

# Contents

[Azure Quantum Documentation](#)

[Azure Quantum \(preview\)](#)

[Overview](#)

[Introduction to Azure Quantum](#)

[Targets](#)

[Prepare your environment](#)

[Workspaces](#)

[Create quantum workspaces with the Azure Portal](#)

[Manage quantum workspaces with the Azure CLI](#)

[Quantum Computing](#)

[Quickstarts](#)

[IonQ quickstart](#)

[Honeywell quickstart](#)

[How-to guides](#)

[Create and run Q# applications in Azure Quantum](#)

[Submit jobs](#)

[Submit jobs with the Azure CLI](#)

[Submit jobs with Q# Jupyter Notebooks](#)

[Submit jobs with Python](#)

[Providers](#)

[IonQ](#)

[IonQ provider & targets](#)

[IonQ support policy](#)

[Honeywell](#)

[Honeywell provider & targets](#)

[Honeywell support policy](#)

[Optimization](#)

[Quickstarts](#)

[Microsoft QIO quickstart](#)

[1QBit quickstart](#)

## How-to guides

[Install and use the Python SDK for Azure Quantum](#)

[Troubleshoot optimization solvers](#)

## Concepts

[Introduction to optimization](#)

[What is quantum inspired optimization \(QIO\)?](#)

[Key concepts](#)

[Which solver should I use?](#)

[Binary optimization \(QUBOs & PUBOs\)](#)

[Ising model](#)

[Cost functions](#)

## Providers

[1QBit](#)

[1QBit provider & targets](#)

[1QBit support policy](#)

[Microsoft QIO](#)

[Overview](#)

[Simulated Annealing](#)

[Population Annealing](#)

[Parallel Tempering](#)

[Tabu Search](#)

[Quantum Monte Carlo](#)

[Substochastic Monte Carlo](#)

## Tutorials

[Samples for using Azure Quantum optimization solvers](#)

[Build your skills with MS Learn](#)

## Python SDK

[Azure.Quantum](#)

[Workspace](#)

[Azure.Quantum.Optimization](#)

[Job](#)

[Problem](#)

[ProblemType](#)

[StreamingProblem](#)

[Solver inputs](#)

[Term](#)

[RangeSchedule](#)

[Usage](#)

[Express a problem](#)

[Apply a solver](#)

[Understand solver results](#)

[Job management](#)

[Re-use problem definitions](#)

[Solve long-running problems](#)

[Resources](#)

[Glossary \(Azure Quantum\)](#)

[Common issues](#)

[Azure CLI commands](#)

[Authenticate using a service principal](#)

[Quantum Development Kit](#)

[Overview](#)

[Introduction to quantum computing](#)

[Key concepts for quantum computing](#)

[Understanding quantum computing](#)

[Quantum computers and simulators](#)

[What are Q# and the QDK?](#)

[Linear algebra for quantum computing](#)

[Get started](#)

[Set up the QDK](#)

[Get started with the QDK](#)

[Q# applications](#)

[Q# with Jupyter Notebooks](#)

[Q# with Python](#)

[Q# with .NET](#)

[Q# with Binder](#)

[Update the QDK](#)

## Tutorials

[Quantum random number generator](#)

[Explore entanglement with Q#](#)

[Implement Grover's search algorithm](#)

[Qubit-level programming](#)

[Learn Q# with the Quantum Katas](#)

[Q# code samples](#)

## Concepts

[Quantum computing history & background](#)

[Vectors and matrices](#)

[Advanced matrix concepts](#)

[The qubit](#)

[Multiple qubits](#)

[Dirac notation](#)

[Pauli measurements](#)

[Quantum circuits](#)

[Quantum oracles](#)

[Theory of Grover's algorithm](#)

## Resources

[Build your skills with MS Learn](#)

[Contributing to the QDK](#)

[Reporting Bugs](#)

[Improving Documentation](#)

[Opening Pull Requests](#)

[Contributing Code](#)

[Contributing Samples](#)

[Style Guide](#)

[API Design Principles](#)

[Q# user guide](#)

[Overview/Contents](#)

[Q# programs](#)

[Ways to run a Q# program](#)

[Testing and debugging](#)

[Q# language guide](#)

[Overview](#)

[Program structure](#)

[Program implementation](#)

[Namespaces](#)

[Type declarations](#)

[Callable declarations](#)

[Specialization declarations](#)

