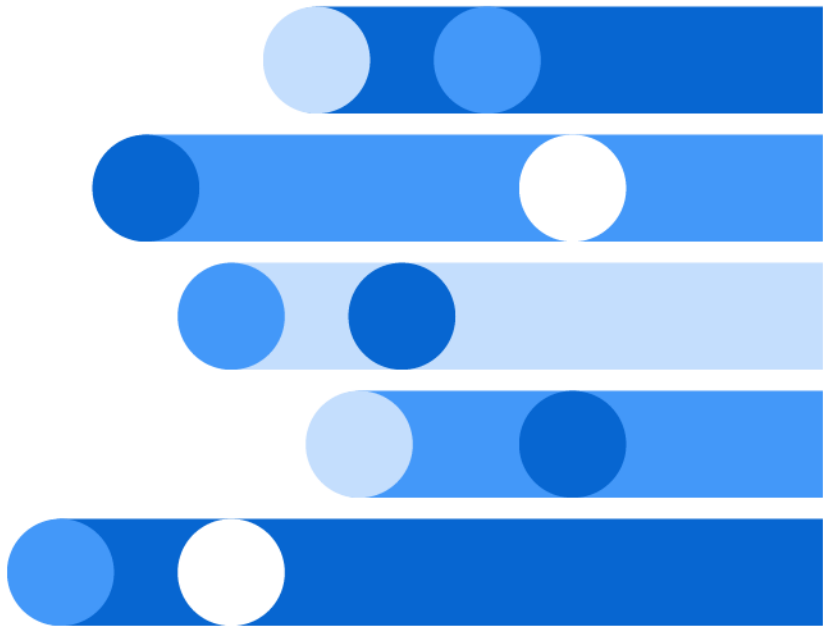
# SAS® Quantum Lab: User's Guide

2026.11*



* This document might apply to additional versions of the software. Open this document in SAS Help Center and click on the version in the banner to see all available versions.



SAS® Documentation
November 17, 2026

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2026. *SAS® Quantum Lab: User's Guide*. Cary, NC: SAS Institute Inc.

**SAS® Quantum Lab: User's Guide**

Copyright © 2026, SAS Institute Inc., Cary, NC, USA

All Rights Reserved. Produced in the United States of America.

**For a hard copy book:** No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

**For a web download or e-book:** Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

**U.S. Government License Rights; Restricted Rights:** The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

November 2026

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

v_001-P1:quantug

# Contents


*Chapter 1 / Getting Started with SAS Quantum Lab* . . . **1**
About SAS Quantum Lab . . . 2
Prerequisites and Setup for SAS Quantum Lab . . . 3
Core Quantum Computing Concepts . . . 5
SAS Interfaces for Quantum Programming . . . 7
Your First Quantum Circuit . . . 9
Using the Quantum Tutor . . . 12

*Chapter 2 / Building a Quantum Circuit* . . . **17**
Components of Quantum Circuits . . . 17
Diagramming a Quantum Circuit . . . 20
Parameterized Circuits . . . 21
Building Batch Circuits . . . 24
QASM Import and Export . . . 25
Circuit Data Model . . . 26
Quantum Gate Reference . . . 29

*Chapter 3 / Running a Quantum Circuit* . . . **35**
Sampling Measurements from a Quantum Circuit . . . 35
Computing Observables for a Quantum Circuit . . . 37
Running Parameterized Circuits . . . 41
Preparing a Circuit for Real Hardware . . . 43
Running Circuits on Remote Hardware . . . 47
Authenticating to a Remote Provider . . . 49

*Chapter 4 / HYPERTUNE Procedure* . . . **51**
Overview: HYPERTUNE Procedure . . . 51
Syntax: HYPERTUNE Procedure . . . 53
Usage: HYPERTUNE Procedure . . . 61
Examples: HYPERTUNE Procedure . . . 68

*Chapter 5 / QRUN Procedure* . . . **77**
Overview: QRUN Procedure . . . 77
Syntax: QRUN Procedure . . . 77

*Chapter 6 / Quantum Gate Macros* . . . **85**
Quantum Gate Reference . . . 86

*Chapter 7 / Quantum Execution Macros* . . . **115**
Using Quantum Execution Macros . . . 115
Macros to Statically Inspect Quantum Circuits . . . 116

*Chapter 8 / Quantum Machine Learning Macros* . . . **139**
Quantum Machine Learning Concepts . . . 139
Examples: . . . 164

# 1

# Getting Started with SAS Quantum Lab


***About SAS Quantum Lab*** **2**
- What Is SAS Quantum Lab? 2
- Intended Use 2
- Limitations on Use 2

***Prerequisites and Setup for SAS Quantum Lab*** **3**
- Prerequisites 3
- Get SAS Quantum Lab in Your SAS Viya Environment 3
- Third-Party Quantum Hardware Access 4

***Core Quantum Computing Concepts*** **5**
- Qubits and Gates 5
- Circuits 5
- Observables 6
- Simulation 6
- Hardware Execution 6
- Quantum Machine Learning 7

***SAS Interfaces for Quantum Programming*** **7**
- DATA Step and Gate Macros 7
- QRUN Procedure 8
- Quantum Macros 8
- Custom Steps for SAS Data and AI Studio 8
- Quantum Machine Learning Macros 8
- HYPERTUNE Procedure 9

***Your First Quantum Circuit*** **9**
- Build the Circuit 9
- Draw the Circuit 10
- Run the Circuit 10
- Modify the Circuit 11
- Next Steps 11

***Using the Quantum Tutor*** **12**
- Accessing and Using the Quantum Tutor 12
- Lesson Outline 12
- Tips for Asking Questions 13
- Example Prompts 14
- Limitations 14

# About SAS Quantum Lab

## What Is SAS Quantum Lab?

The SAS Quantum Lab offering is a set of tools on the SAS Viya platform for learning about, building, and running quantum circuits. SAS Quantum Lab enables you to build circuits with SAS macros, run them on quantum simulators or on supported third-party quantum computing platforms or hardware, and combine the results with SAS analytics and machine learning. SAS Quantum Lab is intended for education, experimentation, and research. The following topics describe the permitted scope, intended use, and limitations on use of SAS Quantum Lab.

## Intended Use

SAS Quantum Lab is intended to enable customers to learn, explore, develop, and evaluate quantum computing applications within the SAS Viya environment. Users can create, execute, and analyze quantum algorithms by using quantum simulators and supported third-party quantum computing platforms, and they can integrate quantum workflows with traditional SAS analytics, machine learning, and data science capabilities. Intended uses include experimentation, skills development, research, proof-of-concept activities, design of quantum circuits, comparison of classical and quantum approaches, exploration of hybrid quantum-classical workflows, and evaluation of potential business applications. The offering and its outputs are intended to support human-led research, analysis, and decision-making, not automated decision-making. Customers should have appropriately qualified personnel independently review and validate outputs before using them for operational, business, regulatory, financial, legal, medical, safety-related, scientific, or other consequential purposes.

## Limitations on Use

SAS Quantum Lab relies on customer-provided data and circuits, third-party software components, and, where applicable, customer-procured quantum computing services. Results may vary based on the selected simulator, hardware platform, and underlying technology. Quantum hardware remains an emerging technology and may be affected by noise, errors, queuing, execution constraints, and other limitations. Because the offering uses experimental, statistical, probabilistic, and quantum computational methods, results may be inaccurate,

incomplete, inconsistent, or nondeterministic. Repeated executions of the same circuit may produce different results. Any particular quantum algorithm, hardware platform, or workflow may or may not provide a computational advantage, business benefit, or superior outcome compared to classical approaches.

# Prerequisites and Setup for SAS Quantum Lab

This topic describes the prerequisites for SAS Quantum Lab, discusses how to make SAS Quantum Lab available in your SAS Viya environment, and presents information about connecting to third-party quantum hardware.

## Prerequisites

The following requirements apply to SAS Quantum Lab:

- SAS Viya: SAS Quantum Lab is an add-on to SAS Viya, SAS Viya Advanced, or SAS Viya Enterprise.
- SAS Cloud Analytic Services (CAS): SAS Quantum Lab requires CAS.
- Infrastructure: Standard SAS Viya platform and CAS infrastructure is sufficient. SAS Quantum Lab does not require a dedicated GPU.
- Sizing: SAS Quantum Lab uses your existing SAS Viya platform and CAS infrastructure. The resources that you need depend on your expected workload and on how many users and jobs run at the same time.

## Get SAS Quantum Lab in Your SAS Viya Environment

To make SAS Quantum Lab available in your SAS Viya environment, complete the following steps:

1. To request SAS Quantum Lab, contact your SAS account representative. SAS Quantum Lab is a separate add-on for the SAS Viya platform that you acquire through the standard SAS ordering process.
2. Confirm that your SAS Viya release is supported. SAS Quantum Lab is available starting in 2026.11. You must update your SAS Viya platform to 2026.11 or later to access SAS Quantum Lab.

3 Update your SAS Viya platform to 2026.11 or later. Follow the standard SAS Viya update process to add SAS Quantum Lab. SAS Quantum Lab does not require a separate installation. For more information, see "Guidelines for Updating to a New Version" in *SAS Viya Platform Operations: Updating Software Tasks*.

4 Apply the new license. You receive the updated license through the ordering process. To apply the license to your SAS Viya environment, follow the instructions in "How to Apply a New License" in *SAS Viya Platform: Licensing*.

5 Set up the credential and secrets and enable SAS Viya Copilot to access the Quantum Tutor. SAS Viya Copilot requires you to set up a credential and secrets. SAS Viya Copilot is not enabled by default. Your SAS Viya platform administrator must enable it in SAS Environment Manager. For more information, see "Set Up the Credential and Secrets for SAS Viya Copilot" in *SAS Viya Platform: Licensing* and "SAS Viya Copilot Page" in *SAS Environment Manager: User's Guide*.

# Third-Party Quantum Hardware Access

Access to a quantum processing unit (QPU) is optional. You can use SAS Quantum Lab and simulate quantum circuits without a connection to external quantum hardware. To run circuits on third-party hardware, you connect to the hardware with your own access. SAS Quantum Lab supports IBM Quantum for hardware execution.

The following information applies to third-party hardware access:

- You obtain and manage your own third-party quantum access, including the account, the entitlement or credits, and the API credentials. The cost of any third-party hardware is not included in the cost of SAS Quantum Lab.
- You configure your third-party credentials for use with SAS Quantum Lab. SAS Quantum Lab uses the credentials to submit supported jobs to the provider and to retrieve the results. SAS does not generate or manage your credentials.
- You are responsible for your third-party account and credentials. You are also responsible for enabling network connectivity from your SAS Viya environment to the provider, which might include firewall or proxy configuration.
- When you select hardware execution, SAS Quantum Lab submits your quantum circuit and execution parameters to the hardware and returns the measurements to your SAS program. SAS Quantum Lab is not intended to be used to send raw business data sets to the QPU.

# Core Quantum Computing Concepts

This section introduces the concepts behind SAS Quantum Lab software. Each topic describes a concept in quantum computing and then explains how the concept applies in SAS Quantum Lab.

## Qubits and Gates

A quantum bit, which is usually referred to as a qubit, is the quantum counterpart of a classical bit. A classical bit is always 0 or 1, whereas a qubit can be in the state 0, the state 1, or a superposition of both states. A superposition is a combination of 0 and 1 that gives a probability for each outcome. A measurement of a qubit always returns a 0 or 1. Qubits can also be entangled, which means that specific qubits are linked and that the state of one qubit cannot be described without the state of another qubit. A gate is an operation that changes the state of one or more qubits. Single-qubit gates, such as the Hadamard gate, act on one qubit. Multi-qubit gates, such as the CNOT gate, act on more than one qubit and can create entanglement.

In SAS Quantum Lab, you set the number of qubits and classical bits when you initialize a circuit by using the %Q_INIT macro, and every qubit starts in the |0⟩ state. Each gate has its own macro, such as the %Q_H macro for the Hadamard gate and the %Q_CX macro for the CNOT gate. The gates are applied in the order in which you call the macros in the circuit. Measurements store results of the circuits in classical bits. For more information about qubits, see “Qubits and Classical Bits” on page 18. For more information about gates, see “Quantum Gates” on page 18.

## Circuits

A quantum circuit is a sequence of operations, which are called gates, that act on qubits. Each qubit starts in the |0⟩ state, the gates change the state, and measurements at the end of the process produce classical results. A quantum circuit is an ordered list of operations, so you can build the circuit once and run it many times. A "shot" in quantum computing refers to one run or execution of a quantum circuit. Quantum measurements and observations are probabilistic; performing many shots helps you determine the behavior of the circuit.

In SAS Quantum Lab, you build a circuit by using %Q_ macros within a DATA step. The circuit is stored as a SAS data set of OpenQASM lines. You can print the data set, draw it by using the %Q_SHOW_CIRCUIT macro, and pass it to the execution macros or to PROC QRUN. For more information about building circuits, see Chapter 2, “Building a Quantum Circuit,” on page 17.

# Observables

In quantum computing, an observable is a quantity that you can measure, and the expectation value of an observable is the average result over many measurements. Observables are important because a single measurement of a qubit returns only a 0 or 1, so one measurement does not describe the state of a circuit. Common quantum observables are built from the Pauli operators X, Y, and Z. For example, the observable Pauli operator ZZ can help identify quantum entanglement and correlation between two qubits.

In SAS Quantum Lab, you can read the results of a circuit in multiple ways. Sampling a quantum circuit returns each measurement result with the number of times that the result occurs and the fraction of runs that the result represents. Observing a quantum circuit returns expectation values for the Pauli terms that you list. You can give each Pauli term a weight. Quantum macros calculate exact expectation values by default. If you specify a number of shots, the macro estimates each expectation value from measurements instead. A circuit that you only observe does not require classical bits. For more information about observables, see “Computing Observables for a Quantum Circuit” on page 37.

# Simulation

Quantum simulation is a process that uses a classical computer to calculate how a quantum circuit behaves. Quantum simulation is useful for testing, learning, and developing algorithms, because simulation does not require access to quantum hardware. The required memory and time for a simulation might grow quickly with each additional qubit in your circuit. Simulators can also model noise to help you understand how errors might affect a result.

In SAS Quantum Lab, circuits run on a local simulator by default. You can select a simulation method and a noise model. Some features, such as exact expectation values and the Bloch sphere and QSphere plots, are available only when you use an ideal local simulation.

# Hardware Execution

Quantum hardware runs circuits on physical quantum processors that you reach through a cloud provider. Real devices have noise, queues, limited connections between qubits, and limits on circuit size, so results can vary between runs and can differ from an ideal simulation. Before a quantum circuit runs on hardware, a transpiler rewrites the circuit to use the gates and connections that the physical device supports.

In SAS Quantum Lab, you can select a provider and a device. Currently only the local simulator and IBM Quantum run circuits. You must explicitly authorize hardware runs because they can incur cost. You can transpile a circuit first to see how a device changes it, including the circuit's depth, without contacting the provider. For more information about hardware providers and preparing a circuit for hardware, see "Preparing a Circuit for Real Hardware" on page 43.

## Quantum Machine Learning

Quantum machine learning combines quantum circuits with classical machine learning. A common approach is to encode each row of data into a circuit, measure expectation values, and use those values as new features for a classical model. The specific data and the problem determine whether quantum features improve a model.

In SAS Quantum Lab, the QML macros automate this approach. The %QML_TRANSFORM macro appends quantum features to a data set. The %QML_LOGSELECT, %QML_GRADBOOST, and %QML_SVMACHINE macros train models with those respective quantum features. The %QML_COMPARE macro compares a classical model to a quantum-enhanced one, and the %QML_SWEEP macro searches for optimal circuit settings. The model macros require a CAS session. For more information about quantum machine learning in SAS Quantum Lab, see "Quantum Machine Learning Concepts" on page 139.

# SAS Interfaces for Quantum Programming

There are several interfaces that you can use to inspect, run, and deploy quantum circuits. There are also interfaces that enable you to build quantum circuits, run quantum machine learning workflows, and run general-purpose optimizations.

## DATA Step and Gate Macros

The DATA step and gate macros are used to build a quantum circuit. The circuit is stored as a SAS data set. These data sets are used by the quantum macros and the QRUN procedure to run your circuit.

For details about circuit data sets and how they are built and used, see "Components of Quantum Circuits" on page 17.

For details about gate macros, see "Quantum Gate Macros" on page 85.

# QRUN Procedure

The QRUN procedure is the flagship interface for running quantum circuits. It enables you to sample, observe, visualize, and transpile circuits.

For details about the PROC QRUN interface, see Chapter 5, "QRUN Procedure," on page 77.

# Quantum Macros

Quantum macros are SAS macros that are automatically defined as part of your SAS Quantum Lab deployment. The macro names start with %Q_. When you run a quantum macro, it expands to PROC QRUN code that then runs as it usually would. The quantum macro calls tend to be much more concise than corresponding PROC QRUN statements; it uses strategic defaults and enables you to focus on the most important aspects of your task.

For details about using quantum macros to inspect, run, and deploy quantum circuits, see "Quantum Execution Macros" on page 115.

# Custom Steps for SAS Data and AI Studio

A custom step defines a user interface within SAS Data and AI Studio software. It defines a low-code method of running quantum circuits. You can share custom steps so that other users at your site can complete a specific task.

For a selection of custom steps that you can install, see https://github.com/sassoftware/sas-studio-custom-steps/.

# Quantum Machine Learning Macros

The quantum machine learning macros are a set of SAS macros that are automatically defined as part of your SAS Quantum Lab deployment. The macro names start with %QML_. These macros enable you to run traditional machine learning algorithms by using features that are generated using quantum circuits.

For information about how these macros work and how to use them, see "Quantum Machine Learning Macros" on page 139.

## HYPERTUNE Procedure

The HYPERTUNE procedure is a general-purpose black-box (derivative-free) optimization solver. It enables you to search for optimizations subject to linear and nonlinear constraints. Examples include searching for the best hyperparameters for machine learning, design parameters for simulation studies, and rotation angles for variational quantum algorithms.

For information about using PROC HYPERTUNE, see Overview: HYPERTUNE Procedure on page 51.

# Your First Quantum Circuit

This tutorial guides you through building your first quantum circuit in SAS Quantum Lab. You build, draw, and run a circuit, and then you modify it. The tutorial assumes that you are familiar with SAS programming and that you have reviewed "Core Quantum Computing Concepts" on page 5.

## Build the Circuit

The first circuit creates a Bell state, a state that results in a pair of qubits that are entangled. Here is a program that builds your first circuit in a DATA step:

```
data bell;
    %q_init(2, 2);
    %q_h(0);
    %q_cx(0, 1);
    %q_measure_all();
```

The program performs the following steps:

1. The DATA step creates a data set named *bell* to store the circuit.
2. The %Q_INIT macro starts the circuit with two qubits and two classical bits. Every qubit starts in the |0⟩ state.
3. The %Q_H macro applies a Hadamard gate to qubit 0, which puts it in a superposition of 0 and 1.
4. The %Q_CX macro applies a CNOT gate with qubit 0 as the control and qubit 1 as the target. This gate entangles the two qubits.
5. The %Q_MEASURE_ALL macro measures each qubit into the classical bit that has the same number.

For more information about creating circuits, see Chapter 2, "Building a Quantum Circuit," on page 17.

# Draw the Circuit

Before you run a circuit, you can draw it to check that the gates are on the qubits that you intended. Here is the code to draw a circuit diagram for your first circuit:

```
%q_show_circuit(circuit=bell);
```

The diagram shows one horizontal line for each qubit, with the gates in the order in which you applied them. For more information about circuit diagrams, see "Diagramming a Quantum Circuit" on page 20.

# Run the Circuit

You can run a circuit in multiple ways; this tutorial uses the %Q_SAMPLE macro. The %Q_SAMPLE macro writes its results to a CAS table, so you must start a CAS session and assign a CAS library before you run the circuit. The %Q_SAMPLE macro runs the circuit many times; each run is called a shot. The following program starts a CAS session, runs the circuit, and prints the results:

```
cas qsession;
libname qlib cas sessref=qsession;

%q_sample(circuit=bell, out=qlib.bell_results, shots=1000);

proc print data=qlib.bell_results noobs; run;
```

The program performs the following steps:

1. The CAS statement starts a CAS session named *qsession*, and the LIBNAME statement assigns a CAS library named *qlib*.
2. The %Q_SAMPLE macro runs the circuit 1,000 times and writes the results to the *qlib.bell_results* table.
3. The PROC PRINT step displays the results.

The Results table has one row for each measurement result that occurred. The _measurement_ column contains the result, the _count_ column contains the number of times that it occurred, and the _probability_ column contains the fraction of shots. The results should be only `00` and `11`, and each result should have occurred around 500 times, or around half of the shots. You should never see `01` or `10`, because the two qubits are entangled. The counts might not be exactly 500 each, because the results are probabilistic, and a second run gives slightly different counts. For more information about running a circuit, see Chapter 3, "Running a Quantum Circuit," on page 35 and Chapter 5, "QRUN Procedure," on page 77.

# Modify the Circuit

To change a circuit, you change the program that builds it and then run the new circuit. The following program extends the Bell state to three qubits by adding a qubit and one more CNOT gate to create a GHZ state:

**Note:** This program reuses the CAS session and library from the previous topic.

```
data ghz;
    %q_init(3, 3);
    %q_h(0);
    %q_cx(0, 1);
    %q_cx(1, 2);
    %q_measure_all();

%q_show_circuit(circuit=ghz);

%q_sample(circuit=ghz, out=qlib.ghz_results, shots=1000);

proc print data=qlib.ghz_results noobs; run;
```

The modification of the program results in these changes:

1. The %Q_INIT macro now creates three qubits and three classical bits.
2. The second %Q_CX macro entangles qubit 2 with qubit 1.
3. The OUT= table is a new CAS table in the same library, so the Bell state results stay available.
4. The diagram now has three lines, and the results have three characters.
5. You see only the results `000` and `111`, each in about half the shots.
6. This circuit creates a GHZ state, which entangles all three qubits.

# Next Steps

You built, drew, ran, and modified a circuit. For more information about circuits and gates in SAS Quantum Lab, see the following topics:

- For more information about specific macros, see "Quantum Gate Macros" on page 85 and "Quantum Execution Macros" on page 115.
- For more information about creating circuits, see Chapter 2, "Building a Quantum Circuit," on page 17.
- For more information about running a circuit, see Chapter 3, "Running a Quantum Circuit," on page 35 and Chapter 5, "QRUN Procedure," on page 77.

# Using the Quantum Tutor

The Quantum Tutor is a conversational AI assistant that helps you learn about quantum computing and use SAS Quantum Lab. You can prompt the Quantum Tutor to explain concepts, answer questions, and generate example code that you can run.

## Accessing and Using the Quantum Tutor

**Note:** You must have SAS Viya Copilot to use the Quantum Tutor. For more information about setting up and accessing SAS Viya Copilot, see "Set Up the Credential and Secrets for SAS Viya Copilot" in *SAS Viya Platform: Licensing*.

To ask the Quantum Tutor a question, open the SAS Viya Copilot chat pane and enter a prompt that relates to quantum computing or SAS Quantum Lab. When you ask about quantum computing or SAS Quantum Lab, the SAS Viya Copilot uses the Quantum Tutor to answer. The Quantum Tutor uses product-specific knowledge that includes the capabilities of SAS Quantum Lab, recommended workflows, and how the components work together. Use the Quantum Tutor for these purposes:

- to explore quantum concepts, such as entanglement or superposition
- to understand SAS Quantum Lab functionality
- to help identify the procedures or macros that fit your task
- to generate example code and workflows

You might not know which SAS Quantum Lab capability to use before you begin. Use a prompt to describe what you want to accomplish, and the Quantum Tutor can help you find a starting point.

## Lesson Outline

The Quantum Tutor can teach quantum computing for SAS Quantum Lab in 10 lessons. Each lesson builds on the previous lessons. Here is the typical lesson sequence:

1. Qubits and measurement: Compare classical bits to qubits, sample a circuit, and read the counts and probabilities in the SAS results tables.
2. Gates, superposition, and interference: Use X, H, and rotation gates to see how superposition and phase change the results. Apply H gates twice and use the H-

Z-H sequence to see interference, and then build an intuitive image of the Bloch sphere.

3 Multiple qubits and entanglement: Compare independent qubits to entangled Bell and GHZ states, and use controlled gates and different measurement bases. The lesson helps you understand what correlations do and do not show.

4 Observables: Learn how Pauli terms and expectation values describe a circuit, the difference between sampling and observing a circuit, and how bit ordering works. Compare sampled results with results that do not use shots.

5 Circuit authoring: Build circuits that use symbolic parameters, bind values to them, run batches of distinct circuits, and visualize the circuits.

6 Quantum algorithms: Build the Bernstein-Vazirani algorithm to see phase kickback, a two-qubit Grover search to see amplitude amplification, and a three-qubit quantum Fourier transform (QFT) to see the connection to phase estimation and period finding.

