplication: $|\psi'\rangle = U|\psi\rangle$.

- **Controlled operators:**
  For a single-qubit unitary $U$, the controlled-$U$ gate is:

$$CU = |0\rangle\langle 0| \otimes I \ + \ |1\rangle\langle 1| \otimes U$$

  *Circuit diagram placeholder (to be added)*
  This applies $U$ to the target qubit if and only if the control qubit is in state $|1\rangle$.

- **Global vs. local phase:**
  A **global phase** $e^{i\theta}$ multiplies the entire state: $e^{i\theta}|\psi\rangle$. It is physically unobservable.
  A **relative (local) phase** affects amplitudes differently, e.g., $|0\rangle + e^{i\phi}|1\rangle$, and is observable via interference.
  Example: $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$ and $e^{i\pi/4}|+\rangle$ are physically identical, but $|+\rangle$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$ are distinguishable.

- **Measurement in the computational basis (Born rule):**
  For a state $|\psi\rangle = \sum_i c_i|i\rangle$, the probability of measuring outcome $|i\rangle$ is:

$$P(i) = |\langle i|\psi\rangle|^2 = |c_i|^2$$

  After measurement yielding $|i\rangle$, the state collapses to $|i\rangle$.

---

**3.3 — Grading rules you should know**

- **Global phase is irrelevant** when comparing statevectors. We grade via fidelity; those are phase-insensitive checks.
- For coding parts, we check both:
  - correctness of outputs, and
  - **structure constraints** (gate sets, single-oracle call, controlled-SWAP present, etc.).
- You may create your own helper functions, but respect the constraints imposed in each question.
- Grading points are indicative only; we may adjust them slightly during grading to reflect difficulty.

---

**Q1 — Single-Qubit Algebra & Bloch Intuition (15)**

Let $|\psi(\theta,\phi)\rangle = \cos(\frac{\theta}{2})|0\rangle + e^{i\phi}\sin(\frac{\theta}{2})|1\rangle$.

1. **Normalization.** Show that $|\psi(\theta,\phi)\rangle$ is a valid quantum state, i.e., normalized for all $\theta \in [0,\pi]$ and $\phi \in [0,2\pi)$.
2. **Apply gates.** Apply $H$ then $Z$ to $|\psi(\theta,\phi)\rangle$. Write the final state explicitly.
3. **Conjugation & rotations.** Prove $HZH = X$. Then derive $HR_Z(\alpha)H = R_X(\alpha)$ (up to global phase). You may use: $R_\sigma(\alpha) = e^{-i\alpha\sigma/2} = \cos(\frac{\alpha}{2})I - i\sin(\frac{\alpha}{2})\sigma, \quad \sigma \in X, Y, Z$.
4. **Bloch sketch.** Sketch the Bloch sphere and indicate where $(\theta,\phi)$ sits; label axes and angles. You may draw by hand, use any software, or use the python package `qutip` (https://qutip.org/docs/4.0.2/guide/guide-bloch.html).

---

## Q2 — Two-Qubit Target State Circuit Design (15)

We will use the 2-qubit target: $|\Psi(\theta,\phi)\rangle = \cos(\frac{\theta}{2})|00\rangle + e^{i\phi}\sin(\frac{\theta}{2})|11\rangle$.

You may only use the gate set $CX, R_Z, R_X$ and start from the **initial quantum state** $|00\rangle$.

1. **Circuit design.** Propose a circuit that prepares $|\Psi(\theta,\phi)\rangle$ from $|00\rangle$.
    1. Draw the circuit. If using LaTeX, you may want to use the `quantikz` package. You may also want to use qpic, a python tool to quickly generate tikz/png/pdf circuits using simple instructions.
    2. Give a short step-by-step explanation about how the state evolves over time. (You may describe key steps in words rather than precise mathematical formulas.)

---

## Q3 — Phase Kickback via Controlled Rotation $\mathrm{CR}_X(\varphi)$ (10)

This question builds intuition for the "phase kickback" effect. You may use the controlled-operators definition from the prerequisites.

1. Show that the CZ gate is symmetric, i.e., control and target may be swapped without changing the operator (up to global phase). Tip: You may use the matrix definition of CZ and apply it to each element of the 2-qubit computational basis ($|00\rangle, |01\rangle, |10\rangle, |11\rangle$), then argue from linearity.
2. Consider $\mathrm{CR}_X(\varphi)$ (Controlled-Rotation-X), where the control is initially in state $|+\rangle$ and the target is initially in state $|+\rangle$.
    1. Show that after applying $\mathrm{CR}_X(\varphi)$ the control acquires a relative phase that depends on $\varphi$. (Note that, since $|+\rangle$ is an $X$-eigenstate with eigenvalue $+1$ and $R_X(\varphi) = e^{-i\varphi X/2}$, we have $R_X(\varphi)|+\rangle = e^{-i\varphi/2}|+\rangle$.)
    2. Then apply a Hadamard $H$ to the **control** and measure only the control qubit in the computational ($Z$) basis. Give $P(|0\rangle_c)$ and $P(|1\rangle_c)$ in terms of $\varphi$.
    3. What do you notice from the above? Give a briefly interpretation of this "phase-kickback" effect.