[Comments](#)

[Statements](#)

[Statements in Q#](#)

[Binding scopes](#)

[Call statements](#)

[Returns and termination](#)

[Variable declaration and reassignment](#)

[Iterations](#)

[Conditional loops](#)

[Conditional branching](#)

[Conjugations](#)

[Quantum memory management](#)

[Expressions](#)

[Expressions in Q#](#)

[Precedence and associativity](#)

[Operators](#)

[Copy-and-update expressions](#)

[Conditional expressions](#)

[Comparative expressions](#)

[Logical expressions](#)

[Bitwise expressions](#)

[Arithmetic expressions](#)

[Concatenations](#)

[Modifiers and combinators](#)

[Partial application](#)

[Functor application](#)

[Item access expressions](#)

[Contextual expressions](#)

[Value literals and default values](#)

[Type system](#)

[Type system in Q#](#)

[Quantum data types](#)

[Immutability](#)

[Operations and functions](#)

[Singleton tuple equivalence](#)

[Subtyping and variance](#)

[Type parameterizations](#)

[Grammar](#)

[Simulators and resource estimators](#)

[Overview](#)

[Full state simulator](#)

[Resources estimator](#)

[Trace simulator](#)

[Distinct inputs checker](#)

[Invalidated qubits use checker](#)

[Primitive operations counter](#)

[Depth counter](#)

[Width counter](#)

[Toffoli simulator](#)

[Q# libraries](#)

[Overview](#)

[Standard libraries](#)

[The prelude](#)

- Classical mathematics
- Type conversions
- Higher-order control flow
- Data structures and modeling
- Quantum algorithms
- Diagnostics
- Characterization
- Error correction
- Applications
- OSS licensing
- Using additional Q# libraries
- Quantum chemistry library
  - Installation and validation
  - Quantum chemistry concepts
    - Quantum dynamics
    - Quantum models for electronic systems
    - Second quantization
    - Symmetries of molecular integrals
    - Jordan-Wigner representation
    - Simulating Hamiltonian dynamics
    - Hartree-Fock theory
    - Correlated wavefunctions
  - Invoking the chemistry library
    - Obtaining energy level estimates
    - Loading a Hamiltonian from file
    - Obtaining resource counts
    - End-to-end with NWChem
  - Schema
    - Broombridge quantum chemistry schema
    - Specification v0.2
    - Specification v0.1
  - Quantum machine learning library

[Introduction to quantum machine learning](#)

[Basic classification](#)

[Design your own classifier](#)

[Load your own datasets](#)

[Glossary \(QML\)](#)

[Quantum numerics library](#)

[Using the numerics library](#)

[Release Notes](#)

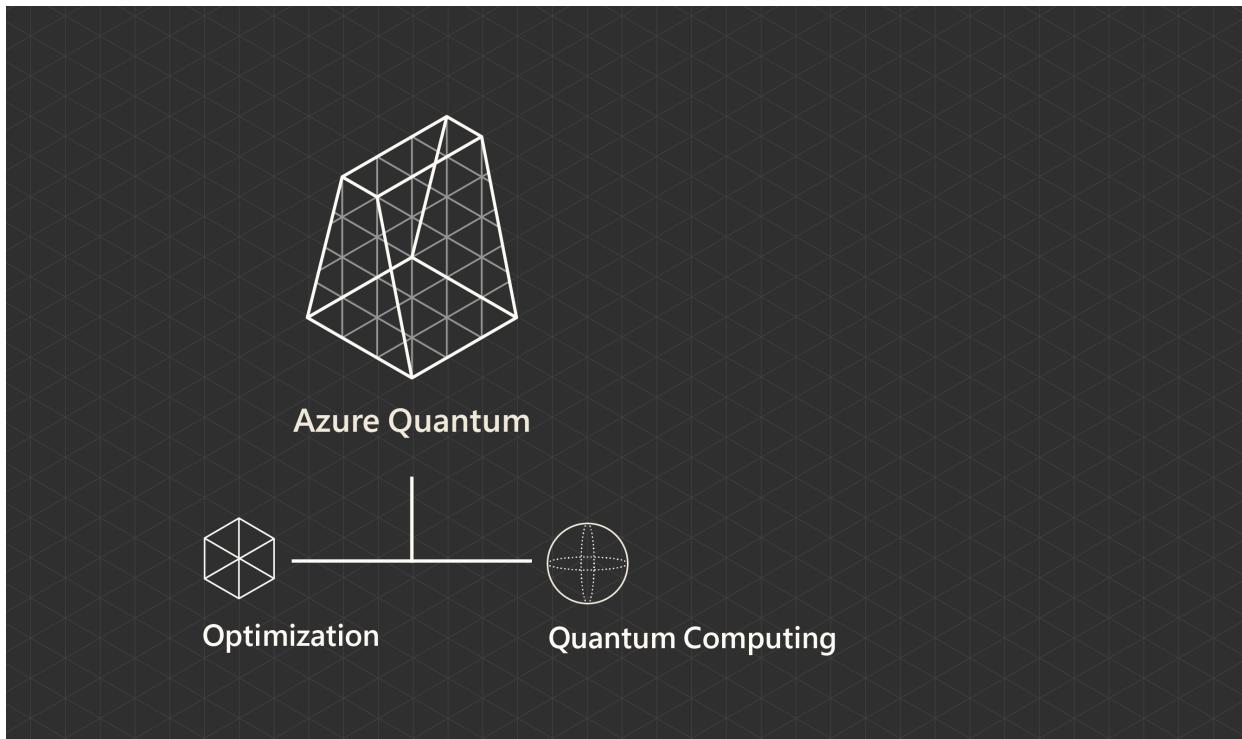
[Glossary](#)