7 Hybrid optimization: Build circuits for a variational quantum eigensolver (VQE) ansatz and for the quantum approximate optimization algorithm (QAOA) for maximum cut problems. This lesson covers cost and mixer layers, objective evaluation, and classical parameter optimization. It also explains the difference between a circuit and a complete optimized algorithm.

8 Execution and realism: Compare simulation methods, noise, and parallel simulation, and prepare circuits by using transpilation. The lesson explains what hardware can do, what it costs, and why hardware execution needs your explicit consent.

9 Quantum-enhanced machine learning: Encode features by using the %QML_TRANSFORM macro, and use topologies and custom observables. Train models by using the %QML_LOGSELECT, %QML_GRADBOOST, and %QML_SVMACHINE macros, and use the %QML_COMPARE macro to compare quantum-enhanced models to classical models. Evaluate each model against classical baselines.

10 Tuning and a practical workflow: Search for good settings by using the %QML_SWEEP macro and documented PROC HYPERTUNE patterns. The lesson covers search budgets, evaluation, and how to choose and keep a final model that you can justify.

This is the typical learning sequence. The Quantum Tutor might add lessons or expand on concepts, depending on your prompts.

# Tips for Asking Questions

Here are tips to help you get useful answers from the Quantum Tutor:

- Mention quantum computing, a quantum concept, or SAS Quantum Lab in your prompt to ensure that the question goes to the Quantum Tutor and not to the standard SAS Viya Copilot.

- Describe the task that you are trying to accomplish. You do not need to know the name of the macro or procedure.
- Tell the Quantum Tutor your baseline knowledge of quantum computing. It adjusts explanations to your level.
- Read the explanation before you read the code. Understanding the concept makes the code easier to follow.
- Run and modify the code examples. Changing an angle or removing a gate shows you how the results change.
- Ask for further explanation if you are confused. The Quantum Tutor can explain the physics behind each gate and measurement.
- Ask for a lesson again if it is unclear to you. You can return to any earlier lesson or expand on any concept.

# Example Prompts

The following prompts show ways to interact with the Quantum Tutor:

- `Teach me quantum computing.`
- `I am new to quantum computing, but I understand SAS programming. How should I get started in SAS Quantum Lab?`
- `Explain quantum superposition.`
- `What does the Hadamard gate do mathematically?`
- `Go back to the entanglement concept and explain it again in more detail.`
- `Generate a program that has three entangled qubits and explain what each gate does.`
- `How do I run this circuit?`
- `Why does a Bell state give only 00 or 11?`
- `What happens if I remove the CNOT gate from this circuit?`

# Limitations

The following limitations apply to the Quantum Tutor:

- The Quantum Tutor is a large language model (LLM) system. It generates responses on the basis of the information and context available to it. It does not guarantee that a response is accurate.
- The additional SAS Quantum Lab knowledge improves the context, but it does not guarantee that every response, recommendation, or code example is correct or appropriate for your situation.

- The Quantum Tutor does not independently make decisions or execute actions.
- Treat generated content as a starting point and not as a substitute for testing and validation. Review generated code, run it in the appropriate environment, and inspect the results.
- Validate that an approach meets the requirements of your application before you use it in production.

# 2

# Building a Quantum Circuit

*Components of Quantum Circuits* . . . 17
Qubits and Classical Bits . . . 18
Quantum Gates . . . 18
Measurements . . . 19
Parameters for Circuit Building . . . 19
Circuit Data Sets . . . 20

*Diagramming a Quantum Circuit* . . . 20
How to Diagram a Circuit . . . 20
Diagram Options . . . 21

*Parameterized Circuits* . . . 21
Defining a Parameter . . . 22
Binding Parameters . . . 22
Common Patterns . . . 23
Example: Running a Circuit for Several Angle Values . . . 23

*Building Batch Circuits* . . . 24

*QASM Import and Export* . . . 25

*Circuit Data Model* . . . 26
Circuit Schema . . . 27
Batch Schema . . . 27
QASM Representation . . . 28

*Quantum Gate Reference* . . . 29
Single-Qubit Gates . . . 29
Rotational Gates . . . 30
Two-Qubit Gates . . . 31
Two-Qubit Ising Gates . . . 32
Three-Qubit Gates . . . 32

## Components of Quantum Circuits

SAS quantum macros consist of a set of %Q_ macros that you use inside a SAS DATA step to build quantum circuits as SAS data sets. Each quantum macro appends rows to the data set that represent gates, measurements, and other circuit

operations. After building a quantum circuit as a SAS data set, you can pass the resulting data set to execution macros, such as %Q_SAMPLE and %Q_OBSERVE, or directly to PROC QRUN.

# Qubits and Classical Bits

A quantum circuit operates on two types of bits: quantum bits (called qubits) and classical bits. Qubits hold the quantum state of the circuit and are the targets of gate operations. Classical bits store the results of measurements. A classical bit is always either 0 or 1, but a qubit can be in the state 0, the state 1, or a superposition of 0 and 1. Every qubit starts in the $|0\rangle$ state, and measuring a qubit gives a state 0 or 1, with probabilities that are determined by its state at the time of measurement.

You set the number of qubits and classical bits when you initialize a circuit by using the %Q_INIT(nq, nc) macro, where *nq* is the number of qubits and *nc* is the number of classical bits. SAS quantum macros use zero-based indexing, so a circuit that is initialized by using %Q_INIT(2, 2) has qubits 0 and 1 and classical bits 0 and 1. Here are some general tips to consider when you are determining qubits and classical bits:

- Set *nc* to 0 for circuits that you only want to observe. The %Q_OBSERVE macro works from the quantum state and does not require any classical bits.
- Set *nc* greater than 0 for any circuits that you sample. The macros %Q_SAMPLE, %Q_MEASURE, and %Q_MEASURE_ALL return errors on circuits with 0 classical bits.
- The number of qubits is not required to match the number of classical bits. Use fewer classical bits than qubits when you read out only some qubits. For example, if you want to read out only the ancilla or helper qubit in your circuit, you might require fewer classical bits than qubits.

# Quantum Gates

Quantum gates are the operations that transform the state of qubits. SAS quantum macros provide macros for single-qubit gates such as %Q_H (Hadamard) and %Q_X (Pauli-X or NOT), rotation gates such as %Q_RX and %Q_RY, and multi-qubit gates such as %Q_CX (CNOT) and %Q_CCX (Toffoli). Each gate macro appends a row to the circuit data set. The order in which you call the macros determines the order in which the gates are applied.

Here is an example that applies three gates in sequence to a single qubit:

```
data gate_order;
    %q_init(1, 1);
    %q_x(0);
    %q_h(0);
    %q_ry(0, - constant('pi') / 2);
    %q_measure(0, 0);
```

```
%q_sample(circuit=gate_order, out=work.gate_results, shots=1024);

proc print data=work.gate_results noobs; run;
```

The qubit starts in the |0⟩ state, is flipped to |1⟩ by a Pauli-X gate, is put into superposition by a Hadamard gate, and is then rotated toward |1⟩ by an RY rotation. The gates are applied in the order in which the macros are called; changing that order might produce a different final state.

For more information about specific gates, see "Quantum Gate Reference" on page 29.

# Measurements

Measurements collapse a qubit's quantum state into a classical value and store the result in a classical bit. Use the %Q_MEASURE(q, c) macro to measure a specific qubit into a specific classical bit. Use the %Q_MEASURE_ALL() macro to measure qubits into their matching classical bits. The number of measurements is limited by whichever is smaller: the number of qubits or the number of classical bits in the circuit. A circuit must have at least one classical bit and at least one measurement before you can sample it by using the %Q_SAMPLE macro. Circuits that are used only for expectation values with the %Q_OBSERVE macro do not require measurements. Here are more details about measurements in quantum circuits:

- Results are probabilistic. Each shot produces one bitstring. It is recommended that you sample many shots and read the counts and probabilities.
- Unmeasured qubits do not appear in the output. Only the qubits that you measure contribute to the results.
- Measurements are typically computed at the end of the circuit, after all the gates are defined.

# Parameters for Circuit Building

Quantum macros require parameters that instruct the macro how to perform. Many parameters are qubit indices, but some gates also require a value that controls the operation itself. The most common parameter other than a qubit index is an angle, measured in radians, that determines how much a rotation or phase gate changes the qubit's state. You can supply an angle as a number, a SAS expression, or a symbolic name that you bind at execution time.

Here is an example where the %Q_P(q, θ) macro takes a qubit index and a phase angle:

```
data phase_example;
    pi = constant('pi');
    %q_init(1, 1);
    %q_h(0);
    %q_p(0, pi);
```

```
    %q_h(0);
    %q_measure(0, 0);

%q_sample(circuit=phase_example, out=work.phase_results, shots=128);

proc print data=work.phase_results noobs; run;
```

The %Q_P macro inserts a phase gate into the circuit. The first argument for the %Q_P macro, `0`, is the index for the qubit that the gate acts on. The second argument, *pi*, is the phase angle. A phase gate leaves the |0⟩ state unchanged and shifts the phase of the |1⟩ state by the phase angle. The phase angle is a parameter because it changes the outcome, whereas the gate stays the same. In the preceding example, if you select the phase angle to be *pi*, the qubit ends in the |1⟩ state, but with a phase angle of `0`, the qubit returns to the |0⟩ state.

For more information about quantum macro gates and their parameters, see "Quantum Gate Reference" on page 29. To supply an angle at execution time instead of fixing it when you build the circuit, see "Running Parameterized Circuits" on page 41.

## Circuit Data Sets

SAS quantum macros represent a circuit as an ordinary SAS data set. The data set has a character column, _qasm_, and each row stores one line of the underlying OpenQASM program. You can inspect the circuit data set by using the PRINT procedure, transport it between sessions, and pass it directly to execution macros or PROC QRUN. The circuit data set must remain an ordinary SAS data set and must not be stored in CAS. A CAS table has no guaranteed row order, and the quantum circuit depends on row order to define the sequence of operations.

# Diagramming a Quantum Circuit

A circuit diagram shows the gates in a circuit data set. The diagram creates a visual representation of the circuit to help you verify the circuit before you run it. A diagram does not sample or observe the circuit.

## How to Diagram a Circuit

To diagram a circuit, use the %Q_SHOW_CIRCUIT macro. The diagram shows one wire for each qubit, and the gates are in the order in which you applied them. Here is an example of code that produces a circuit diagram:

```
data bell;
    %q_init(2, 2);
    %q_h(0);
```

```
    %q_cx(0, 1);
    %q_measure_all();

%q_show_circuit(circuit=bell);
```

Here are more details about the %Q_SHOW_CIRCUIT macro:

- The CIRCUIT= parameter is required in order to select the circuit to diagram.
- The %Q_SHOW_CIRCUIT macro does not run the circuit or create an output table.
- The provider and device that you select do not affect the diagram.
- The %Q_SHOW_CIRCUIT macro enables you to find errors, such as a gate on the wrong qubit or a missing measurement, before you run a circuit.
- The macro draws one circuit at a time. It does not support multiple circuit data sets.

**Note:** If you run PROC QRUN directly, you can specify the PLOTS=CIRCUIT option in the PROC QRUN statement to draw the same circuit diagram.

## Diagram Options

The %Q_SHOW_CIRCUIT macro sends the diagram to the Output Delivery System (ODS). You can use ODS statements to control where the diagram appears and how it is saved. In SAS Studio, the diagram appears in the results. To save the diagram to a file, specify the folder in the ODS LISTING statement with the GPATH= option as follows. Use the ODS GRAPHICS statement to name the file and select the format.

```
ods listing gpath="/home/user/diagrams";
ods graphics / imagename="bell_circuit" imagefmt=png;

%q_show_circuit(circuit=bell);

ods listing close;
```

In this example, the diagram is saved as bell_circuit.png in the **/home/user/diagrams** folder. Replace the path with a folder that you can write to.

# Parameterized Circuits

A parameterized circuit uses symbolic parameter names instead of fixed angle values. You build the circuit once, and then you supply the values when you run the circuit. Parameterized circuits enable you to run the same circuit with different angles without needing to rebuild or edit the circuit. Parameterized circuits are common in parameter sweeps and in variational algorithms, in which a program repeatedly adjusts angles to improve a result.

# Defining a Parameter

To define a parameter, use the %Q_INPUT(*parameter-name*) macro after the %Q_INIT macro. Supply the parameter name as the angle argument for any gate macro that accepts an angle. The following example defines an input named *theta* and uses it in an RY rotation:

```
data ry_circuit;
    %q_init(1, 1);
    %q_input(theta);
    %q_ry(0, theta);
    %q_measure(0, 0);

proc print data=ry_circuit noobs; run;
```

The %Q_INPUT macro defines *theta* as an input and the %Q_RY macro then refers to *theta* by name instead of using a numeric value. This data set defines a parameterized circuit. The *theta* input has no value yet; you supply values for it when you run the circuit.

Here is more information about defining a parameter:

- Define a parameter before you use it in a gate macro.
- After you set a name for the parameter, the macro treats it as a parameter and not as a SAS variable. The macro does not evaluate the name in the DATA step.
- Gate macros that accept an angle support parameters. For example, the %Q_RX, %Q_RY, %Q_RZ, %Q_P, and %Q_CP macros support parameters.
- A circuit can declare more than one parameter. Call the %Q_INPUT macro once to define each parameter.
- Angles are defined in radians.

# Binding Parameters

Binding assigns a value to each declared parameter when the circuit runs. To bind values, supply a CAS table with the DATA= parameter of an execution macro. The table has one column for each declared parameter, and each row is one set of values. The macro runs the circuit once for each row.

Here is some additional information about binding a parameter:

- A column name must match the name that you declare by using the %Q_INPUT macro.
- The DATA= table and the OUT= table must be CAS tables. The circuit data set must remain an ordinary SAS data set.
-

- Columns that do not match a declared input, such as angle_label, are copied to each output row. You can use these columns to match results to your inputs without depending on row order.
- Both the %Q_SAMPLE macro and the %Q_OBSERVE macro accept the DATA= option.
- You cannot use the DATA= option with a batch circuit data set. To run different circuits, use a batch. To run one circuit with different values, use the DATA= option.

For the DATA= table requirements, the output structure, and more information about parameterized execution, see "Running Parameterized Circuits" on page 41.

# Common Patterns

Here are some common patterns in quantum computing that use parameterized circuits:

- In a parameter sweep, you run one circuit over a range of values and observe how the result changes. In the example that follows, the probability of measuring 1 is 0% when *theta* is set to 0 and 100% when *theta* is set to π.
- In a variational circuit, you run the circuit, use the result to choose better values, and run the circuit again. Variational algorithms follow this pattern. The circuit stays the same, and only the values in the DATA= table change.
- In a circuit that has several parameters, you declare one input for each angle that you want to vary. Include one column for each input in the DATA= table.

# Example: Running a Circuit for Several Angle Values

This example shows you can build a circuit with one parameter, supply three values for that parameter, and run the circuit once for each value:

```
cas Qsession;
libname qlib cas sessref=Qsession;

data ry_circuit;
    %q_init(1, 1);
    %q_input(theta);
    %q_ry(0, theta);
    %q_measure(0, 0);

proc print data=ry_circuit noobs; run;

data qlib.theta_values;
    input angle_label $ theta;
    datalines;
```

```
zero      0
half_pi   1.5708
pi        3.14159
;

%q_sample(circuit=ry_circuit, data=qlib.theta_values,
          out=qlib.ry_results, shots=1024);

proc print data=qlib.ry_results noobs; run;
```

The example performs the following steps:

1. The CAS statement starts a CAS session named Qsession. The LIBNAME statement assigns a CAS library named qlib.
2. The first DATA step builds the circuit. The %Q_INPUT macro declares an input named *theta*, and the %Q_RY macro uses *theta* as its angle.
3. The first PROC PRINT step displays the circuit data set, which includes the declaration of *theta*.
4. The second DATA step creates a CAS table named qlib.theta_values. The table contains three values for *theta* and a label for each value.
5. The %Q_SAMPLE macro runs the circuit once for each row in the table, with 1,024 shots for each run. The macro writes the results to the qlib.ry_results table.
6. The second PROC PRINT step displays the results.

The results contain one set of rows for each value of *theta*. When you set *theta* to a value of 0, every shot measures 0. When you set *theta* to a value of approximately π/2, about half the shots measure 0 and half measure 1. When you set *theta* to a value of approximately π, every shot measures 1.

# Building Batch Circuits

A batch circuit data set stores several quantum circuits in one SAS data set. You can run all the circuits by using a single macro or procedure call, and the results are returned in one table. A batch is useful when you want to run and compare multiple related circuits, such as circuits with similar algorithms that contain slightly different gates.

To build a batch, use the %Q_INIT_BATCH macro in place of the %Q_INIT macro. Each call to the %Q_INIT_BATCH macro gives a name to a circuit and starts a new circuit. Add gate macros after the %Q_INIT_BATCH macro to add gates to that specific circuit in the batch. The following program builds a batch circuit that contains a Bell state circuit and a GHZ state circuit, and then it runs both:

```
data batch_circuits;
    %q_init_batch('bell', 2, 2);
    %q_h(0);
```

```
    %q_cx(0, 1);
    %q_measure_all();

    %q_init_batch('ghz', 3, 3);
    %q_h(0);
    %q_cx(0, 1);
    %q_cx(1, 2);
    %q_measure_all();

cas qsession;
libname qlib cas sessref=qsession;

%q_sample_batch(circuit=batch_circuits, out=qlib.batch_results,
shots=1024);

proc print data=qlib.batch_results noobs; run;
```

When you use the %Q_SAMPLE_BATCH macro, the results table has the same columns as the results table of the %Q_SAMPLE macro, plus a _circuit_id_ column that stores the circuit name. You see the results `00` and `11` for the **`bell`** circuit and the results `000` and `111` for the **`ghz`** circuit, each in about half of the shots. For more information about running a batch, see Batch Execution.

Here are tips for using batch circuits:

- Give each circuit a unique name. The name is the value of the _circuit_id_ column, which you use to match results to circuits.
- Circuits in a batch can have different numbers of qubits and classical bits when you sample them.
- To observe a batch, use the %Q_OBSERVE_BATCH or %Q_OBSERVE macro. Each circuit must have a name. Each Pauli string must have one letter for each qubit.
- You cannot use the DATA= parameter with a batch. To run one circuit that has several sets of angle values, use a parameterized circuit instead. For more information about parameterized circuits, see “Parameterized Circuits” on page 21 and “Running Parameterized Circuits” on page 41.
- The rows of a batch data set are in program order, and the rows for each circuit are grouped.
- A batch data set, like any other circuit data set, must remain an ordinary SAS data set and must not be stored in CAS. The OUT= table of the results is a CAS table. For more information about the batch data set, see “Batch Schema” on page 27.

# QASM Import and Export

OpenQASM is a quantum assembly programming language that describes quantum circuits. SAS quantum macros store each circuit as a data set of OpenQASM lines,

so you can export a circuit to a file and import a file into a circuit data set. Exporting enables you to share a circuit or use it in another tool. Importing enables you to run a circuit that another tool created.

Use the %Q_WRITE_QASM macro to export a circuit. Use the %Q_READ_QASM macro to import a circuit. Here is an example that exports a Bell state circuit to a file and then imports the file into a new data set:

```
data bell;
    %q_init(2, 2);
    %q_h(0);
    %q_cx(0, 1);
    %q_measure_all();

%q_write_qasm(circuit=bell, file=/tmp/bell.qasm);
%q_read_qasm(file=/tmp/bell.qasm, out=imported_bell);

proc print data=imported_bell noobs; run;
```

Here is more information about importing and exporting circuits:

- The %Q_WRITE_QASM macro requires the CIRCUIT= parameter that specifies the circuit data set and the FILE= parameter that specifies the path of the output file.
- The %Q_READ_QASM macro requires the FILE= parameter that specifies the path of the QASM file. The OUT= parameter names the new data set; the default name is circuit.
- The exported file uses OpenQASM 2.0 or OpenQASM 3.0.
- The imported data set has one row for each line in the QASM file. The lines match the lines of the circuit that you exported.
- You can use an imported data set in the same way that you use any other circuit data set. For example, you can pass it to the %Q_SAMPLE, %Q_SHOW_CIRCUIT, or %Q_WRITE_QASM macro.
- The imported data set must remain an ordinary SAS data set. Do not write it to a CAS library.
- The %Q_RAW macro writes a literal line of QASM into a circuit inside a DATA step. Use it for custom gate sequences or comments that are easier to write directly than by using the gate macros. The macro uses the same zero-based qubit indexing that the gate macros use. For more information, see "%Q_RAW Macro" on page 103.

# Circuit Data Model

SAS quantum macros represent a circuit as a SAS data set of OpenQASM 3.0 text. This section describes the columns of a circuit data set and of a batch data set, and it shows how macro calls appear as QASM lines.

# Circuit Schema

A circuit data set has one character column that holds the OpenQASM program, as shown in the following table:

***Table 2.1*** *Circuit Data Set Column*

| Column | Type | Length | Description |
|---|---|---|---|
| _qasm_ | Character | 500 | One line of the OpenQASM program |

The following information applies to the circuit schema:

- The order of the rows defines the circuit. For example, row 3 holds line 3 of the program, and row 6 holds line 6 of the program.
- The first rows are a header that the %Q_INIT macro writes. A circuit that has no gates still has several rows.
- The data set can also contain variables that you create in the DATA step.
- The circuit data set must remain an ordinary SAS data set and must not be stored in CAS.

# Batch Schema

A batch data set stores several circuits. To create one, use the %Q_INIT_BATCH macro. A batch data set adds one column to the circuit schema, as shown in the following table:

***Table 2.2*** *Batch Data Set Columns*

| Column | Type | Length | Description |
|---|---|---|---|
| _circuit_id_ | Character | 64 | Name of the circuit that the row belongs to |
| _qasm_ | Character | 500 | One line of the OpenQASM program |

The following information applies to the batch schema:

- Every row has a value for _circuit_id_.
- Each circuit has its own header and register lines so that each circuit can run separately.
- The rows for one circuit are grouped together, in program order.

- Circuits in a batch can have different numbers of qubits and classical bits.

# QASM Representation

A circuit contains the following parts, in this order:

- a header, which typically contains the OpenQASM version line, an include line for the standard OpenQASM library, and the definitions of some gates if they are used in that circuit.
- register declarations, such as `qubit[3] q;` and `bit[3] c;`
- input declarations, such as `input float[64] theta;`
- the operations, with one or more lines for each gate, measurement, or other macro call

The following table shows how macro calls appear as QASM lines in a circuit that contains three qubits:

***Table 2.3*** *Macro Calls and QASM Lines*

| Macro Call | QASM Line |
|---|---|
| %Q_H(0) | `h q[0];` |
| %Q_CX(0, 1) | `cx q[0],q[1];` |
| %Q_CCX(0, 1, 2) | `ccx q[0],q[1],q[2];` |
| %Q_SWAP(0, 1) | `swap q[0],q[1];` |
| %Q_RX(0, pi/2) | `rx(1.5707963268) q[0];` |
| %Q_RZZ(0, 1, pi/3) | `rzz(1.0471975512) q[0],q[1];` |
| %Q_U(2, pi/2, 0, pi) | `U(1.5707963268,0,3.1415926536) q[2];` |
| %Q_RY(1, theta) | `ry(theta) q[1];` |
| %Q_RY(1, beta * 2) | `ry(beta * 2) q[1];` |
| %Q_BARRIER() | `barrier q[0],q[1],q[2];` |
| %Q_RESET(2) | `reset q[2];` |
| %Q_MEASURE(0, 0) | `c[0] = measure q[0];` |
| %Q_RAW('// a comment') | `// a comment` |

The following information applies to the QASM lines:

- Qubit and classical bit numbers in the QASM text are the same zero-based indexes that you use in the macros.
- The macros calculate numeric angle expressions before they write the line, so *pi/2* becomes 1.5707963268.
- An expression that uses a name that you declare in the %Q_INPUT macro stays as text, as in `ry(beta * 2) q[1];`.
- The %Q_BARRIER macro writes a barrier on all the qubits in the circuit.
- The %Q_CSX and %Q_ECR macros write the definition of their gate into the program before the first call to the gate.
- Convenience macros such as %Q_H_ALL and %Q_CX_CHAIN write one line for each gate.
- The %Q_U macro writes the gate with an uppercase U.

# Quantum Gate Reference

## Single-Qubit Gates

You use the macros shown in Table 2.4 on page 29 in a DATA step to add the corresponding gate to the circuit. These gates operate on a single qubit within the circuit. Each macro function accepts an integer *q* that specifies the qubit that the gate should be applied to. Qubits are indexed starting at 0.