---

## Q4 — Frameworks Mini-Comparison (Qiskit & PennyLane) (20, Bonus +10)

**Objective.** Gain experience with two different quantum programming frameworks: `Qiskit` (IBM's circuits-as-objects) and `PennyLane` (Xanadu's circuits-as-functions). Both support statevector simulation. Inside the file `src/Q4_frameworks.py`, implement functions that prepare the same 2-qubit target state from **Q2** using both frameworks, and compare results. The following pages may help: Qiskit documentation, PennyLane documentation.

Implement in `src/Q4_frameworks.py`:

1. `build_target_state_qiskit(theta, phi) -> np.ndarray` Build the circuit from Q2 that prepares $|\Psi(\theta,\phi)\rangle$, and return the a length 4 complex NumPy array that represents the final **statevector**.

2. `build_target_state_pennylane(theta, phi) -> np.ndarray` Do the same in PennyLane (`default.qubit`) and return a NumPy array representation of the final statevector.

3. **(Bonus)** `build_target_state_numpy(theta, phi) -> np.ndarray` Construct NumPy matrices for $CX$, $R_Z$, and $R_X$, then build the corresponding two-qubit operators and apply them starting from the statevector $|00\rangle$. You should obtain the same final NumPy array as in the previous two points.

4. **Short written reflection.** Give one pro & one con for each framework; which felt more natural for you and why? Which one seems easier to extend the circuit? Your choices will not impact your grading as long as they are justified.

---

***From now on, all coding must be done using the Qiskit library only.***

---

## Q5 — Synthesis & Equivalence (20)

**Objective.** Implement state preparation circuits for single-qubit states and Bell states, and a helper function to compare unitaries up to global phase. Complete the functions in `src/Q5_synthesis.py`.

1. Implement `prepare_single(theta, phi) -> QuantumCircuit` to prepare $|\psi(\theta, \phi)\rangle$ from **Q1**, starting from state $|0\rangle$, using the $U$ gate, where $U(\theta, \phi, \lambda)$ is the general single-qubit unitary (documentation: `qiskit.circuit.library.UGate`).
2. Implement `prepare_bell(subscript: str) -> QuantumCircuit` to prepare Bell states from $|00\rangle$ using only gates in the set $H, X, Z, CX$:
   - $|\beta_{00}\rangle = |\Phi^+\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$
   - $|\beta_{10}\rangle = |\Phi^-\rangle = \frac{|00\rangle-|11\rangle}{\sqrt{2}}$
   - $|\beta_{01}\rangle = |\Psi^+\rangle = \frac{|01\rangle+|10\rangle}{\sqrt{2}}$
   - $|\beta_{11}\rangle = |\Psi^-\rangle = \frac{|01\rangle-|10\rangle}{\sqrt{2}}$
3. Implement `unitary_equal(circA, circB, tol)` to check whether two quantum circuits implement the same unitary, i.e. are equivalent up to a global phase. Tips: Read the documentation for `qiskit.quantum_info.Operator` carefully, you only need its constructor and one of its methods to implement this function.

---

## Q6 — Two Mini-Applications (SWAP and BV) (10 + 10)

### Q6a — SWAP Test (10)

**Context.** The SWAP test estimates the **overlap** between two pure states $|\psi\rangle$ and $|\phi\rangle$. With an ancilla prepared in $|0\rangle$, after applying the circuit shown in Figure 1 below to the system:

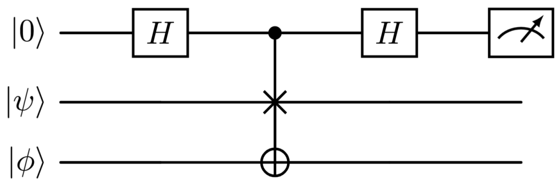

Figure 1: SWAP test circuit

We get: $P(\text{ancilla} = |0\rangle) = \dfrac{1 + |\langle\psi|\phi\rangle|^2}{2}$.

**Objective.** Implement the SWAP test circuit in Qiskit to estimate the overlap between two single-qubit states in `src/Q6a_SwapTest.py`.

1. Implement `swap_test_noiseless(psi: QuantumCircuit, phi: QuantumCircuit) -> float`, which builds and runs the standard SWAP test circuit on `AerSimulator` in a noiseless manner. Then implement `swap_test_noisy(psi: QuantumCircuit, phi: QuantumCircuit, random_seed: int, shots: int = 1000) -> float`, which runs the SWAP test in a noisy simulation. More information is provided in the Python file.
2. Discuss briefly the benefits of this approach in the aspect of the final states of the system after measurement of the ancilla.

### Q6b — Bernstein–Vazirani (10)

**Context.** Given an oracle that implements a function $f : \{0,1\}^n \to \{0,1\}$ in which $f(x)$ is promised to be the dot product between $x$ and a secret string $a \in \{0,1\}^n$ modulo 2, the Bernstein-Vazirani (BV) algorithm finds the unknown bit-string $a$ in **one query** by preparing Hadamards, querying the oracle once, and measuring the input register.

**Objective.** Complete `bernstein_vazirani()` in `src/Q6b_BernsteinVazirani.py`.

1. Implement `bernstein_vazirani(oracle: QuantumCircuit) -> str` that returns the recovered string `a` after building and running the BV circuit on `AerSimulator` in a noiseless scenario. More information is provided in the python file.
2. Consider how you would find the secret string $a$ from $f$ using a classical algorithm. In what way is the BV algorithm more efficient than a classical algorithm for this problem? Is there any part of your BV implementation implementation that could affect this advantage in practice, and why?