[Further reading](#)

# Introduction to Azure Quantum (preview)

6/30/2021 • 5 minutes to read • [Edit Online](#)

Azure Quantum is a Microsoft Azure service that you can use to run quantum computing programs or solve optimization problems in the cloud. Using the Azure Quantum tools and SDKs, you can create quantum programs and run them against different quantum simulators and machines.



## Quantum computing

Azure Quantum service offers you access to different providers of quantum computing devices and enables you to run your Q# quantum programs in real hardware. [Q#](#) is a Microsoft's open-source programming language for developing and running your quantum algorithms. Azure Quantum also offers the option to run algorithms in simulated quantum computers to test your code. To learn more about quantum computers and quantum algorithms see [Introduction to quantum computing](#).

Azure Quantum provides access to trapped ion devices through the providers [IonQ](#) and [Honeywell](#).

## Optimization

Azure Quantum gives you access to a broad set of state-of-the-art optimization algorithms developed by Microsoft and its partners. You can use classic optimization algorithms, included some inspired by standard physics, as well as quantum-inspired optimization algorithms (QIO).

QIO uses algorithms that are based on quantum principles for increased speed and accuracy. Azure Quantum supports QIO to help developers leverage the power of new quantum techniques today without waiting for quantum hardware.

Optimization algorithms are available to run on a variety of classical computing silicon solutions, such as CPU, FPGA, GPU or custom silicon. If you want to learn more about optimization problems see [Introduction to optimization](#).

# Quantum workspace

You use the Azure Quantum service by adding an Azure Quantum workspace resource to your Azure subscription in the Azure portal. A Quantum workspace resource, or workspace for short, is a collection of assets associated with running quantum or optimization applications. One of the properties configured in a workspace is an Azure Storage Account resource, where Azure Quantum stores your quantum programs and optimization problems for access.

## Providers and targets

Another property configured in the workspace is the **provider** that you want to use to run programs in that workspace. A single provider may expose one or more **targets**, which can be quantum hardware or simulators, and are ultimately responsible for running your program. The complete list of targets can be found in the [quantum computing providers](#) and [optimization providers](#).

By default, Azure Quantum adds the Microsoft QIO provider to every workspace, and you can add other providers when you create the workspace or any time afterward. For more information, see the [Microsoft QIO provider](#).

### Provider billing

Each additional provider you add to a workspace requires a billing plan, which defines how that provider bills for usage. Each provider may have different billing plans and methods available. For more information, see the documentation on the provider you would like to add. Also, when you add a provider to a new workspace you can find in the description more information about current pricing options.

You can only select one billing plan for each provider in a single workspace; however, you can add multiple workspaces to your Azure subscription.