***Table 2.4*** *Single-Qubit Gate*

| Macro Name | Description |
|---|---|
| %Q_H (p. 98) | Adds a Hadamard gate to the circuit. |
| %Q_I (p. 99) | Adds an identity gate to the circuit. |
| %Q_S (p. 107) | Adds an S gate to the circuit. |
| %Q_SDG (p. 108) | Adds an S-dagger gate to the circuit. |
| %Q_T (p. 109) | Adds a T gate to the circuit. |
| %Q_TDG (p. 110) | Adds a T-dagger gate to the circuit. |

| Macro Name | Description |
|---|---|
| %Q_X (p. 113) | Adds an X gate to the circuit. This gate is also known as a NOT gate. |
| %Q_Y (p. 114) | Adds a Y gate to the circuit. |
| %Q_Z (p. 114) | Adds a Z gate to the circuit. |

# Rotational Gates

You use the macros shown in Table 2.5 on page 30 in a DATA step to add the corresponding gate to the circuit. These gates operate on a single qubit within the circuit. Each macro function accepts an integer *q* that specifies the qubit that the gate should be applied to. Qubits are indexed starting at 0. In addition, these gates take one or more arguments that specify the angle (or phase) that the qubit state should rotate. All angles are expressed in radians.

***Table 2.5*** *Rotational Gate Macros*

| Macro Name | Description |
|---|---|
| %Q_P (p. 102) | Adds a phase gate to the circuit |
| %Q_RX (p. 104) | Adds a rotational gate (X axis) to the circuit |
| %Q_RY (p. 105) | Adds a rotational gate (Y axis) to the circuit |
| %Q_RZ (p. 106) | Adds a rotational gate (Z axis) to the circuit |
| %Q_U (p. 111) | Adds a three-dimensional rotation gate to the circuit |
| %Q_U1 (p. 111) | Adds a one-dimensional rotation gate (Z axis) to the circuit |
| %Q_U2 (p. 112) | Adds a two-dimensional rotation gate (X and Z axes) to the circuit |
| %Q_U3 (p. 112) | Adds a three-dimensional rotation gate (Euler angles) to the circuit |

# Two-Qubit Gates

You use the macros shown in Table 2.6 on page 31 in a DATA step to add the corresponding gate to the circuit. These gates operate on two qubits within the circuit. Each macro function accepts two integers that specify which qubits the gate should be applied to. Qubits are indexed starting at 0.

Gates that are denoted as controlled gates distinguish between the control qubit *c* and the target qubit *t*. The quantum gate is applied to the target *t* qubit only when a control qubit *c* is in the |1> state. If the control qubit is in the |0> state, no action (the identity operation) is performed on the target *t*. If the control qubit is in a superposition state, this creates an entanglement between the qubits. Thus, the control qubit is also affected.

***Table 2.6*** *Two-Qubit Gate Macros*

| Macro Name | Description |
|---|---|
| %Q_CNOT (p. 91) | Adds a controlled NOT gate (CNOT) to the circuit. |
| %Q_CP (p. 91) | Adds a controlled phase gate to the circuit. |
| %Q_CRX (p. 92) | Adds a controlled rotational gate (X axis) to the circuit. |
| %Q_CRY (p. 92) | Adds a controlled rotational gate (Y axis) to the circuit. |
| %Q_CRZ (p. 93) | Adds a controlled rotational gate (Z axis) to the circuit. |
| %Q_CSX (p. 94) | Adds a controlled square root of X gate to the circuit. |
| %Q_CX (p. 94) | Adds a controlled X gate to the circuit. This gate is identical to the CNOT gate. |
| %Q_CY (p. 95) | Adds a controlled Y gate to the circuit. |
| %Q_CZ (p. 96) | Adds a controlled Z gate to the circuit. |
| %Q_ECR (p. 97) | Adds an echoed cross-resonance gate to the circuit. |
| %Q_SWAP (p. 108) | Adds a swap gate to the circuit. |

# Two-Qubit Ising Gates

You use the macros shown in Table 2.7 on page 32 in a DATA step to add the corresponding gate to the circuit. These gates operate on two qubits within the circuit. Each macro function accepts two integers that specify which qubits the gate should be applied to. Qubits are indexed starting at 0.

The Ising gates RXX, RYY, and RZZ implement pairwise spin-spin interactions from the Ising Hamiltonian. Each takes an angle parameter theta.

***Table 2.7*** *Two-Qubit Ising Gate Macros*

| Macro Name | Description |
|---|---|
| %Q_RXX (p. 105) | Adds an Ising gate to the circuit that is oriented for the X axis |
| %Q_RYY (p. 106) | Adds an Ising gate to the circuit that is oriented for the Y axis |
| %Q_RZZ (p. 107) | Adds an Ising gate to the circuit that is oriented for the Z axis |

# Three-Qubit Gates

You use the macros shown in Table 2.8 on page 32 in a DATA step to add the corresponding gate to the circuit. These gates operate on three qubits within the circuit. Each macro function accepts three integers that specify which qubits the gate should be applied to. Qubits are indexed starting at 0.

Gates that are labeled as controlled gates distinguish between the control qubit(s) and the target qubit(s). The gate is applied to the target qubit(s) only if the control qubit(s) is in the |1> state; the gate is not applied if the control qubit(s) is in the |0> state. If the control qubit(s) is in a superposition state, the gate application entangles the control and target qubit(s). Thus the states of both the control and target qubit(s) are affected.

***Table 2.8*** *Three-Qubit Gate Macros*

| Macro Name | Description |
|---|---|
| %Q_CCX (p. 90) | Adds a CCX gate (also known as a Toffoli gate) to the circuit |
| %Q_CSWAP (p. 93) | Adds a CSWAP gate (also known as a Fredkin gate) to the circuit |

| Macro Name | Description |
|---|---|
| %Q_FREDKIN (p. 98) | Adds a Fredkin gate (also known as a CSWAP gate) to the circuit |
| %Q_TOFFOLI (p. 110) | Adds a Toffoli gate (also known as a CCX gate) to the circuit |

# 3

# Running a Quantum Circuit

*Sampling Measurements from a Quantum Circuit* ... **35**
How to Sample Measurements ... 36
Interpreting Results of a Quantum Sample ... 36

*Computing Observables for a Quantum Circuit* ... **37**
About Pauli Observables ... 37
How to Compute Expected Values ... 38
Specifying Pauli Observables ... 39
Interpreting Results of an Observation ... 39
Computing Exact or Sampled Expected Values ... 40

*Running Parameterized Circuits* ... **41**
Requirements for Binding Parameters to a Circuit ... 41
Defining a Parameter-Binding Table ... 42
How to Run a Parameterized Circuit ... 43

*Preparing a Circuit for Real Hardware* ... **43**
Specifying a Device ... 44
Optimizing a Circuit ... 44
Results of Transpilation ... 44

*Running Circuits on Remote Hardware* ... **47**
Provider Options ... 47
Hardware Submission Safeguard ... 48
Example: Running a Circuit Remotely with PROC QRUN ... 48
Example: Using Macros to Switch Providers ... 49

*Authenticating to a Remote Provider* ... **49**
Authenticating by Using Environment Variables ... 49
Authenticating by Using Inline Credentials ... 50

## Sampling Measurements from a Quantum Circuit

To sample measurements from a quantum circuit means to run or simulate the circuit a number of times and report the results of the measurements that you

defined in the circuit. Because quantum results are inherently stochastic, it is important to run the circuit a number of times to determine its characteristic behavior. The number of times that you run the circuit is referred to as the number of shots.

# How to Sample Measurements

To sample measurements from a quantum circuit, you can use the SAMPLE statement on page 81 of PROC QRUN, or you can use one of the sampling macros: %Q_SAMPLE on page 130 or %Q_SAMPLE_BATCH on page 133.

The following code block shows how you can use the SAMPLE statement to sample a circuit:

```
cas casauto;                1
libname mycaslib cas;

data bell;                  2
    %q_init(2,2);
    %q_h(0);
    %q_cx(0, 1);
    %q_measure_all();
run;

proc qrun circuit=bell;     3
    sample out=mycaslib.counts numshots=1024;
run;
```

1 Initialize a CAS session as casauto and create a SAS library named mycaslib that is connected to that session. CAS is used because of its ability to process a large number of simulations in parallel.

2 Define the circuit that you want to sample. This circuit has two qubits and two classical bits to hold the measurement. When you use the %Q_MEASURE_ALL macro at the end of the circuit, the measured value of the first qubit is recorded in the first classical bit, and the value of the second qubit is recorded in the second classical bit.

3 Invoke the QRUN procedure on the circuit, and use the SAMPLE statement to request measurements to be sampled. You must specify a CAS table for the output, but the circuit data set does not need to be in CAS.

By default, if you do not specify a provider, the circuit is run as a simulation on the CAS server.

# Interpreting Results of a Quantum Sample

The results of quantum sampling are returned as a CAS table with the following schema:

*Table 3.1* *Schema of the Sampling Results*

| Column | Type | Description |
|---|---|---|
| _measurement_ | Character | Reports the measurement observed as a bit string. For example, a measurement might be reported as 00 or 11. |
| _count_ | Number | Reports the number of times that the outcome was observed. |
| _probability_ | Number | Reports the number of times that the outcome was observed in proportion to the total shot count |
| (Input columns) | mixed | Other columns that are copied when the DATA= option is used. |

The measurement is a bit string that denotes the classical bits that are reported from the simulation. Bit strings are reported in little-endian order. For example, if the bit string is 001, this means that bit 0 is 1 and bits 1 and 2 are 0. If you use the default %Q_MEASURE_ALL scheme in which the nth qubit is stored in the nth classical bit, this means that qubit 0 was measured as |1> and qubits 1 and 2 were measured as |0>.

**Note:** Impossible observations are not reported in the table. For example, the Bell state consists only of two possible measurements: |00> and |11>. Therefore, the sampling results of the Bell state do not include entries for |01> and |10> measurements.

# Computing Observables for a Quantum Circuit

Computing observables means computing the expected values of the Pauli operators. You can compute the expected values exactly, or you can request a shot-based estimate of the sample.

## About Pauli Observables

Historically, Pauli observables correspond to the axes of orientation of a particle's spin. However, the operators that represent orientation form a basis for any Hermitian operator for a qubit. This basis applies regardless of whether the qubit is implemented by using spin. Therefore, Pauli observables serve as a general-

purpose mechanism for computing observables of a quantum circuit. Mathematically, the Pauli operators are represented by the following matrices:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

The stated equations are for operators that apply to a single qubit. Operators that apply to multiple qubits use the outer product to combine single-qubit operators. For example, $Z \otimes I$ defines an operator that corresponds to a measurement of I on qubit 0 and Z on qubit 1. The outer product can be used as many times as needed to form an observable for the number of qubits in the circuit.

# How to Compute Expected Values

You can use the OBSERVE statement on page 80 of the QRUN procedure or the %Q_OBSERVE macro on page 123 or the %Q_OBSERVE_BATCH macro on page 126 to compute expected values.

The following code block shows how you can use the OBSERVE statement to compute expected values of Pauli observables from a circuit:

```
cas casauto;                1
libname mycaslib cas;

data bell;                  2
    %q_init(2,2);
    %q_h(0);
    %q_cx(0, 1);
    %q_measure_all();
run;

proc qrun circuit=bell; 3
    observe out=mycaslib.expected paulis="ZZ IX YY";
run;
```

1 Initialize a CAS session as casauto and create a SAS library named mycaslib that is connected to that session.

2 Define the circuit that you want to sample. This circuit has two qubits and two classical bits to hold the measurement. When you use the %Q_MEASURE_ALL macro at the end of the circuit, the measured value of the first qubit is recorded in the first classical bit, and the value of the second qubit is recorded in the second classical bit.

3 Invoke PROC QRUN on the circuit and use the OBSERVE statement to request which observables to compute. There are three Pauli terms to observe, and the output from each is summed to form a single aggregate expectation value. You must specify a CAS table for the output, but the circuit data set does not need to be in CAS.

# Specifying Pauli Observables

The PAULIS= option specifies the list of observables to evaluate as a space-delimited list of terms. Each term specifies a set of Pauli observables to observe for each qubit in the circuit.

The following list describes how you can express a variety of observables:

- Order observables to coincide with the little-endian format of qubits. For example, in a two-qubit circuit, PAULIS="ZI" computes expected values for the Z observable for qubit 1 and the I observable for qubit 0.
- You can specify multiple terms as a space-delimited list, such as PAULIS="XX IZ". By default, the resulting expected values are summed. However, if you use the DETAIL option of the OBSERVE statement, the expected value for each term is presented in its own row.
- You can scale each term in the Pauli option by using a colon, followed by the coefficient that you want to scale the term by. For example, PAULIS="ZZ:2" computes the Z expectation values on each of the two qubits and scales the result by a factor of 2. The default coefficient is 1.

By using coefficients and summing the Pauli terms, you can compute the expected value for arbitrary observables. For example, if for a given experiment you had the Hamiltonian $H = 2(Z \otimes Z) - 0.5(X \otimes X) + (Y \otimes Y)$ then you could compute it by specifying `PAULIS="ZZ:2 XX:-0.5 YY"` and omitting the DETAIL option.

# Interpreting Results of an Observation

The resulting observations are reported as a table that follows one of two schemas, which depend on whether you use the DETAIL option or omit it.

By default, observations are aggregate. This means that if you omit the DETAIL option, the _expectation_ column contains the expected value of the linear combination that you specified in the PAULIS= option. When you use this method, the following schema is used:

*Table 3.2* *Schema of Aggregated Observation Results*

| Column | Type | Description |
|---|---|---|
| _pauli_ | char | Reports the combined Pauli terms that describe the full Hamiltonian. |
| _coefficient_ | char | Reports a space-separated list of coefficients for the Pauli terms. |
| _expectation_ | num | Reports the aggregated expectation value.$\langle H \rangle = \sum c_i P_i$ |
| _weighted_ | num | Reports the same value as the expectation column. This column is included for consistency with the detailed output schema. |
| (Input columns) | mixed | Other columns that are copied when the DATA= option is used. |

If you use the DETAIL option, Pauli terms are reported on their own rows. When you use this method, the following schema is used:

*Table 3.3* *Schema of Detailed Observation Results*

| Column | Type | Description |
|---|---|---|
| _pauli_ | char | Reports the Pauli term of the row |
| _coefficient_ | num | Reports the coefficient for the Pauli terms |
| _expectation_ | num | Reports the unscaled expectation value of the term |
| _weighted_ | num | Reports the expectation value scaled by the coefficient |
| (Input columns) | mixed | Other columns that are copied when the DATA= option is used |

# Computing Exact or Sampled Expected Values

By default, observations are reported as exactly computed expected values. The values are computed directly from the circuit's state vector. Exact computations are supported only for the local provider, where simulations are not required.

You can also choose to run a simulation-based computation by specifying the NUMSHOTS= option. Specifying a higher number of shots in this option leads to a

more precise estimate but can use more computational resources. Using a sample-based observation on a local provider gives you the option to add noise to an otherwise ideal simulation. Finally, you can run the circuit on remote quantum hardware by specifying a nonlocal provider. For information about running circuits remotely, see "Running Circuits on Remote Hardware" on page 47.

# Running Parameterized Circuits

A parameterized circuit uses symbolic parameter names instead of fixed angle values. You build the circuit once, and then you supply the values when you run the circuit. Parameterized circuits enable you to run the same circuit with different angles without needing to rebuild or edit the circuit. Parameterized circuits are common in parameter sweeps and in variational algorithms, in which a program repeatedly adjusts angles to improve a result.

For information about how to build a parameterized circuit, see "Parameterized Circuits" on page 21.

## Requirements for Binding Parameters to a Circuit

Binding parameters to a circuit requires that you define a parameter-binding table. The following list highlights important considerations when you create a parameter-binding table and run the parameterized circuit:

- A column name must match the name that you declare by using the %Q_INPUT macro.
- The DATA= table and the OUT= table must be CAS tables. The circuit data set must remain an ordinary SAS data set.
- 
- Columns that do not match a declared input, such as angle_label, are copied to each output row. You can use these columns to match results to your inputs without depending on row order.
- Both the %Q_SAMPLE macro and the %Q_OBSERVE macro accept the DATA= option.
- You cannot use the DATA= option with a batch circuit data set. To run different circuits, use a batch. To run one circuit with different values, use the DATA= option.

# Defining a Parameter-Binding Table

Parameters are supplied to a circuit by using a parameter-binding table. A parameter-binding table is a data set that contains a column for each parameter in the table. The column names must match the name of the parameters that are specified in the %Q_INPUT macro. Each row of the table corresponds to a different set of values to bind to the circuit. Optionally, you can include additional columns to copy to the results table.

**Note:** The parameter-binding table must be a CAS table.

Suppose that you want to sweep over a set of values for a simple circuit that contains an RY gate that is defined as follows:

```
data ry_circuit;
    %q_init(1, 1);
    %q_input(theta);
    %q_ry(0, theta);
    %q_measure(0, 0);
run;
```

The circuit defines a single parameter *theta*, so you must define a parameter-binding table that contains, at minimum, a column named theta and one or more entries. The following DATA step defines an example parameter-binding table:

```
data mycaslib.parameters;
    input angle_label $ theta;
    datalines;
    zero      0
    half_pi   1.5708
    pi        3.14159
;
run;
```

In addition to the theta column, the parameter-binding table contains a column called angle_label that is copied into the result of any parameterized runs.

***Table 3.4*** *Example Parameter-Binding Table*

| angle_label | theta |
|---|---|
| zero | 0 |
| half_pi | 1.5708 |
| pi | 3.14159 |

# How to Run a Parameterized Circuit

To run a parameterized circuit, use PROC QRUN or a corresponding macro with the DATA= option. The DATA= option specifies the parameter-binding table to use for that run.

The following code snippet shows how you can use the previously defined circuit and parameter-binding table to generate samples:

```
proc qrun circuit=ry_circuit data=mycaslib.parameters;
    sample out=mycaslib.results;
run;
```

***Table 3.5*** *Example Results of Sampling a Parameterized Circuit*

| angle_lable | theta | _measurement_ | _count_ | _probability_ |
|---|---|---|---|---|
| zero | 0 | 0 | 1024 | 1 |
| half_pi | 1.578 | 0 | 496 | 0.484375 |
| half_pi | 1.5708 | 1 | 528 | 0.515625 |
| pi | 3.14159 | 1 | 1024 | 1 |

**Note:** When you use the NUMSHOTS= option, the specified number of shots are run for each row in the parameter-binding table.

# Preparing a Circuit for Real Hardware

A circuit that is built by using SAS quantum functionality can be designed independently of specific hardware considerations. For example, the circuit might use gates that a device does not natively have. Or the circuit might connect qubits that are not adjacent on a physical QPU. To prepare the circuit to run on physical hardware, you transpile it. Transpilation is the process of rewriting the circuit into a form that a specific device can run. Transpilation involves decomposing gates into the device's native set, routing qubits to match the device's connectivity, and optionally optimizing the gate depth. Practically, transpilation is also used to assess how expensive a circuit is to run before you dispatch it to a quantum computing provider.

Transpilation is a local process. It does not contact a quantum provider or require credentials. For information about running the circuit on real quantum hardware, see "Running Circuits on Remote Hardware" on page 47.

# Specifying a Device

To transpile a circuit, you must specify the device that you want to target. If you specify a provider in the PROC QRUN statement, then the TRANSPILATION statement defaults to using the device that you specify in the PROVIDER= option. However, if you want to transpile for a different device, you can specify the DEVICE= option as part of the TRANSPILE option.

The DEVICE= option accepts a device or topology name such as *ibm_brisbane*.

# Optimizing a Circuit

When you transpile a circuit, you can have the layout engine perform varying levels of optimization. This step is important to ensure that your circuit is well designed before you run it on real quantum hardware. You can use the outputs of the transpilation to guide your circuit design and optimization needs. Here are a few examples:

- If a transpiled circuit has a larger swap count than the original circuit, then that signals expensive routing. Consider revisiting the layout or circuit structure, because two-qubit gates often produce QPU errors.
- Reducing circuit depth usually also reduces two-qubit gate count, which is the dominant source of errors on real QPUs.

When you transpile the circuit, you can specify one of the following optimization levels:

0 Checks connectivity only. This value is not suitable for hardware runs.

1 Suited for most production use cases.

2 Suited for experiments that require higher fidelity than normal.

3 Applies the most compression. This level uses stochastic heuristics, so repeated runs can produce different results unless you also specify a seed.

Even transpiling with the OPTIMIZE=3 option, it is possible for a deep circuit to be too noisy to run usefully on hardware.

# Results of Transpilation

There are several possible results from transpilation, based on whether you use the OUT= option, the GATESOUT= option, or the METAOUT= option. The results of each option are ordinary SAS tables, not CAS tables.

## Circuit Output

The transpiled circuit is stored in the data set that you specify in the OUT= option. It has the following schema:

***Table 3.6*** *Schema for the Transpiled Circuit Results*

| Name | Type | Description |
|---|---|---|
| _qasm_ | char | Contains one row per line of the transpiled QASM |

The output circuit table is similar to a circuit table that you define by using gate macros in a DATA step. To preserve line order, it must be a SAS table, not a CAS table, and each line of QASM is put into its own row in the table. Unlike a generic circuit table, the transpiled circuit table is fit to a specific device, and it includes things like a pragma that specifies the mapping from logical qubits to physical qubits.

## Gate Output

When you transpile a circuit, you can also generate an inventory of the gates that are needed for the circuit to run. The gate inventory is produced when you specify the GATESOUT= option. The following table depicts a schema for the results:

***Table 3.7*** *Schema for the Gate Count Results*

| Name | Type | Description |
|---|---|---|
| _gate_ | char | Reports the type of gate in the circuit |
| _count_ | num | Reports the number of each gate in the transpiled circuit |

## Transpilation Metadata Output

You can retain the details of the transpilation by specifying the METAOUT= option. The following table depicts a schema of the results:

*Table 3.8* Schema for the Meta Results

| Name | Type | Description |
|---|---|---|
| _device_ | char | Reports the device that the circuit was transpiled for |
| _optimization_level_ | num | Reports the level of optimization that is used for the circuit |
| _depth_ | num, | Reports the depth (in gates) of the transpiled circuit |
| _num_qubits_ | num | Reports the number of qubits in the transpiled circuit |
| _original_depth_ | num | Reports the depth (in gates) of the original circuit |
| _transpilation_time_ | num | Reports the time (in seconds) that is needed to transpile the circuit |
| _layout_ | char | Reports the layout format that is used by the layout engine |
| _swap_count_ | num | Specifies the number of swap gates that are used for the transpiled circuit |
| _basis_gates_ | char | Reports the basis gates that are used for the transpiled circuit |

## Example: Conditionally Running a Deep Circuit

The following example checks the depth of a circuit before running it:

```
proc qrun circuit=mycircuit provider=ibm(device="ibm_brisbane"); /*1*/
    transpile out=ready metaout=tinfo;
run;

data _null_;
    set tinfo;
    call symputx('depth_ok', (_depth_ <= 100));                  /*2*/
run;

%if &depth_ok %then %do;                                         /*3*/
    proc qrun circuit=ready provider=ibm(device="ibm_brisbane");
        sample out=mycaslib.hw_counts numshots=2048;
    run;
%end;
%else %do;
    %put WARNING: Circuit too deep. Refine before hardware run.;
%end;
```

1. The initial run of PROC QRUN uses the TRANSPILE statement to produce a circuit table and a table of transpilation metadata named `tinfo`.
2. The DATA step reads the `tinfo` table. If the value in the _depth_ column is less than 100, it stores TRUE in the macro variable `depth_ok`.
3. If the value of `depth_ok` is TRUE, then a new PROC QRUN block runs a sample of the transpiled circuit. Otherwise, it prints a custom warning to the log.

# Running Circuits on Remote Hardware

The QRUN procedure and quantum macros enable you to run simulations locally, prepare the circuit to run on various real hardware, and then dispatch the circuits to real quantum computers. By default, all operations run on the local simulator. Operations are not sent for remote execution unless a provider is explicitly specified and hardware submission is enabled. These safeguards enable you to be conscientious about using potentially expensive remote computation.

Before running your circuit on quantum hardware, be sure to transpile the circuit for the specific device that you are targeting. For information about transpiling the circuit, see "Preparing a Circuit for Real Hardware" on page 43.

## Provider Options

You can specify which QPU provider runs the circuit by using the PROVIDER= option of PROC QRUN. When you specify a provider, you also specify provider-specific options, such as a particular device that a provider offers.

Here is additional information about using providers:

- When no provider is specified, circuits are run on the local simulator.
- You can explicitly specify a local provider. Explicitly specifying a local provider enables you to set a device name for local runs.
- No default device is selected for you. You must explicitly choose which device to use.
- Authentication setup varies for each provider. For information about authenticating, see see "Authenticating to a Remote Provider" on page 49.

To see provider-specific options for PROC QRUN, see "PROVIDER=*provider-options*" on page 78.

# Hardware Submission Safeguard

By default, sampling and observing operations are restricted to local execution; transpilation is always local. To enable running sampling or quantum observations on quantum hardware, use the ALLOWHARDWARE option in the PROC QRUN statement. The ALLOWHARDWARE option enables the use of remote hardware for the duration of the procedure.

# Example: Running a Circuit Remotely with PROC QRUN

The following code example shows how you can run a circuit on IBM's Brisbane QPU:

```
proc qrun circuit=mycircuit provider=ibm(device="ibm_brisbane"); /*1*/
    transpile out=transpiled
optimize=2;                                                    /*2*/
run;

proc qrun circuit=transpiled provider=ibm(device="ibm_brisbane")
allowhardware; /*3*/
    sample out=mycaslib.counts numshots=1024;                        /*4
run;
```

1 The PROVIDER= option specifies the provider and the device that are targeted when the circuit is transpiled.

2 The TRANSPILE statement creates a circuit named *transpiled* that targets the previously specified device.

3 The PROVIDER= option is repeated to specify which provider to use to run the circuit. When you run on quantum hardware, be sure to select the transpiled circuit. Specify ALLOWHARDWARE to enable remote hardware execution for the duration of the procedure.

4 The SAMPLE statement uses the CIRCUIT= option to specify the transpiled circuit. Because a nonlocal provider is specified and the ALLOWHARDWARE option is specified, the transpiled circuit is run remotely.

**Note:** This example enables all the options that are required to run on actual hardware. If you are authenticated and run this example without errors, it can incur real quantum computation cost.

## Example: Using Macros to Switch Providers

The following code snippet shows how you can use macro definitions to toggle between different providers:

```
* %let target = provider=ibm(device="ibm_brisbane");
%let target = provider=local(device="generic_5q_fake"); /*1*/
* %let target = provider=local;

proc qrun circuit=bell &target;                          /*2*/
   sample out=mycaslib.counts numshots=4096;
run;
```

1 Uncomment whichever macro variable you want to use as the provider. Comment out any remaining macro definitions.

2 The `&target` macro variable resolves to the provider that you defined.

# Authenticating to a Remote Provider

To run quantum circuits by using a remote provider, you must authenticate your credentials with that provider. You can specify credentials by using provider-specific environment variables or by specifying them inline in the PROVIDER= option. The inline option uses a hardcoded string and is discouraged for programs that are shared, but it can be a useful fallback option. No authentication is needed when you specify the PROVIDER=LOCAL option.

## Authenticating by Using Environment Variables

To use environment variables, you must specify a device, but you do not need to specify any provider-specific authentication options. When you do not specify inline credentials, environment variables are automatically checked, and if valid credentials are found, they are used.

**Note:** Before a circuit is run on remote hardware, you also need to specify the ALLOWHARDWARE option. For more information about running a circuit on remote hardware, see “Running Circuits on Remote Hardware” on page 47.

*Table 3.9* *Environment Variables for IBM*

| Environment Variable | Description |
|---|---|
| IBM_QUANTUM_CHANNEL | Specifies the IBM connection channel |
| IBM_QUANTUM_INSTANCE | Specifies the IBM instance routing |
| IBM_QUANTUM_TOKEN | Specifies the IBM API token |

# Authenticating by Using Inline Credentials

Credentials that you specify in the PROVIDER= option take precedence over those that you set in environment variables.

> **IMPORTANT** Supplying credentials inline puts them in the source code and in the SAS log. Be sure to remove private or personal information before code is shared, checked in, or scheduled.

The following provider-specific options are used for the IBM provider:

*Table 3.10* *Provider Options for IBM*

| Provider Option | Description |
|---|---|
| CHANNEL= | Specifies the IBM connection channel |
| INSTANCE= | Specifies the IBM instance routing |
| TOKEN= | Specifies the IBM instance routing |

The following code snippet illustrates how you can use the PROVIDER= option to supply inline credentials to an IBM device:

```
proc qrun circuit=mycircuit provider=ibm(device="ibm_brisbane"
instance="hub/group/project" channel="ibm_cloud");
    sample out=mycaslib.counts numshots=4096;
run;
```

# 4

# HYPERTUNE Procedure

*Overview: HYPERTUNE Procedure* . . . . . ***51***
Overview of the HYPERTUNE Procedure . . . . . 51

*Syntax: HYPERTUNE Procedure* . . . . . ***53***
PROC HYPERTUNE Statement . . . . . 54
CON Statement . . . . . 57
LINCON Statement . . . . . 57
OBJ Statement . . . . . 59
SUBMIT Statement . . . . . 59
VAR Statement . . . . . 60

*Usage: HYPERTUNE Procedure* . . . . . ***61***
Getting Started with the HYPERTUNE Procedure . . . . . 61
Details about the HYPERTUNE Procedure . . . . . 63

*Examples: HYPERTUNE Procedure* . . . . . ***68***
Example 1: Minimizing a Single Objective . . . . . 68
Example 2: Linear and Nonlinear Constraints . . . . . 69
Example 3: Multiobjective Optimization . . . . . 71
Example 4: Tuning a Quantum Procedure . . . . . 73

## Overview: HYPERTUNE Procedure

### Overview of the HYPERTUNE Procedure

The HYPERTUNE procedure is a general-purpose black-box (derivative-free) optimization solver. Many practical problems share a common structure: you choose values for a set of decision variables, run a costly piece of code, observe one or more objective values, and repeat. Machine learning pipelines search for the best hyperparameters, simulation studies sweep design parameters, and variational

quantum algorithms tune rotation angles. In each case, the objective function is expensive to evaluate and has no usable closed form or derivative information.

A problem is a *black box* when the only way to learn about the objective is to evaluate it: you supply a candidate set of variable values, the code runs, and a result is returned. You cannot inspect an algebraic formula, and you cannot compute a gradient to indicate which direction improves the objective. Conventional nonlinear programming methods, which rely on derivative information, do not apply. Black-box optimizers instead learn about the objective by sampling it strategically, balancing *exploration* (probing unfamiliar regions of the variable space) against *exploitation* (refining the search near the best points that have been found so far). Because each evaluation can be costly, the central concern is to find good solutions in as few evaluations as possible.

PROC HYPERTUNE closes this loop natively in SAS. You declare decision variables with bounds and types, name one or more objectives to minimize or maximize, optionally declare constraints, and supply a SUBMIT block that contains the code that evaluates the objectives and nonlinear constraints. The procedure drives the search by using the algorithm that you select, caches every evaluation to avoid repeated work, and writes the best solutions to output data sets. You can write the code in the SUBMIT block as a SAS macro.

The essential distinction between PROC HYPERTUNE and a conventional SAS optimization procedure, such as those in SAS/OR software, is where the objective lives. In a conventional procedure, you write the objective as an algebraic expression that the procedure parses and differentiates. In PROC HYPERTUNE, the objective is opaque code that the procedure only executes. This makes PROC HYPERTUNE applicable to a broad class of problems for which no closed-form objective exists:

Hyperparameter tuning
: Search for the settings of a modeling procedure (for example, a kernel width or a regularization strength) that optimize a validation metric. The SUBMIT block trains the model and returns the metric.

Simulation-based optimization
: Tune the inputs of a stochastic simulation to optimize an average outcome, where the objective is available only as the output of a simulation run.

Algorithm and process configuration
: Choose configuration parameters that minimize a run time, a cost, or an error rate that you can measure but not write as a formula.

Scientific parameter tuning
: Optimize the parameters of a physical model or a variational quantum circuit, where the objective is the result of a numerical experiment.

Because the SUBMIT block can contain arbitrary SAS code, including a call to another SAS procedure such as PROC HPSVM or PROC QCIRCUIT, the same procedure provides classical hyperparameter tuning, simulation-based optimization, and quantum variational algorithms without any change to its interface.

PROC HYPERTUNE also supports problems that have more than one objective. When several objectives conflict, there is generally no single best solution; instead there is a Pareto front of solutions, each of which represents a different trade-off

among the objectives. The procedure identifies and returns a set of these nondominated solutions so that you can choose among the trade-offs after the run.

PROC HYPERTUNE provides the following features:

- single-objective and multiobjective optimization
- continuous, integer, and nominal decision variables
- variable scaling (linear, logarithmic, and factor-based)
- linear and nonlinear constraints
- objectives that are expressed in SAS through a SUBMIT block
- a choice of four optimization algorithms (EAGLS, random, LHS, grid)
- a persistent evaluation process that loads the SUBMIT code once and then calls it repeatedly, avoiding per-evaluation startup overhead
- a global evaluation cache that reuses repeated points and that can be saved and reloaded to warm-start later runs
- parallel evaluation

**Note:** PROC HYPERTUNE is built on the same proven black-box optimization engine that powers other SAS offerings, including the Autotune CAS action set, the AUTOTUNE statement in several procedures, and the solveBlackbox CAS action. As a result, PROC HYPERTUNE delivers the same optimization algorithms and search behavior that are available within these established tools, now exposed through a general-purpose procedural interface.

The remainder of this chapter is organized as follows. The section Syntax: HYPERTUNE Procedure documents every statement and option. The section Getting Started: HYPERTUNE Procedure on page 61 presents a first example. The section Details: HYPERTUNE Procedure on page 63 describes the problem formulation, the algorithms, the evaluation cache, and the output. The section Examples: HYPERTUNE Procedure on page 68 provides working examples.

# Syntax: HYPERTUNE Procedure

**PROC HYPERTUNE** *hypertune-options*;
  **VAR** *variable* / *variable-options*;
  **OBJ** *objective* / *objective-options*;
  **CON** *constraint* / *constraint-options*;
  **LINCON** DATA=*data-set*
  **SUBMIT**; *code* **ENDSUBMIT**;

# PROC HYPERTUNE Statement

The PROC HYPERTUNE statement invokes the procedure and specifies global options that control the algorithm, execution limits, convergence, evaluation cache, input and output data sets, and logging.

## Syntax

**PROC HYPERTUNE** *hypertune-options*;

### Algorithm Options

**ALGORITHM=EAGLS | GRID | LHS | RANDOM**
specifies the optimization algorithm.

Default EAGLS

| | |
|---|---|
| EAGLS | performs evolutionary algorithm guiding local search. |
| GRID | performs a grid search. |
| LHS | performs Latin hypercube sampling. |
| RANDOM | performs a random search. |

### Convergence Options

**ABSFCONV=*number***
specifies the absolute function convergence criterion. The procedure stops when the improvement in the best objective value between successive iterations falls below the specified value.

Default 1E-6

Range 0 or greater

**FEASTOL=*number***
specifies the feasibility tolerance for constraints. A point whose constraint violation does not exceed the specified value is treated as feasible.

Default 1E-3

Range 0 or greater

**NABSFCONV=*integer***
specifies the number of consecutive iterations without improvement that are required before the procedure declares convergence and terminates.

Default 4

## Execution Limit Options

**MAXFUNC=*integer***
specifies the number of objective function evaluations.

Range 1 or greater

**MAXITER=*integer***
specifies the maximum number of iterations. For population-based algorithms, this is the number of generations.

Default 10

Range 1 or greater

**MAXTIME=*integer***
specifies the maximum time (in seconds) that the procedure runs.

Default unlimited

Range 0 or greater

**NPARALLEL=*integer***
specifies the number of objective evaluations to perform in parallel.

Default 1

**SEED=*integer***
specifies the pseudorandom number seed, which controls the reproducibility of stochastic algorithms.

Range 0 or greater

## Population and Search Control Options

**NGLOBAL=*integer***
specifies the number of global searches.

Default 1

**NLOCAL=*integer***
specifies the number of local searches.

Default 2p where p is the number of decision variables.

**PARETOMAX=*integer***
specifies the maximum number of Pareto-optimal points to retain.

Default 5000

**POPSIZE=*integer***
specifies the population size for population-based algorithms.

Default 10

## Cache Management Options

**CACHEIN=*data-set***
names an input data set of previously computed evaluations. Points in this data set are loaded into the cache before the search begins, which warm-starts the run.

**CACHEOUT=*data-set***
names an output data set that receives the full evaluation cache: every distinct point that is evaluated, together with its objective values, constraint values, and feasibility status.

**CACHETOL=*number***
specifies the tolerance that is used to determine whether a candidate point matches a point already in the cache. The value must be nonnegative.

Default 1E-9

**CACHEMAX=*integer***
specifies the maximum number of entries that the cache retains.

Default 1,000,000 / p, where p is a proxy for the size of the optimization problem and is computed from the number of decision variables, objectives, and constraints.

## Input and Output Data Set Options

**FIRSTGEN=*data-set***
names an input data set of starting points for the search.

**LASTGEN=*data-set***
names an output data set of the points in the final iteration.

**PRIMALIN=*data-set***
names an input data set of initial solutions.

**PRIMALOUT=*data-set***
names the output data set of the best solution(s) that is found.

## Logging Options

**LOGFREQ=*integer***
specifies that the procedure print an iteration log line every $n$ iterations.

Default 1

**LOGLEVEL=0 | 1**
specifies the amount of information to write to the SAS log. Specify LOGLEVEL=0 to suppress log messages or LOGLEVEL=1 for the standard log.

Default 1

**PRINTLEVEL=0 | 1**
specifies the amount of ODS output. Specify PRINTLEVEL=0 to suppress the ODS tables or PRINTLEVEL=1 to create the summary tables.

Default 1

# CON Statement

The CON statement declares one nonlinear constraint. Specify one CON statement for each nonlinear constraint. The constraint name must match the name of a SAS macro in the SUBMIT block that computes the constraint value. Nonlinear constraint functions are evaluated together with the objective functions during each evaluation.

## Syntax

CON *constraint* </ options>;

### Optional Arguments

**LB=*number***
specifies the lower bound on the constraint value.

**UB=*number***
specifies the upper bound on the constraint value.

## Example

The following statements illustrate how to use the CON statement:

```
con c1 / lb=0 ub=100;
con c2 / lb=-10;
```

# LINCON Statement

The LINCON statement specifies a SAS data set that defines linear constraints. Use this statement for algebraic linear constraints that can be supplied as data rather than computed in the SUBMIT block.

## Syntax

LINCON **DATA=***data-set* ;

## Details

The LINCON statement specifies a SAS data set that defines linear constraints. Use this statement for algebraic linear constraints that can be supplied as data rather than computed in the SUBMIT block.

The LINCON data set is row-oriented. Each observation defines one linear constraint and can contain the following variables:

- one or more numeric variables whose names match VAR names; these are the linear coefficients. A coefficient that is missing or 0 is treated as absent.
- _LB_ and _UB_, which give the respective lower and upper bounds on the linear combination. For the linear constraint to be meaningful, at least one _LB_ or _UB_ variable must be present.
- an optional character variable _ID_ that identifies the constraint.

The bounds are interpreted the same way as in the CON statement. For example, you can set the following constraints:

$$x_1 + x_2 \leq 5$$
$$x_1 - x_2 \geq 0$$
$$2x_1 + x_2 = 3$$

You use the following DATA step to set these constraints:

```
data lincon_data;
   length id $32;
   input _id_ $ _lb_ x1 x2 _ub_;
   datalines;
budget    .   1   1   5
balance   0   1  -1   .
target    3   2   1   3
;
run;

proc hypertune ... ;
   var x1 / lb=0 ub=5;
   var x2 / lb=0 ub=5;
   ...
   lincon data=lincon_data;
   ...
run;
```

# OBJ Statement

The OBJ statement declares one objective function. Specify one OBJ statement for each objective; specifying more than one objective produces a multiobjective (Pareto) problem. The objective name must match the name of a SAS macro that is defined in the SUBMIT block.

## Syntax

**OBJ** </ *options*>;

### Optional Argument

**TYPE=MIN | MAX**
specifies whether to minimize or maximize the objective.

Default MIN

## Example

The following statements illustrate how to use the OBJ statement:

```
obj obj1;
obj obj2 / type=max;
obj loss / type=min;
```

# SUBMIT Statement

The SUBMIT block contains the SAS macro code that evaluates the objectives and nonlinear constraints. The code runs once per candidate solution: PROC HYPERTUNE passes the decision variable values in and reads the objective and constraint values out.

## Syntax

**SUBMIT**;
*code*
**ENDSUBMIT**;

## Details

The procedure runs in persistent mode. The SUBMIT block code is loaded once, and the objective and constraint functions are then called repeatedly with different argument values.

The SUBMIT block can contain SAS statements. The decision variable values are available as macro variables (&x1, &x2) inside the submitted code. Define one SAS macro for each objective and each nonlinear constraint. Each macro receives the decision variables as positional parameters and must write its result to a global macro variable:

- For an objective, use the name *objective*_val.
- For a constraint, use the name *constraint*_val.

For example:

```
%macro profit(x1, x2);
   %let profit_val = %sysevalf(&x1 * &x2);
%mend;
```

# VAR Statement

The VAR statement declares one decision variable. Specify one VAR statement for each variable in the problem. The specified variable name must match a parameter name of the objective function in the SUBMIT block.

## Syntax

**VAR** *variable* </ options>;

### Optional Arguments

**LB=*number***
specifies the lower bound of the variable.

**UB=*number***
specifies the upper bound of the variable.

**INIT=*number***
specifies the initial value of the variable.

**TYPE=C | I | N**
specifies the variable type.

C specifies a continuous variable.

I specifies an integer variable.

N specifies a nominal variable.

Default C

**SCALE=FACTOR | LINEAR | LN | LOG10 | NONE**
specifies how to scale the variable during the search.

| | |
|---|---|
| FACTOR | uses a multiplicative factor that the SCALEFACTOR= option supplies. |
| LINEAR | uses linear scaling. |
| LN | uses base e logarithmic scaling. |
| LOG10 | uses base-10 logarithmic scaling. |
| NONE | leaves the variable unscaled. |

Default NONE

**SCALEFACTOR=*number***
specifies the scaling factor to use when SCALE=FACTOR.

## Example

The following statements illustrate how to use the VAR statement:

```
var x1 / lb=0 ub=10 type=I init=0;
var x2 / lb=7 type=C init=10 scale=FACTOR scalefactor=2.0;
var x3 / lb=1 ub=10 type=N;
```

# Usage: HYPERTUNE Procedure

## Getting Started with the HYPERTUNE Procedure

The following statements minimize the two-dimensional Rosenbrock function $f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2$ over the region $[-2, 2] \times [-1, 3]$:

```
proc hypertune seed=1 primalout=best;
  var x1 / lb=-2 ub=2;
  var x2 / lb=-1 ub=3;
  obj rosen / type=min;
  submit;
    %macro rosen(x1,x2);
      %let rosen_val = %sysevalf((1 - &x1)**2 + 100*(&x2 - &x1**2)**2);
    %mend;
  endsubmit;
```

```
run;

proc print data=best; run;
```

The VAR statements declare the two decision variables and their lower and upper bounds. The OBJ statement names the objective and requests minimization. The SUBMIT block defines a SAS macro named rosen, whose name matches the objective and whose argument names match the variable names. PROC HYPERTUNE calls this function repeatedly, using the default EAGLS algorithm to select points, and stops after 200 function evaluations. The best solution that it finds is written to the best data set by the PRIMALOUT= option.

The procedure produces a number of tables . The "Problem Summary" table reports the problem type and the number of variables, constraints, and objectives. The "Option Summary" table reports the settings that control the run. The "Solution Summary" table reports the number of iterations and evaluations that are performed, and how many of those evaluations were served from the cache. The "Performance Information" table reports the execution mode and parallel processing information.

The HYPERTUNE Procedure

Problem Summary

| | |
|---|---|
| Problem Type | NLP |
| Number of Variables | 2 |
| Continuous Variables | 2 |
| Integer Variables | 0 |
| Number of Constraints | 0 |
| Linear Constraints | 0 |
| Nonlinear Constraints | 0 |
| Number of Objectives | 1 |
| Objective Sense | Minimize |

Option Summary

| | |
|---|---|
| Convergence Tolerance | 1E-6 |
| Cache Max Size | 500000 |
| Cache Tolerance | 1E-9 |
| Feasibility Tolerance | 0.001 |
| Max Func Evaluations | 120000 |
| Max Iterations | 10 |
| Max Time | 1.797693E08 |
| Num Global Solvers | 1 |
| Num Local Solvers | 4 |
| Population Size | 20 |
| Seed | 1 |
| Log Level | 1 |

Solution Summary

| | |
|---|---|
| Solution Status | Maximum iterations |
| Objective Value | 0 |
| Infeasibility | 0 |
| Nondominated | 0 |

| | |
|---|---|
| Progress | 0 |
| Iterations | 10 |
| Evaluations | 162 |
| Cached Evaluations | 6 |

The PRINT procedure then displays the best data set, which contains the decision variable values and the objective value at the optimal solution:

| Obs | _tag_ | X1 | X2 | ROSEN |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 |

# Details about the HYPERTUNE Procedure

## Problem Formulation

PROC HYPERTUNE solves problems of the following form. Let $x = (x_1, \ldots, x_p)$ be the vector of decision variables, each of which has a lower bound $l_j$, an upper bound $u_j$, and a type (continuous, integer, or nominal). The procedure searches for values of $x$ that optimize one or more objective functions, $\text{opt } f_i(x), \quad i = 1, \ldots, m$, where each $f_i$ is either minimized or maximized, subject to the bound constraints $l_j \le x_j \le u_j$, any nonlinear constraints $L_k \le c_k(x) \le U_k$ from the CON statements, and any linear constraints from the LINCON statement. The objective functions $f_i$ and the nonlinear constraint functions $c_k$ are treated as black boxes: they are evaluated by the code in the SUBMIT block, and no gradient information is assumed.

## Optimization Algorithms

The ALGORITHM= option selects the search strategy. The available algorithms fall into two broad families:

Population-based methods
: maintain and evolve a set of candidate solutions. EAGLS is an example.

Sampling methods
: draw points from the variable space without using prior evaluations to guide the search. GRID, LHS, and RANDOM are examples.

The remainder of this section describes each available algorithm in detail. In choosing an algorithm, weigh how expensive each evaluation is, how many evaluations your budget allows, whether the variables are continuous or discrete, and whether you seek a single global optimum or a local refinement of a known point.

EAGLS: Evolutionary Algorithm Guiding Local Search
: EAGLS is a population-based global optimizer that combines an evolutionary algorithm (genetic algorithm) with a local search algorithm (generating set search). It maintains a population of candidate solutions that evolve across generations through crossover and mutation, whereas a local-search component refines promising individuals to accelerate convergence. This hybrid of global exploration by the population and local exploitation by the search component makes EAGLS well suited to multimodal problems, to mixed continuous-integer variables, and to multiobjective problems in which a Pareto front is sought. For EAGLS, the MAXITER= option sets the maximum number of iterations and the POPSIZE= option sets the population size; the NGLOBAL= and NLOCAL= options control the amount of global and local search, respectively. This is the default search strategy that PROC HYPERTUNE uses.

GRID: Grid Search
: Grid search evaluates the objective at the points of a regular grid that is laid over the variable bounds. It is exhaustive and reproducible, and it is well suited to low-dimensional problems and to discrete or nominal variables that take a small number of values. Its cost grows exponentially with the number of variables, so grid search becomes impractical as the dimension increases.

LHS: Latin Hypercube Sampling
: Latin hypercube sampling (LHS) is a structured space-filling design. The range of each variable is divided into equally probable intervals, and the sample points are placed so that each interval of each variable is represented exactly once. The result covers the variable space more evenly than independent random sampling of the same size; this makes LHS an efficient choice for an initial design or for building a representative cache. Like random search, LHS does not adapt to the objective values that it observes.

RANDOM: Random Search
: Random search draws each candidate point independently and uniformly at random from within the variable bounds. It uses no information from prior evaluations, so it neither exploits structure nor gets trapped in a local optimum. Random search can be a good baseline and is trivially parallel. Because it does not concentrate its samples, it typically needs many evaluations to locate a precise optimum.

The following table summarizes the trade-offs in choosing an algorithm.

***Table 4.1*** *Choosing an Algorithm*

| **If you want to ...** | **Consider These Algorithms** |
|---|---|
| Establish a baseline or seed the cache | RANDOM, LHS |
| Enumerate a small or discrete space exhaustively | GRID |
| Solve a multimodal, mixed-integer, or multiobjective problem | EAGLS |

## Variable Types and Scaling

Each decision variable has a type: continuous (TYPE=C option), integer (TYPE=I option), or nominal (TYPE=N option). Integer variables are restricted to integer values within their bounds. Nominal variables take values from a discrete set.

The SCALE= option transforms a variable before the optimizer works with it; this can improve the search when variables span very different magnitudes. SCALE=LOG10 and SCALE=LN apply base-10 and natural-logarithm scaling, respectively, and are appropriate for variables that range over several orders of magnitude. SCALE=FACTOR multiplies by the value that is specified in the SCALEFACTOR= option. SCALE=LINEAR applies linear scaling, and SCALE=NONE (the default) leaves the variable unscaled.

## The Evaluation Cache

Because objective evaluations can be expensive, PROC HYPERTUNE caches every distinct point that it evaluates, together with the objective values, constraint values, and feasibility status. When a candidate point matches a cached point within the tolerance that is specified by the CACHETOL= option, the procedure reuses the cached result instead of reevaluating the SUBMIT code. The CACHEMAX= option limits the number of cache entries.

The cache can be persisted and reused. Specify the CACHEOUT= option to write the cache to a data set at the end of the run, and specify the CACHEIN= option to load a previously saved cache before a run begins. Loading a cache warm-starts the search and can substantially reduce the number of new evaluations.