## Jobs

When you run a quantum program or solve an optimization problem in Azure Quantum, you create and run a **job**. The steps to create and run a job depend on the job type and the provider and target that you configure for the workspace. All jobs, however, have the following properties in common:

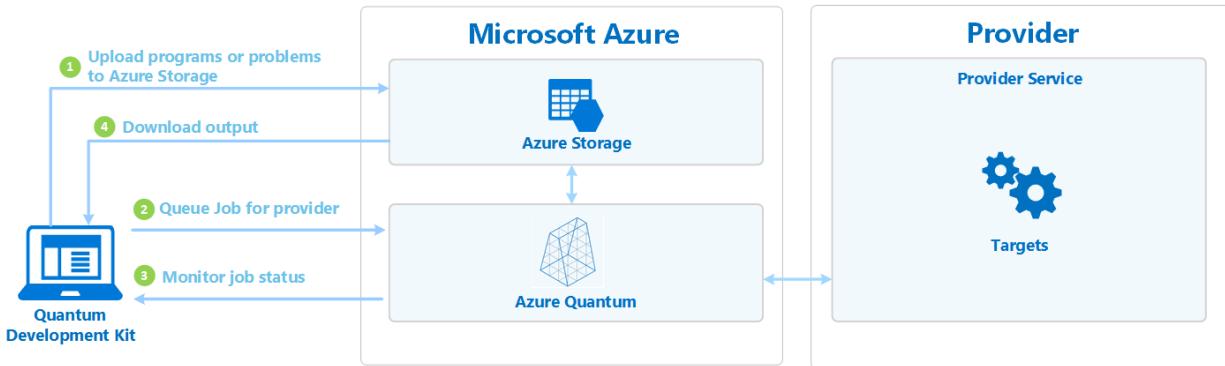
PROPERTY	DESCRIPTION
ID	A unique identifier for the job. It must be unique within the workspace.
Provider	<i>Who</i> you want to run your job. For example, the Microsoft QIO provider, or a third-party provider.
Target	<i>What</i> you want to run your job on. For example, the exact quantum hardware or quantum simulator offered by the provider.
Name	A user-defined name to help organize your jobs.
Parameters	Optional input parameters for targets. See the documentation for the selected target for a definition of available parameters.

Once you create a job, various metadata is available about its state and run history.

## Job lifecycle

You typically create jobs using one of the quantum SDKs (for example, the [Python SDK for Azure Quantum](#) or the [Quantum Development Kit \(QDK\)](#)). Once you've written your quantum program or expressed your QIO problem, you can select a target and submit your job.

This diagram shows the basic workflow after you submit your job:



First, Azure Quantum uploads the job to the Azure Storage account that you configured in the workspace. Then, the job is added to the job queue for the provider that you specified in the job. Azure Quantum then downloads your program and translates it for the provider. The provider processes the job and returns the output to Azure Storage, where it is available for download.

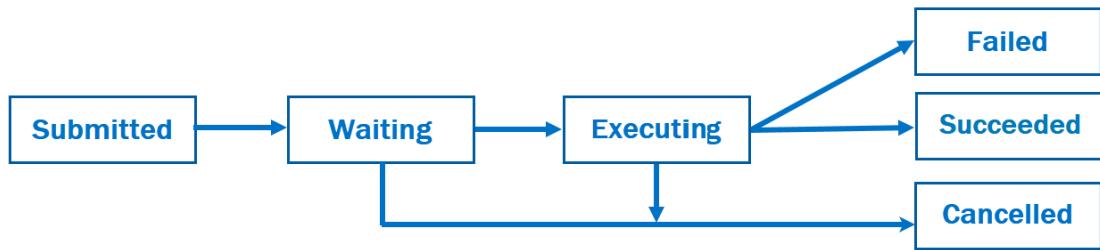
## Monitoring jobs

Once you submit a job, you must poll for the status of the job. Jobs have the following possible states:

STATUS	DESCRIPTION
<code>waiting</code>	The job is waiting to run. Some jobs will perform pre-processing tasks in the waiting state. <code>waiting</code> is always the first state, however, a job may move to the <code>executing</code> state before you can observe it in <code>waiting</code> .
<code>executing</code>	The target is currently running the job.
<code>succeeded</code>	The job has succeeded, and output is available. This is a <i>final</i> state.
<code>failed</code>	The job has failed, and error information is available. This is a <i>final</i> state.
<code>cancelled</code>	The user requested to cancel the job run. This is a <i>final</i> state. For more information, see <a href="#">Job Cancellation</a> in this article.

The `succeeded`, `failed`, and `cancelled` states are considered **final states**. Once a job is in one of these states, no more updates occur, and the corresponding job output data does not change.