## Persistent Evaluation and the SUBMIT Block

PROC HYPERTUNE evaluates the SUBMIT block code in a persistent child process. The code is parsed and loaded once; thereafter the objective and constraint functions are called repeatedly with different argument values. This model avoids the overhead of reinitializing the SAS session on every evaluation.

## Multiobjective Optimization and the Pareto Front

When you specify more than one OBJ statement, PROC HYPERTUNE solves a multiobjective optimization problem. Rather than being a single optimum, the result is a set of Pareto-optimal (nondominated) solutions: that is, solutions for which no objective can be improved without worsening another. The procedure retains up to the number of such solutions that you specify in the PARETOMAX= option and

writes them to the primary output data set. The iteration log reports the number of nondominated points that have been found so far.

## Parallel Evaluation

PROC HYPERTUNE supports solving optimization problems by using multiple levels of parallelism. This means that it can be configured to execute objective function evaluations in parallel while at the same time allowing each of the objective function evaluations to execute in a parallel fashion. The NPARALLEL= option specifies how many objective evaluations you want to execute in parallel. In addition to this PROC HYPERTUNE option, you can write your objective function code in the SUBMIT block to also make use of parallel evaluation threads. You can do this in a variety of ways (such as by using the NTHREADS= option in a given SAS procedure) within your SUBMIT block.

Imagine a scenario in which PROC HYPERTUNE is called with a value of 4 for the NPARALLEL= option and the SAS code within the SUBMIT block calls PROC QRUN with a value of 3 for the NTHREADS= option. Each concurrent evaluation permits up to three PROC QRUN data-row processing threads per CAS worker. Actual total thread usage depends on the workload, number of available action threads, and number of CAS workers. It is extremely important to be aware of the computing resources available in the particular environment that you are using so that PROC HYPERTUNE is not configured in a way that would overload the computing environment.

## Input and Output Data Sets

The following table summarizes the data sets that PROC HYPERTUNE reads and writes.

***Table 4.2*** *Input and Output Data Sets*

| Role | Statement or Option |
|---|---|
| Linear constraints (input) | LINCON |
| Initial primal solutions (input) | PRIMALIN= |
| Cached evaluations (input) | CACHEIN= |
| Starting points (input) | FIRSTGEN= |
| Best solutions (output) | PRIMALOUT= |
| Full evaluation cache (output) | CACHEOUT= |
| Final iteration points (output) | LASTGEN= |

Output data sets that record per-evaluation results include the decision variable values, objective, and constraint values.

## Displayed Output

Unless you specify the PRINTLEVEL=0 option, PROC HYPERTUNE produces the following ODS tables:

Problem Summary
: reports the problem type; the number of variables (and how many are continuous and integer); the number of constraints (linear and nonlinear); the number of objectives and their sense; and the initial sample set, input cache, and first-generation data sets when they are specified.

Option Summary
: reports the settings that control the run, such as the convergence tolerance; the cache size and tolerance; the feasibility tolerance; the maximum function evaluations, iterations, and time; the number of global and local solvers; the population size; the seed; and the log level.

Solution Summary
: reports the solution status, the final objective value, the infeasibility, the number of iterations and evaluations, the number of cached evaluations, and the number of best solutions that are retained.

PROC HYPERTUNE assigns a name to each table that it creates. You can use these names to select or exclude tables through the Output Delivery System (ODS).

***Table 4.3*** *ODS Tables for PROC HYPERTUNE*

| ODS Table Name | Description | Statement or Option |
|---|---|---|
| ProblemSummary | Problem summary | Default (PRINTLEVEL>=1) |
| OptionSummary | Option summary | Default (PRINTLEVEL>=1) |
| SolutionSummary | Solution summary | Default (PRINTLEVEL>=1) |

For each logged iteration, the iteration log (controlled by the LOGLEVEL= and LOGFREQ= options) reports the iteration number, the best objective that has been found so far, the infeasibility, the cumulative number of evaluations, and the total elapsed time.

# Examples: HYPERTUNE Procedure

## Example 1: Minimizing a Single Objective

This example shows how to use the HYPERTUNE procedure to minimize a single objective.

The following statements minimize a simple quadratic objective and write the best solution to a data set named Best:

```
proc hypertune seed=1 primalout=Best;
    var x1 / lb=0 ub=5;
    var x2 / lb=0 ub=5;
    obj loss / type=min;

    submit;
        %macro loss (x1, x2);
        %let loss_val = %sysevalf((&x1 - 2)**2 + (&x2 - 1)**2);
        %mend;
    endsubmit;
run;
```

The following results show the the ODS tables that PROC HYPERTUNE code produces:

The HYPERTUNE Procedure

Problem Summary

| | |
|---|---|
| Problem Type | NLP |
| Number of Variables | 2 |
| Continuous Variables | 2 |
| Integer Variables | 0 |
| Number of Constraints | 0 |
| Linear Constraints | 0 |
| Nonlinear Constraints | 0 |
| Number of Objectives | 1 |
| Objective Sense | Minimize |

Option Summary

| | |
|---|---|
| Convergence Tolerance | 0.0001 |
| Cache Max Size | 333333 |
| Cache Tolerance | 1E-9 |
| Feasibility Tolerance | 0.001 |

| | |
|---|---|
| Max Func Evaluations | 1220 |
| Max Iterations | 10 |
| Max Time | 1.797693E308 |
| Num Global Solvers | 1 |
| Num Local Solvers | 4 |
| Population Size | 10 |
| Seed | 1 |
| Log Level | 1 |

Solution Summary

| Solution Status | Maximum iterations |
|---|---|
| Objective | 0.0013074939 |
| Infeasibility | 0 |
| Iterations | 10 |
| Evaluations | 107 |
| Cached Evaluations | 1 |

# Example 2: Linear and Nonlinear Constraints

## Linear and Nonlinear Constraints

This example shows how to use PROC HYPERTUNE to solve a problem that has both linear and nonlinear constraints and a linear objective. The optimization problem is described as follows:

Maximize:

$$f(x) = 9x_1 + 15x_2 - 6x_3 - 16x_4 - 10x_5$$

Subject to:

$$\begin{aligned} x_3 + x_4 &= x_6 + x_7 \\ x_6 + x_8 &= x_1 \\ x_7 + x_9 &= x_2 \\ x_8 + x_9 &= x_5 \\ 2.5x_1 - x_{10}x_6 - 2x_8 &\geq 0 \\ 1.5x_2 - x_{10}x_7 - 2x_9 &\geq 0 \\ 3x_3 + x_4 - x_{10}(x_3 + x_4) &= 0 \end{aligned}$$

And subject to:

$$\begin{aligned} &0 \leq x_1 \leq 100 \\ &0 \leq x_2 \leq 200 \\ &1 \leq x_{10} \leq 3 \\ &0 \leq x_i, \text{ for } i = 3, \ldots, 9 \end{aligned}$$

The PROC HYPERTUNE code to solve this optimization problem looks like this:

```
/* A data set defining the linear constraints */
data lindata;
   input _id_ $ _lb_ x1-x10 _ub_;
   datalines;
   c1  0  0  0  1  1  0 -1 -1  0  0  0  0
   c2  0 -1  0  0  0  0  1  0  1  0  0  0
   c3  0  0 -1  0  0  0  0  1  0  1  0  0
   c4  0  0  0  0  0 -1  0  0  1  1  0  0
;

proc hypertune
   primalout=p_out
   popsize=160
   maxiter=200
   seed=1234
   nparallel=16
   nabsfconv=10;

   /* Define the optimization variables */
   var x1  / lb=0 ub=100;
   var x2  / lb=0 ub=200;
   var x3  / lb=0;
   var x4  / lb=0;
   var x5  / lb=0;
   var x6  / lb=0;
   var x7  / lb=0;
   var x8  / lb=0;
   var x9  / lb=0;
   var x10 / lb=1 ub=3;

   obj obj1 / type=max;

   lincon data=lindata;

   con c5 / lb=0;
   con c6 / lb=0;
   con c7 / lb=0 ub=0;

   submit;
      /* Compute the objective value: obj1 */
      %macro obj1 (x1, x2, x3, x4, x5, x6, x7, x8, x9, x10);
         %let obj1_val = %sysevalf(9*&x1 + 15*&x2 - 6*&x3 - 16*&x4 -
10*&x5);
      %mend;

      /* Compute the nonlinear constraint values: c5, c6, c7 */
      %macro c5 (x1, x2, x3, x4, x5, x6, x7, x8, x9, x10);
         %let c5_val = %sysevalf(2.5*&x1 - &x10*&x6 - 2*&x8);
      %mend;

      %macro c6 (x1, x2, x3, x4, x5, x6, x7, x8, x9, x10);
         %let c6_val = %sysevalf(1.5*&x2 - &x10*&x7 - 2*&x9);
      %mend;

      %macro c7 (x1, x2, x3, x4, x5, x6, x7, x8, x9, x10);
```

```
            %let c7_val = %sysevalf(3*&x3 + &x4 - &x10*&x3 - &x10*&x4);
         %mend;
      endsubmit;
   run;
```

The following shows the ODS tables that are produced by this PROC HYPERTUNE code.

The HYPERTUNE Procedure

Problem Summary

| | |
|---|---|
| Problem Type | NLP |
| Number of Variables | 10 |
| Continuous Variables | 10 |
| Integer Variables | 0 |
| Number of Constraints | 7 |
| Linear Constraints | 4 |
| Nonlinear Constraints | 3 |
| Number of Objectives | 1 |
| Objective Sense | Maximize |

Option Summary

| | |
|---|---|
| Convergence Tolerance | 0.0001 |
| Cache Max Size | 58823 |
| Cache Tolerance | 1E-9 |
| Feasibility Tolerance | 0.001 |
| Max Func Evaluations | 592000 |
| Max Iterations | 200 |
| Max Time | 1.797693E308 |
| Num Global Solvers | 1 |
| Num Local Solvers | 20 |
| Population Size | 160 |
| Seed | 1234 |
| Log Level | 1 |

Solution Summary

| | |
|---|---|
| Solution Status | Function convergence |
| Objective | 400.05414652 |
| Infeasibility | 0.0009860459 |
| Iterations | 76 |
| Evaluations | 8780 |
| Cached Evaluations | 926 |

# Example 3: Multiobjective Optimization

This example shows how to use PROC HYPERTUNE to optimize multiple nonlinear objectives. The problem minimizes the following:

$$f_1(x) = (x_1 - 1)^2 + (x_1 - x_2)^2$$
$$f_2(x) = (x_1 - x_2)^2 + (x_2 - 3)^2$$

Subject to:

$$0 \leq x_1, x_2 \leq 5.$$

The following code shows how to represent this multiobjective optimization problem in PROC HYPERTUNE syntax:

```
proc hypertune
   popsize=80
   nparallel=8
   primalout=pareto;
   var x1 / lb=0 ub=5;
   var x2 / lb=0 ub=5;
   obj obj1 / type=min;
   obj obj2 / type=min;
   submit;

      %macro obj1 (x1, x2);
         %let obj1_val = %sysevalf((&x1 - 1)**2 + (&x1 - &x2)**2);
      %mend;

      %macro obj2 (x1, x2);
         %let obj2_val = %sysevalf((&x1 - &x2)**2 + (&x2 - 3)**2);
      %mend;

   endsubmit;
run;

proc sgplot data=pareto;
   scatter x=obj1 y=obj2;
run;
```

This code produces the following output:

The HYPERTUNE Procedure

Problem Summary

| | |
|---|---|
| Problem Type | NLP |
| Number of Variables | 2 |
| Continuous Variables | 2 |
| Integer Variables | 0 |
| Number of Constraints | 0 |
| Linear Constraints | 0 |
| Nonlinear Constraints | 0 |
| Number of Objectives | 2 |
| Objective Sense | Minimize |

Option Summary

| | |
|---|---|
| Convergence Tolerance | 0.0001 |
| Cache Max Size | 250000 |
| Cache Tolerance | 1E-9 |
| Feasibility Tolerance | 0.001 |

| | |
|---|---|
| Max Func Evaluations | 1920 |
| Max Iterations | 10 |
| Max Time | 1.797693E308 |
| Num Global Solvers | 1 |
| Num Local Solvers | 4 |
| Population Size | 80 |
| Seed | 1 |
| Log Level | 1 |

Solution Summary

| Solution Status | Maximum iterations |
|---|---|
| Nondominated | 135 |
| Progress | 0.0087098288 |
| Infeasibility | 0 |
| Iterations | 10 |
| Evaluations | 558 |
| Cached Evaluations | 8 |

***Figure 4.1*** *Plot of Pareto-Optimal Set*

# Example 4: Tuning a Quantum Procedure

This example shows how to use PROC HYPERTUNE to tune a single parameter, theta, in a simulated quantum computing workflow. The procedure searches for a value of theta that minimizes the objective, energy. For each candidate value, PROC HYPERTUNE calls the energy macro, which uses the %Q_INIT and %Q_RY macros to build a data set that describes a single-qubit quantum circuit. The parameter theta specifies the rotation angle (in radians) for the circuit's Y-axis rotation. PROC QRUN simulates the circuit and computes its Pauli-Z expectation value, which is used here as the energy. The macro stores this result in the variable energy_val, which PROC HYPERTUNE reads to guide its search.

```
proc hypertune
   maxfunc=100
   primalout=quantum_best;
   var theta / lb=0 ub=6.283185307179586 type=C;
   obj energy / type=MIN;
   submit;

      %macro energy(theta);
         libname mycas cas;

         data ry_objective;
            %q_init(1, 1);
            %q_ry(0, &theta);
         run;

         proc qrun circuit=ry_objective simmethod=statevector;
```

```
            observe out=mycas.z_energy paulis="Z" detail;
         run;

         data _null_;
            set mycas.z_energy;
            call symputx('energy_val', _expectation_, 'G');
         run;
      %mend;

   endsubmit;
run;

proc print data=quantum_best;
run;
```

This PROC HYPERTUNE code produces the following output:

The HYPERTUNE Procedure

Problem Summary

| | |
|---|---|
| Problem Type | NLP |
| Number of Variables | 1 |
| Continuous Variables | 1 |
| Integer Variables | 0 |
| Number of Constraints | 0 |
| Linear Constraints | 0 |
| Nonlinear Constraints | 0 |
| Number of Objectives | 1 |
| Objective Sense | Minimize |

Option Summary

| | |
|---|---|
| Convergence Tolerance | 0.0001 |
| Cache Max Size | 500000 |
| Cache Tolerance | 1E-9 |
| Feasibility Tolerance | 0.001 |
| Max Func Evaluations | 100 |
| Max Iterations | 10 |
| Max Time | 1.797693E308 |
| Num Global Solvers | 1 |
| Num Local Solvers | 2 |
| Population Size | 10 |
| Seed | 1 |
| Log Level | 1 |

Solution Summary

| | |
|---|---|
| Solution Status | Function convergence |
| Objective | -0.999595736 |
| Infeasibility | 0 |
| Iterations | 6 |
| Evaluations | 46 |
| Cached Evaluations | 2 |

| Obs | _tag_ | THETA | ENERGY |
|---|---|---|---|
| 1 | 0 | 3.11316 | -0.99960 |

You can see from printing the quantum_best output table that PROC HYPERTUNE successfully found the values of theta = 3.11316 and energy = -0.99960. This result closely matches the theoretical minimum of -1 at theta = π.

# 5

# QRUN Procedure

*Overview: QRUN Procedure* . . . . . *77*
About the QRUN Procedure . . . . . 77
*Syntax: QRUN Procedure* . . . . . *77*
PROC QRUN Statement . . . . . 78
OBSERVE Statement . . . . . 80
SAMPLE Statement . . . . . 81
TRANSPILE Statement . . . . . 81

## Overview: QRUN Procedure

### About the QRUN Procedure

The QRUN procedure executes quantum circuit data sets that are built by using SAS quantum macros. You can use the procedure to sample measurement outcomes, compute Pauli expectation value, render circuit or quantum-state visualizations, and transpile circuits for a target device. The default target is the local simulator; you select the cloud hardware or cloud simulators by using the PROVIDER= option in the PROC QRUN statement.

## Syntax: QRUN Procedure

**PROC QRUN** CIRCUIT=*libref.data-table* <*proc-qrun-options*>;

**OBSERVE** OUT=*libref.data-table*PAULIS=*list* <*observer-options*>;

**SAMPLE** OUT=*libref.data-table* <sample-options>;

**TRANSPILE** <*transpile-options*>;

# PROC QRUN Statement

The PROC QRUN statement specifies the circuit input, execution target, and other procedure-level options.

## Syntax

**PROC QRUN** CIRCUIT=*libref.data-table* <DATA=*libref.data-table*><NOISE=*character*><NTHREADS=*integer*<PROVIDER=*provider-options*>>;

## Summary of Optional Arguments

DATA=*libref.data-table*
: specifies a parameter-binding table, which runs one parameter set per input row and preserves other columns as copyvars.

NOISE=*string*
: specifies a simulator noise model.

NTHREADS=*integer*
: specifies the maximum number of threads to allocate per worker.

PROVIDER=*provider-options*
: specifies the execution provider.

SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR
: specifies the representation of the local simulation.

### Required Argument

**CIRCUIT=*libref.data-table***
: specifies the name of the two-level circuit data set to execute, display, or transpile.

### Optional Arguments

**DATA=*libref.data-table***
: specifies a parameter-binding table, which runs one parameter set per input row and preserves other columns as copyvars.

Restriction Mutually exclusive with batch input.

**PROVIDER=*provider-options***
: specifies the execution provider. You can specify the following provider-options:

**IBM(*ibm-options*)**
runs your circuit on IBM hardware with the specified options.

You can specify the following *ibm-options*:

**CHANNEL=*char***
specifies the IBM connection channel.

**DEVICE=**
specifies the device or simulator to run on. You must specify this suboption whenever you specify a named provider. The DEVICE= option is not required when you use the local provider.

**INSTANCE=*char***
specifies the IBM instance routing.

**TOKEN=*char***
specifies the IBM token directly in SAS code.

See "Authenticating to a Remote Provider" on page 49

Important Use of the TOKEN= option is discouraged, but it is useful as a fallback authentication method.

**LOCAL(*local-options*)**

**DEVICE=**
specifies the device or simulator to run on. You must specify this suboption whenever you specify a named provider.

**NOISE=*string***
specifies a simulator noise model.

**NTHREADS=*integer***
specifies the maximum number of threads to allocate per worker.

Default 1

Range 1 or greater

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

# OBSERVE Statement

The OBSERVE statement computes the Pauli expectation values of the observables.

## Syntax

**OBSERVE** OUT=*libref.data-table* PAULIS=*observables-list*; <*observe-options*>

### Required Arguments

**OUT=*libref.data-table***
specifies the output data set that expectation values are written to.

**PAULIS=*observables-list***
specifies the list of observables to evaluate as a space-delimited list of terms. Each term specifies the Pauli observables to observe for each qubit in the circuit. For example, in a three-qubit circuit, `PAULIS="IXX"` computes expectation values for the identity on qubit 2 and for the X observable on the other two qubits. It can have an optional coefficient suffix. You can specify multiple terms as a space-delimited list, such as `PAULIS="IXX ZIZ"`.

You can scale the expectation value of an observable by following the observable with a colon and the coefficient that you want to scale it by. For example, `PAULIS="ZZ:2"` computes the Z expectation values on each of the two qubits and scales the result by a factor of 2. The default coefficient is 1.

### Optional Arguments

**DETAIL**
requests one row per Pauli term rather than a single combined result.

**NUMSHOTS=*integer***
requests a sampling-based estimate. Samples are computed based on of the specified number of shots. By default, an exact local computation is performed.

Range 1 or greater

# SAMPLE Statement

The SAMPLE statement runs the circuit and returns measurement outcomes.

## Syntax

**SAMPLE** OUT=*libref.data-table*;

### Required Argument

**OUT=*libref.data-table***
specifies the output data set that measurement counts are written to.

### Optional Argument

**NUMSHOTS=*integer***
specifies the number of repeated circuit runs.

Default 1024

Range 1 or greater

# TRANSPILE Statement

The TRANSPILE statement rewrites a circuit for a target device without executing it.

## Syntax

**TRANSPILE** *options*;

## Summary of Optional Arguments

DEVICE=*character*
specifies the target device of the transpilation.

GATESOUT=*libref.data-table*
specifies the output data set for the transpiled circuit in quantum assembly language (QASM) row format.

MAXTIME=*integer*

specifies the maximum amount of time (in seconds) to wait for transpilation.

METAOUT=*libref.data-table*
specifies the output data set for transpilation summary statistics.

OPTIMIZE=*integer*
specifies the optimization effort. Higher *integers* signify more effort.

OUT=*libref.data-table*
specifies the output data set for the per-gate counts of the transpiled circuit.

SEED=*integer*
specifies a seed to make stochastic transpilation reproducible.

## Optional Arguments

**DEVICE=*character***
specifies the target device of the transpilation. This option is passed through and validated by the provider.

Default Depends on the value of the PROVIDER= option.

**GATESOUT=*libref.data-table***
specifies the output data set for the transpiled circuit in quantum assembly language (QASM) row format.

Restriction You must specify at least one GATESOUT=, METAOUT=, or OUT= option.

**MAXTIME=*integer***
specifies the maximum amount of time (in seconds) to wait for transpilation.

Default 120

Range 1 or greater

**METAOUT=*libref.data-table***
specifies the output data set for transpilation summary statistics.

Restriction You must specify at least one GATESOUT=, METAOUT=, or OUT= option.

**OPTIMIZE=*integer***
specifies the optimization effort. Higher *integers* signify more effort.

Range 0–3

0 Checks connectivity only. This value is not suitable for hardware runs.

1 Suited for most production use cases.

2 Suited for experiments that require higher fidelity than normal.

3 Applies the most compression. This level uses stochastic heuristics, so repeated runs can produce different results unless you also specify a seed.

**OUT=*libref.data-table***
specifies the output data set for the per-gate counts of the transpiled circuit.

Restriction You must specify at least one GATESOUT=, METAOUT=, or OUT= option.

**SEED=*integer***
specifies a seed to make stochastic transpilation reproducible. By default, a pseudorandom number seed is used.

Range 0 or greater

# 6

# Quantum Gate Macros

*Quantum Gate Reference* . . . . . 86
Single-Qubit Gates . . . . . 86
Rotational Gates . . . . . 86
Two-Qubit Gates . . . . . 86
Two-Qubit Ising Gates . . . . . 86
Three-Qubit Gates . . . . . 86
*%Q_BARRIER Macro* . . . . . 90
*%Q_CCX Macro* . . . . . 90
*%Q_CNOT Macro* . . . . . 91
*%Q_CP Macro* . . . . . 91
*%Q_CRX Macro* . . . . . 92
*%Q_CRY Macro* . . . . . 92
*%Q_CRZ Macro* . . . . . 93
*%Q_CSWAP Macro* . . . . . 93
*%Q_CSX Macro* . . . . . 94
*%Q_CX Macro* . . . . . 94
*%Q_CX_CHAIN Macro* . . . . . 95
*%Q_CY Macro* . . . . . 95
*%Q_CZ Macro* . . . . . 96
*%Q_CZ_CHAIN Macro* . . . . . 96
*%Q_CZ_RING Macro* . . . . . 97
*%Q_ECR Macro* . . . . . 97
*%Q_FREDKIN Macro* . . . . . 98
*%Q_H Macro* . . . . . 98
*%Q_H_ALL Macro* . . . . . 99
*%Q_I Macro* . . . . . 99
*%Q_INIT Macro* . . . . . 99

%Q_INIT_BATCH Macro 100
%Q_INPUT Macro 101
%Q_MEASURE Macro 101
%Q_MEASURE_ALL Macro 102
%Q_P Macro 102
%Q_RAW Macro 103
%Q_RESET Macro 104
%Q_RX Macro 104
%Q_RXX Macro 105
%Q_RY Macro 105
%Q_RYY Macro 106
%Q_RZ Macro 106
%Q_RZZ Macro 107
%Q_S Macro 107
%Q_SWAP Macro 108
%Q_SDG Macro 108
%Q_SX Macro 109
%Q_T Macro 109
%Q_TDG Macro 110
%Q_TOFFOLI Macro 110
%Q_U Macro 111
%Q_U1 Macro 111
%Q_U2 Macro 112
%Q_U3 Macro 112
%Q_X Macro 113
%Q_Y Macro 114
%Q_Z Macro 114

# Quantum Gate Reference

## Single-Qubit Gates

You use the macros shown in Table 6.22 on page 87 in a DATA step to add the corresponding gate to the circuit. These gates operate on a single qubit within the

circuit. Each macro function accepts an integer *q* that specifies the qubit that the gate should be applied to. Qubits are indexed starting at 0.

***Table 6.1*** *Single-Qubit Gate*

| Macro Name | Description |
|---|---|
| %Q_H (p. 98) | Adds a Hadamard gate to the circuit. |
| %Q_I (p. 99) | Adds an identity gate to the circuit. |
| %Q_S (p. 107) | Adds an S gate to the circuit. |
| %Q_SDG (p. 108) | Adds an S-dagger gate to the circuit. |
| %Q_T (p. 109) | Adds a T gate to the circuit. |
| %Q_TDG (p. 110) | Adds a T-dagger gate to the circuit. |
| %Q_X (p. 113) | Adds an X gate to the circuit. This gate is also known as a NOT gate. |
| %Q_Y (p. 114) | Adds a Y gate to the circuit. |
| %Q_Z (p. 114) | Adds a Z gate to the circuit. |

# Rotational Gates

You use the macros shown in Table 6.23 on page 87 in a DATA step to add the corresponding gate to the circuit. These gates operate on a single qubit within the circuit. Each macro function accepts an integer *q* that specifies the qubit that the gate should be applied to. Qubits are indexed starting at 0. In addition, these gates take one or more arguments that specify the angle (or phase) that the qubit state should rotate. All angles are expressed in radians.

***Table 6.2*** *Rotational Gate Macros*

| Macro Name | Description |
|---|---|
| %Q_P (p. 102) | Adds a phase gate to the circuit |
| %Q_RX (p. 104) | Adds a rotational gate (X axis) to the circuit |
| %Q_RY (p. 105) | Adds a rotational gate (Y axis) to the circuit |

| Macro Name | Description |
|---|---|
| %Q_RZ (p. 106) | Adds a rotational gate (Z axis) to the circuit |
| %Q_U (p. 111) | Adds a three-dimensional rotation gate to the circuit |
| %Q_U1 (p. 111) | Adds a one-dimensional rotation gate (Z axis) to the circuit |
| %Q_U2 (p. 112) | Adds a two-dimensional rotation gate (X and Z axes) to the circuit |
| %Q_U3 (p. 112) | Adds a three-dimensional rotation gate (Euler angles) to the circuit |

# Two-Qubit Gates

You use the macros shown in Table 6.24 on page 88 in a DATA step to add the corresponding gate to the circuit. These gates operate on two qubits within the circuit. Each macro function accepts two integers that specify which qubits the gate should be applied to. Qubits are indexed starting at 0.

Gates that are denoted as controlled gates distinguish between the control qubit *c* and the target qubit *t*. The quantum gate is applied to the target *t* qubit only when a control qubit *c* is in the |1> state. If the control qubit is in the |0> state, no action (the identity operation) is performed on the target *t*. If the control qubit is in a superposition state, this creates an entanglement between the qubits. Thus, the control qubit is also affected.

***Table 6.3*** *Two-Qubit Gate Macros*

| Macro Name | Description |
|---|---|
| %Q_CNOT (p. 91) | Adds a controlled NOT gate (CNOT) to the circuit. |
| %Q_CP (p. 91) | Adds a controlled phase gate to the circuit. |
| %Q_CRX (p. 92) | Adds a controlled rotational gate (X axis) to the circuit. |
| %Q_CRY (p. 92) | Adds a controlled rotational gate (Y axis) to the circuit. |
| %Q_CRZ (p. 93) | Adds a controlled rotational gate (Z axis) to the circuit. |
| %Q_CSX (p. 94) | Adds a controlled square root of X gate to the circuit. |
| %Q_CX (p. 94) | Adds a controlled X gate to the circuit. This gate is identical to the CNOT gate. |

| Macro Name | Description |
|---|---|
| %Q_CY (p. 95) | Adds a controlled Y gate to the circuit. |
| %Q_CZ (p. 96) | Adds a controlled Z gate to the circuit. |
| %Q_ECR (p. 97) | Adds an echoed cross-resonance gate to the circuit. |
| %Q_SWAP (p. 108) | Adds a swap gate to the circuit. |

# Two-Qubit Ising Gates

You use the macros shown in Table 6.25 on page 89 in a DATA step to add the corresponding gate to the circuit. These gates operate on two qubits within the circuit. Each macro function accepts two integers that specify which qubits the gate should be applied to. Qubits are indexed starting at 0.

The Ising gates RXX, RYY, and RZZ implement pairwise spin-spin interactions from the Ising Hamiltonian. Each takes an angle parameter theta.

***Table 6.4*** *Two-Qubit Ising Gate Macros*

| Macro Name | Description |
|---|---|
| %Q_RXX (p. 105) | Adds an Ising gate to the circuit that is oriented for the X axis |
| %Q_RYY (p. 106) | Adds an Ising gate to the circuit that is oriented for the Y axis |
| %Q_RZZ (p. 107) | Adds an Ising gate to the circuit that is oriented for the Z axis |

# Three-Qubit Gates

You use the macros shown in Table 6.26 on page 90 in a DATA step to add the corresponding gate to the circuit. These gates operate on three qubits within the circuit. Each macro function accepts three integers that specify which qubits the gate should be applied to. Qubits are indexed starting at 0.

Gates that are labeled as controlled gates distinguish between the control qubit(s) and the target qubit(s). The gate is applied to the target qubit(s) only if the control qubit(s) is in the |1> state; the gate is not applied if the control qubit(s) is in the |0> state. If the control qubit(s) is in a superposition state, the gate application

entangles the control and target qubit(s). Thus the states of both the control and target qubit(s) are affected.

***Table 6.5*** *Three-Qubit Gate Macros*

| Macro Name | Description |
|---|---|
| %Q_CCX (p. 90) | Adds a CCX gate (also known as a Toffoli gate) to the circuit |
| %Q_CSWAP (p. 93) | Adds a CSWAP gate (also known as a Fredkin gate) to the circuit |
| %Q_FREDKIN (p. 98) | Adds a Fredkin gate (also known as a CSWAP gate) to the circuit |
| %Q_TOFFOLI (p. 110) | Adds a Toffoli gate (also known as a CCX gate) to the circuit |

# %Q_BARRIER Macro

inserts a barrier into the circuit. Optimizations are prevented across barriers.

## Syntax

```
%Q_BARRIER();
```

## Details

During transpilation, logical gates from your circuit are replaced with gates that are available on the specified device. Whenever you transpile a circuit, you can specify varying levels of optimization. Including a barrier in your circuit enables optimization to occur on both sides of the barrier, but not across it.

# %Q_CCX Macro

adds a CCX gate to the circuit.

Alias: %Q_TOFFOLI

## Syntax

%Q_CCX(*c1*, *c2*, *t*);

### Required Arguments

**c1=*integer***
specifies the first control qubit for the gate.

**c2=*integer***
specifies the second control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

# %Q_CNOT Macro

adds a controlled NOT gate to the circuit.

Alias: %Q_CX

## Syntax

%Q_CNOT(*c*, *t*);

### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

# %Q_CP Macro

adds a controlled phase gate to the circuit.

### Syntax

%Q_CP(*c*, *t*, *theta*);

#### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

**theta=*float***
specifies the angle (in radians) of the phase gate to apply.

## %Q_CRX Macro

adds a controlled rotation gate for the X axis to the circuit.

### Syntax

%Q_CRX(*c*, *t*, *theta*);

#### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

**theta=*float***
specifies the angle (in radians) of the rotation gate to apply.

## %Q_CRY Macro

adds a controlled rotation gate for the Y axis to the circuit.

## Syntax

%Q_CRY(*c*, *t*, *theta*);

### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

**theta=*float***
specifies the angle (in radians) of the rotation gate to apply.

# %Q_CRZ Macro

adds a controlled rotation gate for the Z axis to the circuit.

## Syntax

%Q_CRZ(*c*, *t*, *theta*);

### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

**theta=*float***
specifies the angle (in radians) of the rotation gate to apply.

# %Q_CSWAP Macro

adds a CSWAP gate to the circuit.

Alias: %Q_FREDKIN

### Syntax

%Q_CSWAP(*c*, *t1*, *t2*);

### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t1=*integer***
specifies the first target qubit for the gate.

**t2=*integer***
specifies the second target qubit for the gate.

## %Q_CSX Macro

adds a controlled square root of X gate to the circuit.

### Syntax

%Q_CSX(*c*, *t*);

### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

## %Q_CX Macro

adds a controlled X gate to the circuit. This gate is also known as a controlled NOT (or CNOT) gate.

Alias: %Q_CNOT

## Syntax

%Q_CX(*c*, *t*);

### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

# %Q_CX_CHAIN Macro

creates a linear CX chain across the qubits in the circuit.

## Syntax

%Q_CX_CHAIN();

## Details

A linear chain connects the first qubit to the second, the second qubit to the third qubit, and so on. The last qubit is not connected (directly) to the first qubit.

# %Q_CY Macro

adds a controlled Y gate to the circuit.

## Syntax

%Q_CY(*c*, *t*);

### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

# %Q_CZ Macro

adds a controlled Z gate to the circuit.

## Syntax

%Q_CZ(*c*, *t*);

### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

# %Q_CZ_CHAIN Macro

creates a linear CZ chain across the qubits in the circuit.

## Syntax

%Q_CZ_CHAIN();

## Details

A linear chain connects the first qubit to the second, the second qubit to the third, and so on. The last qubit is not connected (directly) to the first.

# %Q_CZ_RING Macro

creates a CZ ring across the qubits in the circuit.

## Syntax

%Q_CZ_RING();

## Details

A ring connects the first qubit to the second, the second qubit to the third, and so on. The last qubit is connected directly to the first.

# %Q_ECR Macro

adds an echoed cross-resonance gate to the circuit.

## Syntax

%Q_ECR(*q1*, *q2*);

### Required Arguments

**q1=*integer***
specifies the first qubit for the gate.

**q2=*integer***
specifies the second qubit for the gate.

# %Q_FREDKIN Macro

adds a Fredkin gate to the circuit.

Alias: %Q_CSWAP

## Syntax

%Q_CSWAP(*c*, *t1*, *t2*);

### Required Arguments

**c=*integer***
specifies the control qubit for the gate.

**t1=*integer***
specifies the first target qubit for the gate.

**t2=*integer***
specifies the second target qubit for the gate.

# %Q_H Macro

adds a Hadamard gate to the circuit.

Note: $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$

## Syntax

%Q_H(*q*);

### Required Argument

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_H_ALL Macro

applies a Hadamard gate to each qubit in a circuit.

## Syntax

%Q_H_ALL();

# %Q_I Macro

adds an identity gate to the circuit.

Note: $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

## Syntax

%Q_I(*q*);

### Required Argument

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_INIT Macro

initializes the circuit data set.

## Syntax

%Q_INIT(*nq*, *nc)*;

### Required Arguments

**nq=*integer***
specifies the number of qubits in the circuit.

**nc=*integer***
specifies the number of classical bits in the circuit.

## Details

The %Q_INIT macro is used in a DATA step to initialize a circuit data set. For details about how to choose the number of classical and quantum bits, see "Qubits and Classical Bits" on page 18.

**Note:** The %Q_INIT and %Q_INIT_BATCH macros are not technically gates, but like gate macros, they are used in a DATA step to generate a circuit data set.

# %Q_INIT_BATCH Macro

initializes a batch circuit data set.

## Syntax

%Q_INIT_BATCH(*name*, *nq*, *nc)*;

### Required Arguments

**name=*character***
specifies the name of the circuit.

**nq=*integer***
specifies the number of qubits in the circuit.

**nc=*integer***
specifies the number of classical bits in the circuit.

## Details

The %Q_INIT_BATCH macro is used in a DATA step to initialize a circuit data set. Batch circuits contain more than one circuit in the data set. Use the %Q_INIT_BATCH macro to initialize each circuit in the batch. For details about creating a batch circuit, see "Building Batch Circuits" on page 24.

**Note:** The %Q_INIT and %Q_INIT_BATCH macros are not technically gates, but like gate macros, they are used in a DATA step to generate a circuit data set.

# %Q_INPUT Macro

declares a QASM 3.0 input parameter.

## Syntax

%Q_INPUT(*name*);

### Required Argument

**name=*character***
specifies the name of the parameter to add.

## Details

A parameterized circuit uses symbolic parameter names instead of numeric angles, letting you bind concrete values at execution time. The %Q_INPUT macro declares the name of the parameter so that you can use it elsewhere in the circuit.

For details about building a parameterized circuit, see "Parameterized Circuits" on page 21.

# %Q_MEASURE Macro

adds a measurement operation to the specified qubit in the circuit.

## Syntax

%Q_MEASURE(*q*,*c*);

### Required Arguments

**q=*integer***
specifies which qubit to measure.

Range 0 to N–1, where N is the number of qubits in the circuit.

**c=*integer***
specifies which classical bit the measurement should be recorded in.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_MEASURE_ALL Macro

adds a measurement operation to all qubits in the circuit.

## Syntax

%Q_MEASURE_ALL();

## Details

The %Q_MEASURE_ALL macro applies a measurement to each qubit in the circuit. Each measurement is recorded in the corresponding qubit. That is, the $n$th qubit is recorded in the $n$th classical bit.

# %Q_P Macro

adds a gate to the circuit that rotates the qubit state around the Z axis.

## Syntax

%Q_P(*q*, *theta*);

### Required Arguments

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

**theta=*float***
specifies the angle (in radians) by which to rotate the qubit state.

## Comparisons

This gate is identical to the RZ gate up to a phase factor.

# %Q_RAW Macro

injects raw QASM 3.0 code into the circuit.

## Syntax

%Q_RAW(*line*);

### Required Argument

**line=*character***
specifies the line of QASM code that you want to include in the circuit data set.

## Example

```
data my_circuit;
    %q_init(3, 3);

    %q_h(0);
    %q_raw('// Custom controlled rotation sequence');
    %q_raw('cx q[0],q[1];');
    %q_raw('rz(0.5) q[1];');
    %q_raw('cx q[0],q[1];');
```

```
    %q_cx(1, 2);

    %q_measure_all();
run;
```

# %Q_RESET Macro

resets the specified qubit to the |0> state.

## Syntax

%Q_RESET(*q*);

### Required Argument

**q=*integer***
specifies which qubit should be reset.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_RX Macro

adds a gate to the circuit that rotates the qubit state around the X axis.

## Syntax

%Q_RX(*q*, *theta*);

### Required Arguments

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

**theta=*float***
specifies the angle (in radians) by which to rotate the qubit state.

# %Q_RXX Macro

adds an Ising gate to the circuit that implements a pairwise spin-spin interaction by using the X axis on the two specified qubits.

Note: $e^{i\frac{\theta}{2}X \otimes X}$

## Syntax

%Q_XX(*q1*, *q2*, *theta*);

### Required Arguments

**q1=*integer***
specifies the first qubit for the gate.

**q2=*integer***
specifies the second qubit for the gate.

**theta=*float***
specifies the angle to use in the Ising Hamiltonian.

# %Q_RY Macro

adds a gate to the circuit that rotates the qubit state around the Y axis.

## Syntax

%Q_RY(*q*, *theta*);

### Required Arguments

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

**theta=*float***
specifies the angle (in radians) by which to rotate the qubit state.

# %Q_RYY Macro

adds an Ising gate to the circuit that implements a pairwise spin-spin interaction by using the Y axis on the two specified qubits.

Note: $e^{i\frac{\theta}{2}Y \otimes Y}$

## Syntax

%Q_YY(*q1*, *q2*, *theta*);

### Required Arguments

**q1=*integer***
specifies the first qubit for the gate.

**q2=*integer***
specifies the second qubit for the gate.

**theta=*float***
specifies the angle to use in the Ising Hamiltonian.

# %Q_RZ Macro

adds a gate to the circuit that rotates the qubit state around the Z axis.

## Syntax

%Q_RZ(*q*, *theta*);

### Required Arguments

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

**theta=*float***
specifies the angle (in radians) by which to rotate the qubit state.

# %Q_RZZ Macro

adds an Ising gate to the circuit that implements a pairwise spin-spin interaction by using the Z axis on the two specified qubits.

Note: $e^{i\frac{\theta}{2}Z \otimes Z}$

## Syntax

%Q_ZZ(*q1*, *q2*, *theta*);

### Required Arguments

**q1=*integer***
specifies the first qubit for the gate.

**q2=*integer***
specifies the second qubit for the gate.

**theta=*float***
specifies the angle to use in the Ising Hamiltonian.

# %Q_S Macro

adds an S gate to the circuit.

Note: $\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$

## Syntax

%Q_S(*q*)

### Required Argument

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_SWAP Macro

adds a swap gate to the circuit.

## Syntax

%Q_SWAP(*q1*, *q2*);

### Required Arguments

**q1=*integer***
specifies the first qubit for the gate.

**q2=*integer***
specifies the second qubit for the gate.

# %Q_SDG Macro

adds an S-dagger gate to the circuit.

Note: $\begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$

## Syntax

%Q_SDG(*q*)

### Required Argument

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_SX Macro

adds a square root of X gate to the circuit.

Note: $\frac{1}{2}\begin{pmatrix}1+i & 1-i\\ 1-i & 1+i\end{pmatrix}$

## Syntax

%Q_SX(*q*)

### Required Argument

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_T Macro

adds a T gate to the circuit.

Note: $\begin{pmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{pmatrix}$

## Syntax

%Q_T(*q*)

### Required Argument

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_TDG Macro

adds a T-dagger gate to the circuit.

Note: $\begin{pmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{pmatrix}$

## Syntax

%Q_TDG(*q*)

### Required Argument

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_TOFFOLI Macro

adds a Toffoli gate to the circuit.

Alias: %Q_CCX

## Syntax

%Q_TOFFOLI(*c1*, *c2*, *t*);

### Required Arguments

**c1=*integer***
specifies the first control qubit for the gate.

**c2=*integer***
specifies the second control qubit for the gate.

**t=*integer***
specifies the target qubit for the gate.

# %Q_U Macro

adds a general rotation gate. The 3-D rotations are specified by Euler angles.

## Syntax

%Q_U(*q*, *theta*, *phi*, *lambda*);

### Required Arguments

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

**theta=*float***
specifies the angle (in radians) by which to rotate the qubit state.

**phi=*float***
specifies the angle (in radians) by which to rotate the qubit state.

**lambda=*float***
specifies the angle (in radians) by which to rotate the qubit state.

# %Q_U1 Macro

adds a gate to the circuit that rotates the qubit state around the Z axis.

## Syntax

%Q_U1(*q*, *theta*);

### Required Arguments

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

**theta=*float***
specifies the angle (in radians) by which to rotate the qubit state.

## Comparisons

This gate is identical to the RZ gate up to a phase factor.

# %Q_U2 Macro

adds a gate to the circuit that rotates the qubit state around the Z axis and the X axis.

Note:

$$U2(\theta,\phi) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & e^{-i\phi} \\ e^{i\theta} & e^{i(\theta+\phi)} \end{pmatrix}$$

## Syntax

%Q_U2(*q*, *theta*, *phi*);

### Required Arguments

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

**theta=*float***
specifies the angle (in radians) by which to rotate the qubit state.

**phi=*float***
specifies the angle (in radians) by which to rotate the qubit state.

# %Q_U3 Macro

adds a gate to the circuit that rotates the qubit state in three dimensions.

Note:

$$U3(\theta,\phi,\lambda) = \frac{1}{\sqrt{2}}\begin{pmatrix} \cos\frac{\theta}{2} & e^{-i\phi}\sin\frac{\theta}{2} \\ e^{i\theta}\sin\frac{\theta}{2} & e^{i(\theta+\phi)}\cos\frac{\theta}{2} \end{pmatrix}$$

## Syntax

%Q_U3(*q*, *theta*, *phi*, *lambda*);

### Required Arguments

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

**theta=*float***
specifies the angle (in radians) by which to rotate the qubit state.

**phi=*float***
specifies the angle (in radians) by which to rotate the qubit state.

**lambda=*float***
specifies the angle (in radians) by which to rotate the qubit state.

## Comparisons

This gate is identical to the U gate.

# %Q_X Macro

adds a Pauli X gate to the circuit. This is also known as the NOT gate.

Note: $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

## Syntax

%Q_X(*q*)

### Required Argument

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_Y Macro

adds a Pauli Y gate to the circuit.

Note: $\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$

## Syntax

%Q_Y(*q*)

### Required Argument

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

# %Q_Z Macro

adds a Pauli Z gate to the circuit.

Note: $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$

## Syntax

%Q_Z(*q*)

### Required Argument

**q=*integer***
specifies which qubit the gate should be applied to.

Range 0 to N–1, where N is the number of qubits in the circuit.

# 7

# Quantum Execution Macros

*Using Quantum Execution Macros* 115
*Macros to Statically Inspect Quantum Circuits* 116
*%Q_COUNT_OPS Macro* 116
*%Q_DEPTH Macro* 117
*%Q_ESTIMATE_COST Macro* 118
*%Q_NUM_NONLOCAL_GATES Macro* 122
*%Q_OBSERVE Macro* 123
*%Q_OBSERVE_BATCH Macro* 126
*%Q_PLOT_BLOCH Macro* 128
*%Q_PLOT_QSPHERE Macro* 129
*%Q_PLOT_RESULTS Macro* 129
*%Q_READ_QASM Macro* 130
*%Q_SAMPLE Macro* 130
*%Q_SAMPLE_BATCH Macro* 133
*%Q_SHOW_CIRCUIT Macro* 135
*%Q_TRANSPILE Macro* 135
*%Q_WRITE_QASM Macro* 138

## Using Quantum Execution Macros

Whereas gate macros on page 86 enable you to build a quantum circuit, execution macros enable you to run a quantum circuit. Running a quantum circuit in this context can mean generating sample measurements, computing expectation values, and producing visualizations.

Unlike gate macros, execution macros are run outside a DATA step. Execution macros produce either a data set or ODS output. You can inspect any resulting data sets by using PROC PRINT or any other methods that you would usually use.

# Macros to Statically Inspect Quantum Circuits

Most quantum macros run the circuit on some backend. However, there are a few macros that simply inspect the QASM code and report on the structure. These macros never run the circuit, so they do not incur any cloud cost. Because the macros operate on QASM statically, they can be used on circuits that are too large to simulate.

These are the macros that statically report information about a circuit:

***Table 7.1*** *Circuit-Inspection Macros*

| Name | Description |
|---|---|
| %Q_COUNT_OPS (p. 116) | Returns a table that contains the type of operation and the number of times that the operation occurs in the circuit |
| %Q_DEPTH (p. 117) | Reports the length of the critical path through the circuit |
| %Q_ESTIMATE_COST | Computes an estimate of the cost to run a circuit on the basis of a specified pricing model |
| %Q_NUM_NONLOCAL_GATES (p. 122) | Reports the number of gates that act on two or more qubits |

# %Q_COUNT_OPS Macro

returns a table that contains the type of operation and the number of times that the operation occurs in the circuit.

Note: This macro does not run the circuit but inspects the QASM code statically.

## Syntax

%Q_COUNT_OPS(CIRCUIT=*circuit-data-set*, *observe-options*);

### Required Argument

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

### Optional Argument

**OUT=*output-dataset-name***
stores the table in the specified data set.

Default q_op_counts

## Details

The counts follow Qiskit conventions: measure, reset, and barrier all appear in the results that are reported by the %Q_COUNT_OPS macro. Gates that apply to a whole register rather than to indexed qubits are counted as operations but do not contribute to the depth.

# %Q_DEPTH Macro

reports the length (in gates) of the critical path through the circuit.

Note: This macro does not run the circuit but inspects the QASM code statically.

## Syntax

%Q_DEPTH(CIRCUIT=*circuit-data-set*, *observe-options*);

### Required Argument

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

### Optional Argument

**OUTVAR=*macro-variable-name***
stores the result of the computation in the specified macro variable.

Default _q_depth

## Details

The counts follow Qiskit conventions: barriers do not contribute to the number that is reported by the %Q_DEPTH macro. Gates that apply to a whole register rather than to indexed qubits are counted as operations but do not contribute to the depth.

# %Q_ESTIMATE_COST Macro

estimates cost by using a hypothetical vendor-neutral planning model.

## Syntax

%Q_ESTIMATE_COST(*estimate-options*);

### General Options

**CIRCUIT=*circuit-data-set***
specifies an existing circuit data set that is used to derive gate counts or run time.

**DETAILS=0 | 1**
specifies the level of details to write.

Default 0

0 prints a compact table.

1 prints the full pricing details.

**NSHOTS=*integer***
specifies the number of shots per task.

Default 1024

Range Greater than 0

**NTASKS=*integer***
specifies the number of workload tasks

Default 1

Range Greater than 0

**NUMBER_1Q_GATES=*integer***
specifies the number of single-qubit gates when no circuit is provided.

**NUMBER_2Q_GATES=*integer***
specifies the number of multiqubit gates when no circuit is provided.

**OUT=*data-set***
specifies the data set to write the output to.

Default cost_estimate

**PRICING_METHOD=ALL | GATE | RUNTIME | SHOT**
specifies the pricing model.

For details, see "Pricing Models" on page 120.

Default SHOT

## Gate Model Options

**PRICE_1Q_GATES=*number***
specifies an illustrative price per single-qubit gates.

Default 0.00022

**PRICE_2Q_GATES=*number***
specifies an illustrative price per multiqubit gates.

Default 0.001

## RUNTIME Model Options

**REP_DELAY=*number***
specifies the delay (in seconds) between shots. This option is used by the RUNTIME pricing model when a circuit data set is provided.