This diagram shows the possible job state transitions:



After a job completes successfully, it displays a link to the output data in your Azure Storage account. How you access this data depends on the SDK or tool you used to submit the job.

### Job cancellation

When a job is not yet in a final state (for example, `succeeded`, `failed`, or `cancelled`), you can request to cancel the job. All providers will cancel your job if it is in the `waiting` state. However, not all providers support cancellation if your job is in the `executing` state.

#### NOTE

If you cancel a job after it has started running, your account may still be billed a partial or full amount for that job. See the billing documentation for your selected provider.

## Next steps

When you're ready to get started, begin by [creating an Azure Quantum workspace](#).

# Targets in Azure Quantum

4/29/2021 • 2 minutes to read • [Edit Online](#)

This article introduces the different type of targets available in Azure Quantum and the Quantum Development Kit (QDK). Targets in Azure Quantum can be solvers for optimization problems or quantum devices (either physical or simulated) that you can use to run Q# quantum applications.

Currently, Azure Quantum includes the following types of targets:

## Optimization solvers

Azure Quantum offers optimization targets to solve binary optimization problems on classical CPUs, or hardware accelerated on field-programmable gate arrays (FPGA), GPUs or hardware annealers.

For more information on optimization, see [Optimization solvers](#).

## Quantum devices

Azure Quantum also offers a variety of quantum solutions, such as different hardware devices and quantum simulators. At this time, because of the early development stage of the field, these devices have some limitations and requirements for programs that run on them. The Quantum Development Kit and Azure Quantum will keep track of these requirements in the background so that you can run Q# programs on Azure Quantum targets.

### Quantum Processing Units (QPU): different profiles

A quantum processing unit (QPU) is a physical or simulated processor that contains a number of interconnected qubits that can be manipulated to compute quantum algorithms. It's the central component of a quantum computer.

Quantum devices are still an emerging technology, and not all of them can run all Q# code. As such, you need to keep some restrictions in mind when developing programs for different targets. Currently, Azure Quantum and the QDK manage three different profiles for QPUs:

- **Full:** This profile can run any Q# program within the limits of memory for simulated quantum processing units (QPU) or the number of qubits of the physical quantum hardware.
- **No Control Flow:** This profile can run any Q# program that doesn't require the use of the results from qubit measurements to control the program flow. Within a Q# program targeted for this kind of QPU, values of type `Result` do not support equality comparison.
- **Basic Measurement Feedback:** This profile has limited ability to use the results from qubit measurements to control the program flow. Within a Q# program targeted for this kind of QPU, you can only compare values of type `Result` as part of conditions within `if` statements in operations. The corresponding conditional blocks may not contain `return` or `set` statements.

## Next steps

You can find a complete list of the Azure Quantum targets for quantum computing [here](#) and for optimization [here](#).

# Prepare your environment to use Azure Quantum from the command prompt

5/27/2021 • 2 minutes to read • [Edit Online](#)

Azure Quantum uses the Azure CLI `quantum` extension to enable submitting Q# programs from the command line. This guide provides the steps to install and configure the Azure CLI extension on your system for use with Azure Quantum.

## Prerequisites

Before installing the Azure CLI `quantum` extension, ensure that the following packages are installed:

- The Microsoft [Quantum Development Kit](#)
- The latest version of [Azure CLI](#) (version 2.5.0 or higher)

## Installation

To install the Azure CLI `quantum` extension, open a command prompt, and then run the following command:

```
az extension add -n quantum
```

## Uninstall the extension

To uninstall the Azure CLI `quantum` extension, run the following command:

```
az extension remove -n quantum
```

## Update the extension

If you need to update an existing installation of the the Azure CLI `quantum` extension, you can run:

```
az extension update -n quantum
```

### NOTE

If you have previously installed a pre-release version of this extension, or you are not sure about your current installation you can uninstall it and then install it again using the instructions above.

## Next steps

Now that you have installed the tools to use Azure Quantum you can learn to submit jobs.

### For optimization users

Learn how to [use the Python SDK](#) to solve optimization problems.

### For quantum computing users

Learn how to [create Q# applications](#) and run them on Azure Quantum.

# Create Azure Quantum workspaces with the Azure portal

6/1/2021 • 2 minutes to read • [Edit Online](#)

In this guide, learn to create Azure Quantum workspaces and the required Resource Groups and Storage Accounts using the Azure portal, and start running your quantum applications in Azure Quantum.

## Prerequisites

In order to use the Azure Quantum service, you will need an active Azure subscription. To create an Azure subscription go to the [Azure free sign-up page](#) and click on the green **Start free** button to start the process of creating an Azure subscription.

### NOTE

You will need to introduce your billing information (for example, a valid credit card) to create your free Azure subscription.

## Create your Azure free account today

Get started with 12 months of free services

[Start free](#)

[Or buy now >](#)

Once you have an active Azure subscription, you can continue with the next section [Create an Azure Quantum workspace](#).

## Create an Azure Quantum workspace

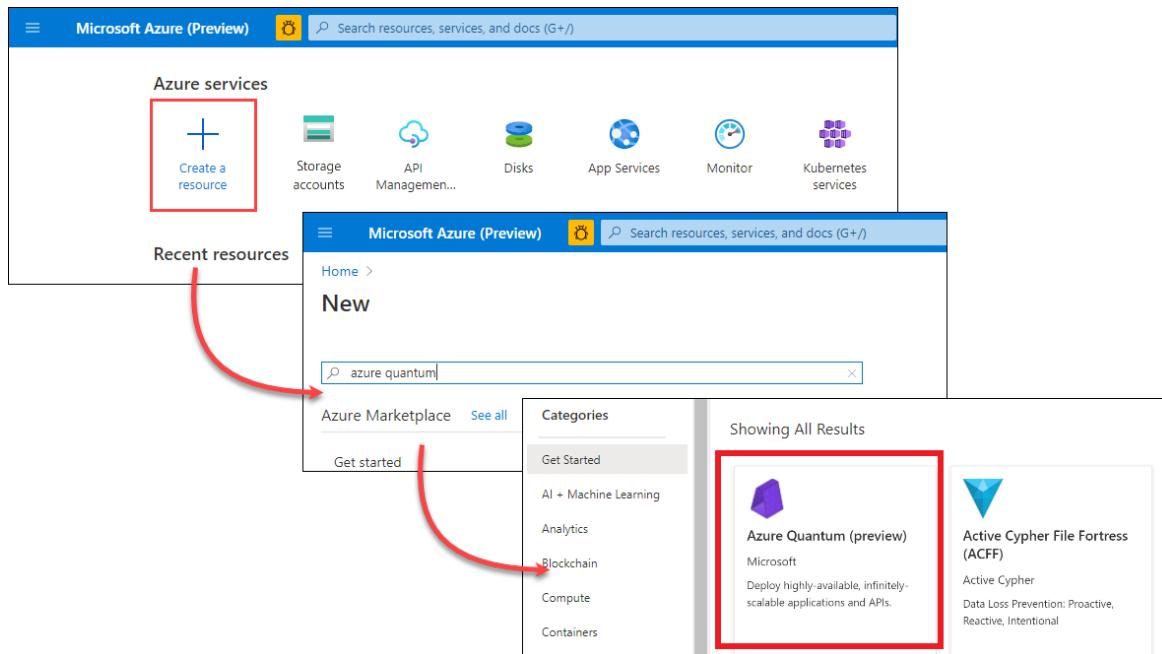
You use the Azure Quantum service by adding an Azure Quantum workspace resource to your Azure subscription in the Azure portal. An Azure Quantum workspace resource, or workspace for short, is a collection of assets associated with running quantum or optimization applications.

To open the Azure Portal, go to <https://portal.azure.com> and then follow these steps:

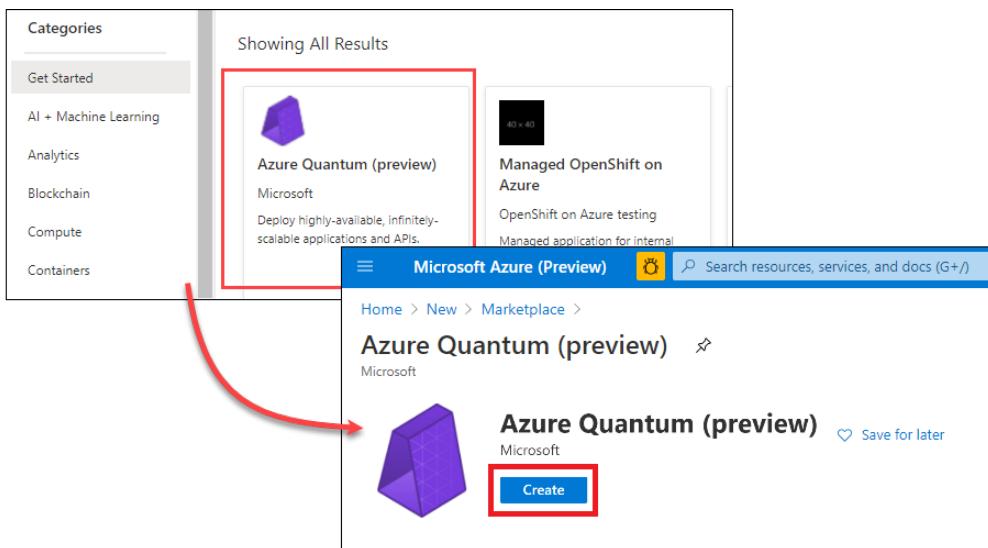
### NOTE

If you just created your Azure subscription, you might be asked to confirm your email address in order to use the service.

1. Click **Create a resource** and then search for **Azure Quantum**. On the results page, you should see a tile for the **Azure Quantum (preview)** service.



2. Click **Azure Quantum (preview)** and then click **Create**. This opens a form to create a workspace.



3. Fill out the details of your workspace:

- **Subscription:** The subscription that you want to associate with this workspace.
- **Resource group:** The resource group that you want to assign this workspace to.
- **Name:** The name of your workspace.
- **Region:** The region for the workspace.
- **Storage Account:** The Azure storage account to store your jobs and results. If you don't have an existing storage account, click **Create a new storage account** and complete the necessary fields. For this preview, we recommend using the default values.

**Project details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

<b>Subscription *</b> ⓘ └── Resource group * ⓘ	Azure Quantum PM and Outreach
Documentation	
<a href="#">Create new</a>	
<b>Instance details</b>	
Name * ⓘ	bbQuantumWksp3
Region * ⓘ	(US) West US
Storage Account ⓘ	quantumdoc
<a href="#">Create a new storage account</a>	

[Review + create](#) [Previous](#) [Providers >](#)

#### NOTE

You must be an Owner of the selected resource group to create a new storage account. For more information about how resource groups work in Azure, see [Control and organize Azure resources with Azure Resource Manager](#).

- After completing the information, click the **Providers** tab to add providers to your workspace. A provider gives you access to a quantum service, which can be quantum hardware, a quantum simulator, or an optimization service.

**Available providers**

 IQcloud Optimization Platform Optimization  1QBit 1Qcloud Optimization Platform with Quantum Inspired Solutions  <a href="#">+ Add</a>	 Honeywell Honeywell Quantum Solutions Quantum Computing  Access to Honeywell Quantum Solutions' trapped-ion systems  <a href="#">+ Add</a>	 IonQ Quantum Computing  IonQ's trapped ion quantum computers perform calculations by manipulating charged atoms of Ytterbium held in a vacuum with lasers.  <a href="#">+ Add</a>	 Microsoft QIO Optimization  Ground-breaking optimization algorithms inspired by decades of quantum research.   <a href="#">Added</a>
---	--	--	--

**Providers added to this workspace**

Name ↑↓	Provider type ↑↓	SKU
 Microsoft QIO Microsoft	Optimization	Learn & Develop

[Review + create](#) [Previous](#) [Tags >](#)

#### NOTE

By default, Azure Quantum adds the Microsoft QIO provider to every workspace.

- After adding the providers that you want to use, click **Review + create**.
- Review the settings and approve the *Terms and Conditions of Use* of the selected providers. If everything is correct, click **Create** to create your workspace.



## Create Quantum Workspace

Quantum Workspace

Basics Providers Tags Review + create

Terms and conditions need to be accepted for the providers you selected. [Accept terms](#)

### Terms and conditions

Please read and accept the terms and condition before enabling the following providers:

^ IonQ

I accept the terms and conditions.

**NOTE**

Pricing for Azure Quantum varies by provider. Please consult the information in the Providers tab of your Azure Quantum workspace in the Azure portal for the most up-to-date pricing information.

## Next steps

Now that you created a workspace, learn about the different [targets to run quantum algorithms in Azure Quantum](#).