Default 0.00025

**SUBJOB_OVERHEAD=*number***
specifies the fixed overhead (in seconds) per task. This option is used for the RUNTIME pricing model when a circuit data set is provided.

Default 2

**PRICE_RUNTIME_SECOND=*number***
specifies an illustrative price per QPU second for the RUNTIME pricing model.

Default 1.6666666667

### Shot Model Options

**PRICE_SHOT=*number***
specifies an optional illustrative shot-price override.

**PRICE_TASK=*number***
specifies an illustrative task price for the shot model.

Default 0.30

## Details

### Overview of the %Q_ESTIMATE_COST Macro

The %Q_ESTIMATE_COST macro is a vendor-neutral, hypothetical planning aid. It does not retrieve live prices, inspect a billing account, submit a circuit, or produce a quote. Every price is an illustrative assumption, and actual costs might differ.

### Output of the %Q_ESTIMATE_COST Macro

The output is a row-oriented comparison data set rather than one observation in which every model is populated. The macro also uses PROC PRINT to print the rows. Common columns include pricing_method, estimate_type, pricing_model, estimate_disclaimer, ntasks, nshots, total_shots, and estimated_cost. The estimate_type value is shot_range, gate_floor, runtime, or circuit_runtime. When the PRICING_METHOD= option value is SHOT, estimated_cost equals most_expensive_cost, and price_task and cheapest_cost are also provided. When the PRICING_METHOD= option value is GATE, the gate counts and prices are included. When the PRICING_METHOD= option value is RUNTIME, the output adds price_runtime_second, rep_delay_sec, subjob_overhead_sec, runtime_label, and runtime_seconds. The circuit_runtime rows also carry circuit_duration_sec and a runtime_disclaimer. Non-applicable columns might be missing from a row. The printed table is compact by default, and the DETAILS=1 option adds the pricing model and the price and timing inputs. The macro also writes ODS disclaimer text stating that the result is a hypothetical estimate, not a quote.

### Pricing Models

Each pricing model takes a different approach to estimating costs. In each case, the total number of shots can be computed as

$$total_{shots} = N_{tasks} \times N_{shots}$$

### The GATE Model

The GATE model uses the number of gates in a circuit to estimate the cost. If you provide a circuit, it is scanned to determine the gate counts. Measurement, reset, and barrier operations are excluded from the count. If no circuit is provided, you can specify explicit counts by using the NUMBER_1Q_GATES= and

NUMBER_2Q_GATES= options. It is an error to specify a circuit and an explicit number of gates.

$$cost = \max\left(200, total_{shots} \times (N_{1q} \times PRICE_{1q} + N_{2q} \times PRICE_{2q})\right)$$

**Note:** The $200 minimum is a part of this pricing model, not a provider charge.

### The RUNTIME Model

The RUNTIME model computes the number of seconds of use to estimate the cost.

When you omit the CIRCUIT= option, the output contains reference rows for 1 second, 1 minute, 10 minutes, and 1 hour. When you specify the CIRCUIT= option, the macro also schedules the circuit's operations with generic gate durations and finds the critical-path duration d. The generic durations are a low case and a high case: 50 and 100 ns for a one-qubit gate, 300 and 600 ns for a gate on two or more qubits, and 2.8 and 5.6 microseconds for a measurement or reset. Barriers take no time. These durations are planning assumptions, not data for calibrating a device. The low and high cases give two circuit_runtime rows; each is calculated as follows:

$$runtime_{seconds} = SUBJOB_{overhead} \times NTASKS + (REP_{delay} + d) \times total_{shots})$$

Cost is then computed as follows:

$$cost = runtime_{s}econds \times PRICE_{RUNTIME_{S}ECOND}$$

**Note:** The RUNTIME model does not accept a batch circuit data set; you need to estimate each circuit separately.

### The SHOT Model

The SHOT model estimates task and shot pricing, and then it sums them to get a total price.

$$cost = N_{tasks} \times PRICE_{task} + total_{shots} \times PRICE_{shot}$$

When the PRICE_SHOT option is omitted, the shot model reports a low and high range by using $0.0005 and $0.03. When the PRICE_SHOT option is specified, only the specified value is used.

## Examples

### Example 1: Gate-Based Estimate from Explicit Counts

```
%q_estimate_cost(ntasks=10, nshots=1024, pricing_method=gates,
                 number_1q_gates=120, number_2q_gates=24);
```

### Example 2: Shot-Based Planning Estimate

```
%q_estimate_cost(ntasks=100, nshots=4096, pricing_method=shot);
```

### Example 3: Estimate from an Existing Circuit

```
%q_estimate_cost(ntasks=25, nshots=2048, pricing_method=all,
                 circuit=work.my_circuit, out=work.cost_estimate);
```

# %Q_NUM_NONLOCAL_GATES Macro

reports the number of gates that act on two or more qubits.

Note: This macro does not run the circuit but inspects the QASM code statically.

## Syntax

%Q_NUM_NONLOCAL_GATES(CIRCUIT=*circuit-data-set*, *observe-options*);

### Required Argument

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

### Optional Argument

**OUTVAR=*macro-variable-name***
stores the result of the computation in the specified macro variable.

Default _q_num_nonlocal_gates

## Details

Nonlocal gate count is the single best predictor of fidelity on real hardware, so it is worth checking before you submit a job to a provider. Compare the count before and after transpiling to see what a target device costs you.

The counts follow Qiskit conventions: the %Q_NUM_NONLOCAL_GATES macro counts multiqubit gates only, never barriers or measurements.

# %Q_OBSERVE Macro

computes expectation values of the Pauli observables. The %Q_OBSERVE macro supports exact computation and shot-based estimates.

## Syntax

%Q_OBSERVE(CIRCUIT=*circuit-data-set*, PAULIS=*list-of-observables*, *observe-options*);

### Required Arguments

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

**PAULIS=*observables-list***
specifies the list of observables to evaluate as a space-delimited list of terms. Each term specifies the Pauli observables to observe for each qubit in the circuit. For example, in a three-qubit circuit, `PAULIS=IXX` computes expectation values for the identity on qubit 2 and the X observable on the other two qubits. This option can have an optional coefficient suffix. You can specify multiple terms as an unquoted, space-delimited list, such as `PAULIS=IXX ZIZ`.

You can scale the expectation value of an observable by following the observable with a colon and the coefficient that you want to scale it by. For example, `PAULIS=ZZ:2` computes the Z expectation values on each of the two qubits and scales the result by a factor of 2. The default coefficient is 1.

### Optional Arguments

**DATA=*parameter-data-set***
specifies a data set that contains parameter values for multiple runs.

Note The data set must be a CAS table. This parameter is supported only for the PROC QRUN backend.

**DETAIL=0 | 1**
when set to 1, requests one row per Pauli term rather than a single combined result.

Default 0

**NOISE=*string***
specifies a simulator noise model.

**NTHREADS=*integer***
specifies the maximum number of threads to allocate per worker.

Default 1

Range 1 or greater

**OUT=*output-data-set***
specifies the data set to write the results to.

Note The data set must be a CAS table. This parameter is supported only for the PROC QRUN backend.

**SHOTS=*integer***
specifies the number of measurement shots to sample. By default, exact expectation values are computed.

Default None

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

## Details

The output of the %Q_OBSERVE macro depends on the value of the DETAIL= option.

When DETAIL=0, the output combines the Pauli terms into a single aggregated expectation value that represents the expected Hamiltonian directly. One row is produced for each parameter set. The schema for the output is as follows:

***Table 7.2*** *Schema of %Q_OBSERVE Output When DETAIL=0*

| Column | Type | Description |
|---|---|---|
| _pauli_ | char | Reports the combined Pauli terms that describe the full Hamiltonian. |
| _coefficient_ | char | Reports a space-separated list of coefficients for the Pauli terms. |
| _expectation_ | num | Reports the aggregated expectation value.$\langle H \rangle = \sum c_i P_i$ |
| _weighted_ | num | Reports the same value as the expectation column. This column is included for consistency with the DETAIL=1 output schema. |
| (Input columns) | mixed | Other columns that are copied when the DATA= option is used. |

When DETAIL=1, one row is still produced for each parameter set, but the output is also split so that each Pauli term appears on its own row. The schema for the output is as follows:

***Table 7.3*** *Schema of %Q_OBSERVE Output When DETAIL=1*

| Column | Type | Description |
|---|---|---|
| _pauli_ | char | Reports the Pauli term of the row |
| _coefficient_ | num | Reports the coefficient for the Pauli terms |
| _expectation_ | num | Reports expectation value of the term before scaling |
| _weighted_ | num | Reports the expectation value scaled by the coefficient |
| (Input columns) | mixed | Other columns that are copied when the DATA= option is used |

# %Q_OBSERVE_BATCH Macro

computes expectation values of the Pauli observables for a batch of circuits.

## Syntax

%Q_OBSERVE_BATCH(CIRCUIT=*circuit-data-set*, PAULIS=*list-of-observables*, *observe-options*);

### Required Arguments

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

**PAULIS=*observables-list***
specifies the list of observables to evaluate as a space-delimited list of terms. Each term specifies the Pauli observables to observe for each qubit in the circuit. For example, in a three-qubit circuit, `PAULIS=IXX` computes expectation values for the identity on qubit 2 and the X observable on the other two qubits. This option can have an optional coefficient suffix. You can specify multiple terms as an unquoted, space-delimited list, such as `PAULIS=IXX ZIZ`.

You can scale the expectation value of an observable by following the observable with a colon and the coefficient that you want to scale it by. For example, `PAULIS=ZZ:2` computes the Z expectation values on each of the two qubits and scales the result by a factor of 2. The default coefficient is 1.

### Optional Arguments

**DETAIL=0 | 1**
when set to 1, requests one row per Pauli term rather than a single combined result.

Default 0

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

**NTHREADS=*integer***
specifies the maximum number of threads to allocate per worker.

Default 1

Range 1 or greater

**OUT=*output-data-set***
specifies the data set to write the results to.

**SHOTS=*integer***
specifies the number of measurement shots to sample. By default, exact expectation values are computed.

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

# %Q_PLOT_BLOCH Macro

displays Bloch sphere representations of qubits in a circuit.

## Syntax

%Q_PLOT_BLOCH(CIRCUIT=*circuit-data-set*, *plot-bloch-options*);

### Required Argument

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

### Optional Arguments

**TITLE=*character***
specifies the title of the plot.

Default Measurement Results

**QUBITS=*list-of-integers***
specifies a space-separated list of integers that denote the qubits to depict.

Note Qubits are numbered starting from 0.

## Example

The following code fragment shows how you could plot the Bloch representations of the first and third qubit:

```
%q_plot_bloch(circuit=ghz, title=Bloch Representation, qubits=0 2);
```

# %Q_PLOT_QSPHERE Macro

displays a QSphere representation of the state vector.

## Syntax

%Q_PLOT_QSPHERE(CIRCUIT=*circuit-data-set*, *plot-qsphere-options*);

### Required Argument

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

# %Q_PLOT_RESULTS Macro

plots measurement results as a bar chart.

## Syntax

%Q_PLOT_RESULTS(DATA=*measurement-data-set*, *plot-results-options*);

### Required Argument

**DATA=*measurement-data-set***
specifies the data set to plot.

### Optional Argument

**TITLE=*character***
specifies the title of the plot.

Default Measurement Results

# %Q_READ_QASM Macro

imports a circuit data set from an OpenQASM 3.0 or 2.0 file on disk.

## Syntax

%Q_READ_QASM(FILE=*filepath*, OUT=*dataset*);

### Required Arguments

**FILE=*filepath***
specifies the file to read the QASM code from.

**OUT=*dataset***
specifies the data set to write the circuit definition to.

# %Q_SAMPLE Macro

runs a circuit and returns the sampled measurements.

Note: This macro produces an error when you pass it a circuit that has zero classical bits, because this implies that no measurement is made.

## Syntax

%Q_SAMPLE(CIRCUIT=*circuit-data-set*, qsample-options);

### Required Argument

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

### Optional Arguments

**DATA=*parameter-data-set***
specifies a data set that contains parameter values for multiple runs.

Note The data set must be a CAS table. This parameter is supported only for the PROC QRUN backend.

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

**NTHREADS=*integer***
specifies the maximum number of threads to allocate per worker.

Default 1

Range 1 or greater

**OUT=*output-data-set***
specifies the data set to write the results to.

Default quant_result

Note The data set must be a CAS table. This parameter is supported only for the PROC QRUN backend.

**SHOTS=*integer***
specifies the number of measurement shots to sample.

Default 1024

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

## Details

The resulting data set has the following schema:

***Table 7.4*** *Schema of the Sampling Results*

| Column | Type | Description |
|---|---|---|
| _measurement_ | Character | Reports the measurement observed as a bit string. For example, a measurement might be reported as 00 or 11. |
| _count_ | Number | Reports the number of times that the outcome was observed. |
| _probability_ | Number | Reports the number of times that the outcome was observed in proportion to the total shot count. |
| (Input columns) | mixed | Other columns that are copied when the DATA= option is used. |

## Example: Use %Q_SAMPLE to Sample a Quantum Circuit

This example shows how you can define a circuit to create a Bell state and then sample it.

```
data bell;            /*1*/
    %q_init(2, 2);
    %q_h(0);
    %q_cx(0, 1);
    %q_measure(0, 0);
    %q_measure(1, 1);
run;

%q_sample(circuit=bell, out=mycaslib.results, shots=1024); /*2*/
```

1 Build a circuit that defines a Bell state.

2 Draw 1,024 samples from the circuit. Store the samples in a data set called "results".

# %Q_SAMPLE_BATCH Macro

runs all circuits in a batch data set and returns the sampled measurements.

Note: This macro produces an error when you pass it a circuit that has zero classical bits, because this implies that no measurement is made.

## Syntax

%Q_SAMPLE_BATCH(CIRCUIT=*circuit-data-set*, qsample-options);

### Required Argument

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

### Optional Arguments

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

**NTHREADS=*integer***
specifies the maximum number of threads to allocate per worker.

Default 1

Range 1 or greater

**OUT=*output-data-set***
specifies the data set to write the results to.

Default quant_result

Note The data set must be a CAS table. This parameter is supported only for the PROC QRUN backend.

**SHOTS=*integer***
specifies the number of measurement shots to sample.

Default 1024

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

# %Q_SHOW_CIRCUIT Macro

displays the circuit diagram.

## Syntax

%Q_SHOW_CIRCUIT(CIRCUIT=*circuit-data-set*);

### Required Argument

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

## Details

Output from this macro is sent through ODS. Use `ods listing gpath=` and `ods graphics /` to control the file output.

# %Q_TRANSPILE Macro

transpiles a circuit to a target quantum device.

# Syntax

%Q_TRANSPILE(CIRCUIT=*circuit-data-set*, DEVICE=*quantum-device*, *transpile-options*);

## Required Arguments

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

**DEVICE=*quantum-device***
specifies the quantum device to transpile the circuit to.

## Optional Arguments

**GATESOUT=*gateout-data-set***
specifies the data set to write the gate counts to.

**METAOUT=*metaout-data-set***
specifies the data set to write the transpilation summary to.

**OPTIMIZE=*integer***
specifies the level of optimization to perform while transpiling the circuit.

Default 1

Range 0-3

- 0 Checks connectivity only. This value is not suitable for hardware runs.
- 1 Suited for most production use cases.
- 2 Suited for experiments that require higher fidelity than normal.
- 3 Applies the most compression. This level uses stochastic heuristics, so repeated runs can produce different results unless you also specify a seed.

**OUT=*data-set***
specifies the data set to write to.

**SEED=*integer***
specifies the seed to use to make transpilation reproducible. By default, a pseudorandom number seed is used.

# Details

The %Q_TRANSPILE macro writes transpilation summary info, per-gate counts, and the transpiled circuit to separate output data sets.

*Table 7.5* *Schema for the Transpiled Circuit Results*

| Name | Type | Description |
|---|---|---|
| _qasm_ | char | Contains one row per line of the transpiled QASM |

*Table 7.6* *Schema for the Gate Count Results*

| Name | Type | Description |
|---|---|---|
| _gate_ | char | Reports the type of gate in the circuit |
| _count_ | num | Reports the number of each gate in the transpiled circuit |

*Table 7.7* *Schema for the Meta Results*

| Name | Type | Description |
|---|---|---|
| _device_ | char | Reports the device that the circuit was transpiled for |
| _optimization_level_ | num | Reports the level of optimization that is used for the circuit |
| _depth_ | num, | Reports the depth (in gates) of the transpiled circuit |
| _num_qubits_ | num | Reports the number of qubits in the transpiled circuit |
| _original_depth_ | num | Reports the depth (in gates) of the original circuit |
| _transpilation_time_ | num | Reports the time (in seconds) that is needed to transpile the circuit |
| _layout_ | char | Reports the layout format that is used by the layout engine |
| _swap_count_ | num | Specifies the number of swap gates that are used for the transpiled circuit |
| _basis_gates_ | char | Reports the basis gates that are used for the transpiled circuit |

# %Q_WRITE_QASM Macro

exports a circuit data set to an OpenQASM 3.0 file on disk.

## Syntax

%Q_WRITE_QASM(CIRCUIT=*circuit-data-set*, FILE=*filepath*);

### Required Arguments

**CIRCUIT=*circuit-data-set***
specifies the data set that contains the circuit definition.

**FILE=*filepath***
specifies the file to write the QASM code to.

# 8

# Quantum Machine Learning Macros

*Quantum Machine Learning Concepts* ... 139
Quantum Feature Transformation ... 139
Quantum Feature Encoding Strategies ... 139
Pauli Feature Extraction ... 139
*%QML_COMPARE Macro* ... 141
*%QML_GRADBOOST Macro* ... 145
*%QML_LOGSELECT Macro* ... 149
*%QML_SVMACHINE Macro* ... 152
*%QML_SWEEP Macro* ... 156
*%QML_TRANSFORM Macro* ... 159
*Example: End-to-End QML Workflow Using Synthetic Data* ... 164
End-to-End QML Workflow Using Synthetic Data ... 164

## Quantum Machine Learning Concepts

### Quantum Feature Transformation

The core capability of the quantum machine learning (QML) macros is quantum feature transformation. Quantum feature transformation encodes classical data into quantum circuits and extracts expectation values as new feature columns. This is analogous to kernel methods or principal component analysis (PCA), but it uses quantum mechanical phenomena to create feature spaces that are richer than classical feature maps.

The general workflow is as follows:

1. Preprocess input features. This step is optional.
2. Encode feature values into a parameterized quantum circuit.
3. Entangle qubits to create correlations between the encoded features.
4. Measure Pauli expectation values as quantum features.
5. Append the quantum features to the original data set.

# Quantum Feature Encoding Strategies

Encoding determines how classical features are put into a quantum circuit. The method of encoding determines the type of interactions and therefore determines the type of quantum features you can generate.

Specifying the ENCODING=ANGLE option in one of the QML macros uploads the data once as the first step of the circuit. The circuit has a number of encoding slots equal to the number of qubits times the number of encoding gates. Each slot corresponds to a single data-dependent rotation. Whenever there are fewer predictors than slots, predictors cycle over the slots in the order in which the VARS= option specifies them. Whenever there are more predictors than slots, the extra predictors do not affect the quantum features. You can pass the extra predictors to the classical model by specifying the KEEPORIGINAL=1 option. Each layer after the upload applies seeded random rotations from the ROTGATES= option, the entangling gates from the ENTGATES= option over the chosen topology, and a second set of seeded random rotations. The random angles come from the SEED= option and are the same for every observation, and the NLAYERS= option repeats this block.

The ENCODING=IQP option uses the instantaneous quantum polynomial circuit structure, which creates second-order feature interactions (products of pairs of feature values) directly in the circuit. This is particularly effective for classification tasks with nonlinear decision boundaries.

The ENCODING=DENSE option packs multiple features per qubit by applying several rotations per qubit. This is useful when the number of available qubits is limited but you want to encode many features.

# Pauli Feature Extraction

After the circuit is constructed, expected values are computed for a set of Pauli operators. Larger observable sets extract more features from the same circuit, capturing richer quantum correlations at the cost of more computation. Most users are advised to specify the OBSERVABLES=AUTO option in the QML macros and let the macro choose the best set for the encoding strategy.

The following table describes the observable sets in increasing number of features:

***Table 8.1*** *Pauli Observable Sets*

| Value | Number of Features Generated | Description |
|---|---|---|
| ALL_ZZ | $N_q + \binom{N_q}{2}$ | Measures the polarization of each qubit and all pairwise correlations of qubits. |
| FULL | $N_q + 3\binom{N_q}{2}$ | Measures all single-qubit and pairwise correlations. |
| QRC | Up to $3N_q - 1$ | Measures polarization of each qubit and the correlation of nearest neighbors in multiple bases. |
| Z | $N_q$ | Measures polarization of each qubit. |
| ZZ | Up to $2N_q$ | Measures polarization of each qubit and the correlation of nearest neighbors. (Duplicate Pauli strings are removed.) |

$N_q$ denotes the number of qubits to use.

# %QML_COMPARE Macro

compares a classical machine learning model to a quantum machine learning model.

## Syntax

%QML_COMPARE(DATA=*data-set*, TARGET=*variable-name*, *qml-compare-options*);

### Required Arguments

**DATA=*data-set***
specifies the input data set for the comparison.

**TARGET=*variable-name***
specifies the target variable of the comparison.

## Compare Options

**MODEL=GRADBOOST | LOGSELECT | SVMACHINE**
specifies the model type to try.

Default LOGSELECT

**OUT=*data-set***
specifies the name of a data set to write the output to.

Default qml_compare_out

**TESTDATA=*data-set***
specifies the testing data set.

## Quantum Transform Arguments

These arguments influence the behavior of the quantum features that are created by using the %QML_TRANSFORM macro:

**ENCODING=ANGLE | DENSE | IQP**
specifies the encoding scheme.

Default ANGLE

See "Quantum Feature Encoding Strategies" on page 140

You can specify the following values:

ANGLE maps each feature to a data-dependent rotation. Generated circuits use the selected layer, topology, and gate settings. This is a good general-purpose encoding.

DENSE uses multiple rotations per qubit per feature, packing more information into each qubit. This encoding is best for situations in which few qubits are available.

IQP uses the encoding of the instantaneous quantum polynomial (IQP) circuit, which has pairwise feature interactions and RZZ entanglement. This encoding is best for classification tasks that involve nonlinear boundaries.

**NLAYERS=*integer***
specifies the number of encoding layers to use.

Default 2

**NOISE=*string***
specifies a simulator noise model.

**NQUBITS=*integer***
specifies the number of qubits to use.

Default 8

**NTHREADS=*integer***
specifies the maximum number of threads to use.

**OBSERVABLES=AUTO | ALL_ZZ | CUSTOM | FULL | QRC | Z | ZZ**
controls which quantum measurements to take after the circuit runs. Each set is a curated collection of Pauli operators that captures a different level of quantum correlation. The specified values pertain to the level of detail rather than the specific observables.

Default AUTO

See "Pauli Feature Extraction" on page 140

You can specify the following observable sets:

AUTO selects observables by using the encoding strategy.

ALL_ZZ selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are classical only.

CUSTOM selects observables by using the Pauli string that you specify in the PAULIS= option.

FULL selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are both classical and quantum.

QRC selects observables that consist of per-qubit polarization, ring-neighbor ZZ correlations, and adjacent-pair XX correlations.

Z selects observables that consist of per-qubit polarization polarization only.

ZZ selects observables that consist of per-qubit polarization and ring-neighbor ZZ correlations. Duplicate Pauli strings are removed.

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

**TOPOLOGY=CUSTOM | FULL | LINEAR | NONE | RING | STAR**
specifies the entanglement topology.

Default RING

You can specify the following topologies:

CUSTOM enables you to specify entanglement as edges in a graph that is supplied to the ENTMAP= option.

FULL entangles every qubit with every other qubit.

LINEAR entangles qubits as a chain without entangling the last qubit with the first qubit.

NONE uses no entanglement.

RING entangles qubits as a chain in which the last qubit wraps around to entangle the first.

STAR entangles every qubit with qubit 0, treating qubit 0 as the central qubit.

## Details

The following schema describes the output table of the %QML_COMPARE macro:

***Table 8.2*** *Schema of the %QML_COMPARE Results*

| Column | Description |
|---|---|
| _model_ | Reports the type of model used. Examples include GRADBOOST, LOGSELECT, and SVMACHINE. |
| _type_ | Reports the type of features that are used. Examples include quantum or classical. |
| _train_accuracy_ | Reports the training accuracy as a percentage. |
| _test_accuracy_ | Reports the test accuracy as a percentage. |
| _nqubits_ | Reports the number of qubits that are used. |
| _encoding_ | Reports the encoding scheme that is used. |

| Column | Description |
|---|---|
| _topology_ | Reports the entanglement topology of the qubits. |

# %QML_GRADBOOST Macro

performs gradient boosting by using quantum features.

## Syntax

%QML_GRADBOOST(DATA=*data-set*, TARGET=*variable-name*, *qml-gradboost-options);*

### Required Arguments

**DATA=*data-set***
specifies the input data set for the gradient boosting.

**TARGET=*variable-name***
specifies the target variable of the gradient boosting.

### Gradient Boosting Arguments

**LEARNINGRATE=*number***
specifies the shrinkage rate.

Default 0.1

Range [0, 1]

**MAXDEPTH=*integer***
specifies the maximum tree depth. The number of levels in a tree is equal to the depth plus 1.

Default 5

Range 0 or greater

**NTREES=*integer***
specifies the number of boosting iterations.

Default 100

Range 0 or greater

**OUT=*data-set***
specifies the name of a data set to write the output to.

Default qml_gradboost_out

**OUTMODEL=*output-model-store***
specifies the name of the analytic store to save the model to.

Default _qmlcas.qml_gradboost_model

**TESTDATA=*data-set***
specifies the testing data set.

## Quantum Transform Arguments

These arguments influence the behavior of the quantum features that are created by using the %QML_TRANSFORM macro:

**ENCODING=ANGLE | DENSE | IQP**
specifies the encoding scheme.

Default ANGLE

See "Quantum Feature Encoding Strategies" on page 140

You can specify the following values:

ANGLE maps each feature to a data-dependent rotation. Generated circuits use the selected layer, topology, and gate settings. This is a good general-purpose encoding.

DENSE uses multiple rotations per qubit per feature, packing more information into each qubit. This encoding is best for situations in which few qubits are available.

IQP uses the encoding of the instantaneous quantum polynomial (IQP) circuit, which has pairwise feature interactions and RZZ entanglement. This encoding is best for classification tasks that involve nonlinear boundaries.

**KEEPORIGINAL=0 | 1**
specifies whether to keep or drop the predictor columns. When KEEPORIGINAL=0, the predictor columns are dropped.

Default 1

**NLAYERS=*integer***
specifies the number of encoding layers to use.

Default 2

**NOISE=*string***
specifies a simulator noise model.

**NQUBITS=*integer***
specifies the number of qubits to use.

Default 8

**NTHREADS=*integer***
specifies the maximum number of threads to use.

**OBSERVABLES=AUTO | ALL_ZZ | CUSTOM | FULL | QRC | Z | ZZ**
controls which quantum measurements to take after the circuit runs. Each set is a curated collection of Pauli operators that captures a different level of quantum correlation. The specified values pertain to the level of detail rather than the specific observables.

Default AUTO

See "Pauli Feature Extraction" on page 140

You can specify the following observable sets:

| | |
|---|---|
| AUTO | selects observables by using the encoding strategy. |
| ALL_ZZ | selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are classical only. |
| CUSTOM | selects observables by using the Pauli string that you specify in the PAULIS= option. |
| FULL | selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are both classical and quantum. |
| QRC | selects observables that consist of per-qubit polarization, ring-neighbor ZZ correlations, and adjacent-pair XX correlations. |
| Z | selects observables that consist of per-qubit polarization polarization only. |
| ZZ | selects observables that consist of per-qubit polarization and ring-neighbor ZZ correlations. Duplicate Pauli strings are removed. |

**SHOWCIRCUIT=0 | 1**
specifies whether to print the circuit to the log. When SHOWCIRCUIT=1, the circuit is printed.

Default 0

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

**PREPROCESSING=NONE | NORMALIZE | STANDARDIZE**
specifies the preprocessing mode. PREPROCESSING=STANDARDIZE standardizes the input feature vectors before they are encoded.

Default NORMALIZE

NONE performs no scaling.

NORMALIZE scales predictors to fit in the Bloch sphere.

STANDARDIZE scales predictors so that the mean of the feature components is zero and the standard deviation is zero.

**TOPOLOGY=CUSTOM | FULL | LINEAR | NONE | RING | STAR**
specifies the entanglement topology.

Default RING

You can specify the following topologies:

CUSTOM enables you to specify entanglement as edges in a graph that is supplied to the ENTMAP= option.

FULL entangles every qubit with every other qubit.

LINEAR entangles qubits as a chain without entangling the last qubit with the first qubit.

NONE uses no entanglement.

RING entangles qubits as a chain in which the last qubit wraps around to entangle the first.

STAR entangles every qubit with qubit 0, treating qubit 0 as the central qubit.

# %QML_LOGSELECT Macro

performs logistic regression and feature selection by using quantum features.

## Syntax

%QML_LOGSELECT(DATA=*data-set*, TARGET=*variable-name*, *logselect-options*);

### Required Arguments

**DATA=*data-set***
specifies the input data set for the logistic regression.

**TARGET=*variable-name***
specifies the target variable of the logistic regression.

### Logistic Regression Options

These arguments influence the behavior of the logistic regression algorithm:

**OUTMODEL=*output-model-store***
specifies the name of the analytic store to save the model to.

Default _qmlcas.qml_logselect_model

**SELECTION=NONE | LASSO | ELASTICNET**
specifies the regularization scheme to use for fitting the model.

**TESTDATA=*data-set***
specifies the testing data set.

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

## Quantum Transform Arguments

These arguments influence the behavior of the quantum features that are created by using the %QML_TRANSFORM macro:

**ENCODING=ANGLE | DENSE | IQP**
specifies the encoding scheme.

Default ANGLE

See "Quantum Feature Encoding Strategies" on page 140

You can specify the following values:

ANGLE maps each feature to a data-dependent rotation. Generated circuits use the selected layer, topology, and gate settings. This is a good general-purpose encoding.

DENSE uses multiple rotations per qubit per feature, packing more information into each qubit. This encoding is best for situations in which few qubits are available.

IQP uses the encoding of the instantaneous quantum polynomial (IQP) circuit, which has pairwise feature interactions and RZZ entanglement. This encoding is best for classification tasks that involve nonlinear boundaries.

**KEEPORIGINAL=0 | 1**
specifies whether to keep or drop the predictor columns. When KEEPORIGINAL=0, the predictor columns are dropped.

Default 1

**MAXITER=*integer***
specifies the maximum number of iterations in any LOGSELECT procedure optimization.

Default 50

**NQUBITS=*integer***
specifies the number of qubits to use.

Default 8

**NLAYERS=*integer***
specifies the number of encoding layers to use.

Default 2

**NOISE=*string***
specifies a simulator noise model.

**NTHREADS=*integer***
specifies the maximum number of threads to use.

**OBSERVABLES=AUTO | ALL_ZZ | CUSTOM | FULL | QRC | Z | ZZ**
controls which quantum measurements to take after the circuit runs. Each set is a curated collection of Pauli operators that captures a different level of quantum correlation. The specified values pertain to the level of detail rather than the specific observables.

Default AUTO

See "Pauli Feature Extraction" on page 140

You can specify the following observable sets:

| | |
|---|---|
| AUTO | selects observables by using the encoding strategy. |
| ALL_ZZ | selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are classical only. |
| CUSTOM | selects observables by using the Pauli string that you specify in the PAULIS= option. |
| FULL | selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are both classical and quantum. |
| QRC | selects observables that consist of per-qubit polarization, ring-neighbor ZZ correlations, and adjacent-pair XX correlations. |
| Z | selects observables that consist of per-qubit polarization polarization only. |
| ZZ | selects observables that consist of per-qubit polarization and ring-neighbor ZZ correlations. Duplicate Pauli strings are removed. |

**OUT=*data-set***
specifies the name of a data set to write the output to.

Default qml_logselect_out

**PREPROCESSING=NONE | NORMALIZE | STANDARDIZE**
specifies the preprocessing mode. PREPROCESSING=STANDARDIZE standardizes the input feature vectors before they are encoded.

Default NORMALIZE

| | |
|---|---|
| NONE | performs no scaling. |

NORMALIZE scales predictors to fit in the Bloch sphere.

STANDARDIZE scales predictors so that the mean of the feature components is zero and the standard deviation is zero.

**SHOWCIRCUIT=0 | 1**
specifies whether or not to print the circuit to the log. When SHOWCIRCUIT=1, the circuit is printed.

Default 0

**TOPOLOGY=CUSTOM | FULL | LINEAR | NONE | RING | STAR**
specifies the entanglement topology.

Default RING

You can specify the following topologies:

CUSTOM enables you to specify entanglement as edges in a graph that is supplied to the ENTMAP= option.

FULL entangles every qubit with every other qubit.

LINEAR entangles qubits as a chain without entangling the last qubit with the first qubit.

NONE uses no entanglement.

RING entangles qubits as a chain in which the last qubit wraps around to entangle the first.

STAR entangles every qubit with qubit 0, treating qubit 0 as the central qubit.

# %QML_SVMACHINE Macro

performs support vector machine by using quantum features.

## Syntax

%QML_SVMACHINE(DATA=*data-set*, TARGET=*variable-name*, *qml-svmachine-options);*

### Required Arguments

**DATA=*data-set***
specifies the input data set for the support vector machine.

**TARGET=*variable-name***
specifies the target variable of the support vector machine.

## Support Vector Machine Arguments

**C=*number***
specifies the regularization parameter.

Default 1

Range 0 or greater

**KERNEL=*kernel-statement***
specifies the kernel to use during training. This parameter is passed directly to the KERNEL statement of the SVMACHINE procedure. You can supply additional kernel parameters following a / (forward slash) delimiter. For example, `kernel=polynomial / degree=3` specifies that a third-degree polynomial should be used for the kernel.

Default LINEAR

**OUT=*data-set***
specifies the name of a data set to write the output to.

Default qml_svmachine_out

**OUTMODEL=*output-model-store***
specifies the name of the analytic store to save the model to.

Default _qmlcas.qml_svmachine_model

**TESTDATA=*data-set***
specifies the testing data set.

## Quantum Transform Arguments

These arguments influence the behavior of the quantum features that are created by using the %QML_TRANSFORM macro:

**ENCODING=ANGLE | DENSE | IQP**
specifies the encoding scheme.

Default ANGLE

See "Quantum Feature Encoding Strategies" on page 140

You can specify the following values:

ANGLE maps each feature to a data-dependent rotation. Generated circuits use the selected layer, topology, and gate settings. This is a good general-purpose encoding.

DENSE uses multiple rotations per qubit per feature, packing more information into each qubit. This encoding is best for situations in which few qubits are available.

IQP uses the encoding of the instantaneous quantum polynomial (IQP) circuit, which has pairwise feature interactions and RZZ entanglement. This encoding is best for classification tasks that involve nonlinear boundaries.

**KEEPORIGINAL=0 | 1**
specifies whether to keep or drop the predictor columns. When KEEPORIGINAL=0, the predictor columns are dropped.

Default 1

**NLAYERS=*integer***
specifies the number of encoding layers to use.

Default 2

**NOISE=*string***
specifies a simulator noise model.

**NQUBITS=*integer***
specifies the number of qubits to use.

Default 8

**NTHREADS=*integer***
specifies the maximum number of threads to use.

**OBSERVABLES=AUTO | ALL_ZZ | CUSTOM | FULL | QRC | Z | ZZ**
controls which quantum measurements to take after the circuit runs. Each set is a curated collection of Pauli operators that captures a different level of quantum correlation. The specified values pertain to the level of detail rather than the specific observables.

Default AUTO

See "Pauli Feature Extraction" on page 140

You can specify the following observable sets:

AUTO selects observables by using the encoding strategy.

ALL_ZZ selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are classical only.

CUSTOM selects observables by using the Pauli string that you specify in the PAULIS= option.

FULL selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are both classical and quantum.

QRC selects observables that consist of per-qubit polarization, ring-neighbor ZZ correlations, and adjacent-pair XX correlations.

Z selects observables that consist of per-qubit polarization polarization only.

ZZ selects observables that consist of per-qubit polarization and ring-neighbor ZZ correlations. Duplicate Pauli strings are removed.

**SHOWCIRCUIT=0 | 1**
specifies whether or not to print the circuit to the log. When SHOWCIRCUIT=1, the circuit is printed.

Default 0

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

**PREPROCESSING=NONE | NORMALIZE | STANDARDIZE**
specifies the preprocessing mode. PREPROCESSING=STANDARDIZE standardizes the input feature vectors before they are encoded.

Default NORMALIZE

NONE performs no scaling.

NORMALIZE scales predictors to fit in the Bloch sphere.

STANDARDIZE scales predictors so that the mean of the feature components is zero and the standard deviation is zero.

**TOPOLOGY=CUSTOM | FULL | LINEAR | NONE | RING | STAR**
specifies the entanglement topology.

Default RING

You can specify the following topologies:

| | |
|---|---|
| CUSTOM | enables you to specify entanglement as edges in a graph that is supplied to the ENTMAP= option. |
| FULL | entangles every qubit with every other qubit. |
| LINEAR | entangles qubits as a chain without entangling the last qubit with the first qubit. |
| NONE | uses no entanglement. |
| RING | entangles qubits as a chain in which the last qubit wraps around to entangle the first. |
| STAR | entangles every qubit with qubit 0, treating qubit 0 as the central qubit. |

# %QML_SWEEP Macro

performs a quantum hyperparameter sweep.

## Syntax

%QML_SWEEP(DATA=*data-set*, TARGET=*variable-name*);

### Required Arguments

**DATA=*data-set***
specifies the input data set for the model.

**TARGET=*variable-name***
specifies the target variable of the model.

### Sweep Arguments

**BEST=*data-set***
specifies the output data set to write the top-ranked hyperparameters to.

**METRIC=ACCURACY | AUC | MISCLASS**
specifies the metric to use to evaluate models.

**MODEL=GRADBOOST | LOGSELECT | SVMACHINE**
specifies the model type to try.

Default LOGSELECT

**OUT=*data-set***
specifies the data set to write the ranked results to.

**TESTDATA=*data-set***
specifies the test data set.

## Quantum Transform Arguments

These arguments influence the behavior of the quantum features that are created by using the %QML_TRANSFORM macro:

**ENCODING=ANGLE | DENSE | IQP**
specifies a space-separated list of encoding schemes to try.

Default ANGLE IQP

See "Quantum Feature Encoding Strategies" on page 140

**NLAYERS=*integer***
specifies a space-separated list of number of encoding layers to use.

Default 1 2 3

**NQUBITS=*integer***
specifies a space-separated list of number of qubits to try.

Default 4 6 8

**OBSERVABLES=AUTO | ALL_ZZ | CUSTOM | FULL | QRC | Z | ZZ**
controls which quantum measurements to take after the circuit runs. Each set is a curated collection of Pauli operators that captures a different level of quantum correlation. The specified values pertain to the level of detail rather than the specific observables.

Default AUTO

See "Pauli Feature Extraction" on page 140

You can specify the following observable sets:

| | |
|---|---|
| AUTO | selects observables by using the encoding strategy. |
| ALL_ZZ | selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are classical only. |
| CUSTOM | selects observables by using the Pauli string that you specify in the PAULIS= option. |
| FULL | selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are both classical and quantum. |
| QRC | selects observables that consist of per-qubit polarization, ring-neighbor ZZ correlations, and adjacent-pair XX correlations. |
| Z | selects observables that consist of per-qubit polarization polarization only. |

ZZ selects observables that consist of per-qubit polarization and ring-neighbor ZZ correlations. Duplicate Pauli strings are removed.

**TOPOLOGY=CUSTOM | FULL | LINEAR | NONE | RING | STAR**
specifies a space-separated list of entanglement topologies to try.

Default RING

You can specify the following topologies:

CUSTOM enables you to specify entanglement as edges in a graph that is supplied to the ENTMAP= option.

FULL entangles every qubit with every other qubit.

LINEAR entangles qubits as a chain without entangling the last qubit with the first qubit.

NONE uses no entanglement.

RING entangles qubits as a chain in which the last qubit wraps around to entangle the first.

STAR entangles every qubit with qubit 0, treating qubit 0 as the central qubit.

## Details

The %QML_SWEEP macro searches over quantum hyperparameters to find the best configuration for a given data set. It trains a model for every combination of the specified parameter values, evaluates each value, and produces a table of ranked comparisons.

The %QML_SWEEP macro performs exhaustive grid search over all combinations. For large search spaces, use PROC HYPERTUNE instead. PROC HYPERTUNE supports Bayesian, random, and evolutionary search strategies that find good configurations with far fewer evaluations.

The OUT= data set contains one row per configuration. Rows are ordered by the _test_metric_ column first, then by the _train_metric_ column. The following schema describes the columns of the OUT= table:

***Table 8.3*** *Schema of the %QML_SWEEP Output Table*

| Column | Description |
|---|---|
| _nqubits_ | Reports the number of qubits that are used |
| _nlayers_ | Reports the number of layers that are used |
| _encoding_ | Reports the encoding scheme that are used |

| Column | Description |
|---|---|
| _topology_ | Reports the topology that are used |
| _train_metric_ | Reports the result of the metric that is used on the training data set |
| _test_metric_ | Reports the result of the metric that is used on the test data set |
| _nfeatures_ | Reports the number of features that are used |
| _rank_ | Reports the ranking of the model compared to others in the data set |

# %QML_TRANSFORM Macro

transforms quantum features.

## Syntax

%QML_TRANSFORM(DATA=*data-set*, *transform-options*);

### Required Arguments

**DATA=*data-set***
specifies the input data set.

**TARGET=*character***
specifies the target variable to exclude from the transform.

Interaction Any variables that you do not specify in the TARGET= or VARS= option are omitted from the analysis.

### Optional Arguments

**CONTROLLEDROTTHETA=*list-of-angles***
specifies a space-separated list of angles (in radians) to use for rotational gates. Examples include `CRX=1.57 CRY=0.78 CRZ=0.39`.

**ENCGATES=*list-of-gates***
specifies a space-separated list of gates to use for data encoding.

Default RY

**ENCODING=ANGLE | DENSE | IQP**
specifies the encoding scheme.

Default ANGLE

See "Quantum Feature Encoding Strategies" on page 140

You can specify the following values:

ANGLE maps each feature to a data-dependent rotation. Generated circuits use the selected layer, topology, and gate settings. This is a good general-purpose encoding.

DENSE uses multiple rotations per qubit per feature, packing more information into each qubit. This encoding is best for situations in which few qubits are available.

IQP uses the encoding of the instantaneous quantum polynomial (IQP) circuit, which has pairwise feature interactions and RZZ entanglement. This encoding is best for classification tasks that involve nonlinear boundaries.

**ENTGATES=*list-of-gates***
specifies a space-separated list of entangling gates that are used in later layers. Entangling gates include CX, CY, CZ, CRX, CRY, and CRZ.

Default CZ

**ENTMAP=*list-of-edges***
specifies a list of edges in a graph that defines entanglement. Edges are expressed as an ordered pair from one qubit to the next. For example, `[(0,1),(1,2)]` entangles qubit 1 with qubit 0 and qubit 2 with qubit 1.

Interaction This option is used only when TOPOLOGY=CUSTOM.

**KEEPORIGINAL=0 | 1**
specifies whether to keep or drop the predictor columns. When KEEPORIGINAL=0, the predictor columns are dropped.

Default 1

**NTHREADS=*integer***
specifies the maximum number of threads to use.

**NQUBITS=*integer***
specifies the number of qubits.

Interaction The default value of the NQUBITS= option depends on the encoding scheme that you use.

**NLAYERS=**
specifies the total number of encoding and entanglement layers.

Default 2

**NOISE=*string***
specifies a simulator noise model.

**OBSERVABLES=AUTO | ALL_ZZ | CUSTOM | FULL | QRC | Z | ZZ**
controls which quantum measurements to take after the circuit runs. Each set is a curated collection of Pauli operators that captures a different level of quantum correlation. The specified values pertain to the level of detail rather than the specific observables.

Default AUTO

See "Pauli Feature Extraction" on page 140

You can specify the following observable sets:

| | |
|---|---|
| AUTO | selects observables by using the encoding strategy. |
| ALL_ZZ | selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are classical only. |
| CUSTOM | selects observables by using the Pauli string that you specify in the PAULIS= option. |
| FULL | selects observables that consist of per-qubit polarization and all pairwise ZZ correlations. Pairwise ZZ correlations are both classical and quantum. |
| QRC | selects observables that consist of per-qubit polarization, ring-neighbor ZZ correlations, and adjacent-pair XX correlations. |
| Z | selects observables that consist of per-qubit polarization polarization only. |
| ZZ | selects observables that consist of per-qubit polarization and ring-neighbor ZZ correlations. Duplicate Pauli strings are removed. |

**OUT=*data-set***
specifies the name of the output data set.

Default qml_transform_out

**PAULIS=*pauli-string***
specifies custom Pauli strings to use for custom observables.

Interaction This option is used only when OBSERVABLES=CUSTOM.

**PREBUILTCIRCUIT=*qasm-file***
specifies the OpenQASM circuit file to use instead of generating one.

**PREFIX=*character***
specifies the prefix to use in generated column names.

Default qf_

**PREPROCESSING=NONE | NORMALIZE | STANDARDIZE**
specifies the preprocessing mode. PREPROCESSING=STANDARDIZE standardizes the input feature vectors before they are encoded.

Default NORMALIZE

NONE performs no scaling.

NORMALIZE scales predictors to fit in the Bloch sphere.

STANDARDIZE scales predictors so that the mean of the feature components is zero and the standard deviation is zero.

**ROTGATES=*list-of-gates***
specifies a space-separated list of single-qubit rotational gates that are used in later layers.

Default RX RY RZ

**SCALE=*number***
specifies a global scaling factor that is applied before encoding.

**SEED=*integers***
specifies the seed to use for reproducible generated rotational angles.

**SHOWCIRCUIT=0 | 1**
specifies whether or not to print the circuit to the log. When SHOWCIRCUIT=1, the circuit is printed.

Default 0

**SIMMETHOD=AUTO | DENSITYMATRIX | MPS | STABILIZER | STATEVECTOR**
specifies the representation of the local simulation.

Default AUTO

Note The SIMMETHOD= option applies only when the local provider is used. Cloud providers select their own simulation internally.

AUTO allows the simulator to choose the appropriate simulation method for the specified circuit.

DENSITYMATRIX uses a density matrix representation, which is best for a noisy simulation with midcircuit measurement. It uses approximately half the qubits of the representation that you specify by using the STATEVECTOR value.

MPS uses a matrix-product-state (MPS) tensor representation. MPS is best for circuits with a low qubit entanglement ratio or when the statevector method is infeasible because of high memory usage.

STABILIZER uses a Clifford stabilizer representation, which is best for large Clifford-only circuits.

STATEVECTOR uses a dense state vector, which is the best general-purpose exact simulation with a small to medium number of qubits.

**TOPOLOGY=CUSTOM | FULL | LINEAR | NONE | RING | STAR**
specifies the entanglement topology.

Default RING

You can specify the following topologies:

CUSTOM enables you to specify entanglement as edges in a graph that is supplied to the ENTMAP= option.

FULL entangles every qubit with every other qubit.

LINEAR entangles qubits as a chain without entangling the last qubit with the first qubit.

NONE uses no entanglement.

RING entangles qubits as a chain in which the last qubit wraps around to entangle the first.

STAR entangles every qubit with qubit 0, treating qubit 0 as the central qubit.

**VARS=*list-of-columns***
specifies a space-separated list of predictor columns to encode.

Interaction When you omit the VARS= option, all numeric columns are used except for columns that you specify in the TARGET= option.

## Details

The %QML_TRANSFORM macro encodes numeric predictors into a parameterized quantum circuit, computes Pauli expectation values, and appends them as new feature columns. This method is analogous to principal component analysis (PCA) or kernel-based feature expansion, but it uses quantum circuits to generate nonlinear features in a higher-dimensional Hilbert space.

# Example: End-to-End QML Workflow Using Synthetic Data

## Example 1: End-to-End QML Workflow Using Synthetic Data

The following example shows how you can use the quantum machine learning macros to perform an end-to-end quantum machine learning task.

```
data mydata;                          /* 1 */
    call streaminit(42);
    do i = 1 to 200;
        x1 = rand('normal');
        x2 = rand('normal');
        y = (x1**2 + x2**2 > 1);
        output;
    end;
    drop i;
run;

data train test;                     /* 2 */
    set mydata;
    if rand('uniform') < 0.7 then output train;
    else output test;
run;

%qml_compare(data=train, target=y, testdata=test,
model=LOGSELECT); /* 3 */

%qml_sweep(data=train, target=y,
testdata=test,                        /* 4 */
           model=LOGSELECT,
           nqubits=2,
           encoding=ANGLE IQP,
           topology=RING,
           out=sweep_results);

%qml_logselect(data=train, target=y,
testdata=test,                   /* 5 */
              nqubits=6, encoding=IQP, topology=FULL);
```

1 Use a DATA step to create a synthetic data set.

2 Use a DATA step to split the data into a training data set and a testing data set.

3 Use %QML_COMPARE to quickly determine whether or not using quantum features increases the effectiveness of the model.

4 Use a parameter sweep to find the ideal quantum parameters to use for the model.

5 Train the model by using the best configuration from the quantum parameter sweep.