# Manage quantum workspaces with the Azure CLI

5/27/2021 • 3 minutes to read • [Edit Online](#)

In this guide, learn to create Azure Quantum workspaces and the required Resource Groups and Storage Accounts using the Azure Command-Line Interface (Azure CLI) and start running your quantum applications in Azure Quantum.

## Prerequisites

In order to use the Azure Quantum service, you will need:

- An active Azure account and subscription. For more information, see the Microsoft Learn module [Create an Azure account](#).
- The [Azure CLI](#).
- The [necessary utilities to use Azure Quantum](#) (includes the `quantum` extension for the Azure CLI).
- An Azure resource group where the quantum workspace will live.
- A storage account in the resource group to be associated with the quantum workspace. Multiple workspaces can be associated with the same account.

## Environment setup

1. Log in to Azure using your credentials. You'll get a list of subscriptions associated with your account.

```
az login
```

2. Specify the subscription you want to use from those associated with your Azure account.

```
az account set -s <Your subscription ID>
```

3. If this is the first time you will be creating quantum workspaces in your subscription, register the resource provider with this command:

```
az provider register --namespace Microsoft.Quantum
```

## Create an Azure Quantum workspace

In order to create a new Azure Quantum workspace, you'll need to know:

- The location or Azure region name where the resource will live. You can use the [list of regions and their resource manager codes](#) supported by the Azure CLI tool (for example, `westus`).
- The resource group associated with the new workspace. (for example, `MyResourceGroup`).
- An storage account on the same resource group and subscription than the quantum workspace. It's possible to [create a new storage account from the Az CLI tool](#). (for example, `MyStorageAccount`)
- The name of the quantum workspace to create. (for example, `MyQuantumWorkspace`)
- The list of Azure Quantum providers to use in the workspace. A provider offers a set of SKUs, each of them represents a plan with associated terms and conditions, cost and quotas. To create workspaces you'll need to specify not only providers but also the corresponding SKU.

If you already know the providers and SKU names to use in your workspace, you can skip to step 4 below. Otherwise, we should determine which ones to use first.

1. To retrieve the list of quantum providers available, you can use the following command (using `westus` as example location) :

```
az quantum offerings list -l westus -o table
```

2. Once you determine the provider and SKU to include in your workspace, you can review terms using this command, assuming `MyProviderID` and `MySKU` as example values:

```
az quantum offerings show-terms -l westus -p MyProviderId -k MySKU
```

3. The output of the command above includes a Boolean field `accepted` that shows if the terms for this provider have been accepted already or not, as well as a link to the license terms to review. If you decide to accept those terms, use the following command to record your acceptance.

```
az quantum offerings accept-terms -l westus -p MyProviderId -k MySKU
```

4. Once you have reviewed and accepted all terms and conditions required, you can create your workspace specifying a list of provider/SKU combinations separated by commas, as in the example below:

```
az quantum workspace create -l westus -g MyResourceGroup -w MyQuantumWorkspace -a MyStorageAccount -r "MyProvider1/MySKU1, MyProvider2/MySKU2"
```

Once a workspace has been created, you can still add or remove providers using the Azure Portal.

## Delete a quantum workspace

If you know the name and resource group of a quantum workspace you want to delete, you can do it with the following command (using the same names as the example above):

```
az quantum workspace delete -g MyResourceGroup -w MyQuantumWorkspace
```

### TIP

If you don't remember the exact name, you can view the entire list of quantum workspaces in your subscription using

```
az quantum workspace list -o table
```

After you delete a workspace, you will still see it listed while it's being deleted in the cloud, however, the `provisioningState` property of the workspace will immediately change to indicate it's being deleted. You can see this information by using the following command:

```
az quantum workspace show -g MyResourceGroup -w MyQuantumWorkspace
```

### NOTE

In case you used the `az quantum workspace set` command previously to specify a default quantum workspace, then calling the command without parameters will delete (and clear) the default workspace.

```
az quantum workspace delete
```

## Next steps

Now that you can created and delete workspaces, you can learn about the different [targets to run quantum algorithms in Azure Quantum](#).

# IonQ quickstart for Azure Quantum

6/1/2021 • 8 minutes to read • [Edit Online](#)

Learn how to use Azure Quantum to run Q# problems against the IonQ simulator or QPU.

## Prerequisites

- To complete this tutorial you need an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- In this guide we'll use [Visual Studio Code](#) which you can download and use for free.

## Install the Quantum Development Kit (QDK) and other resources

Before you can write a Q# program and run it with IonQ, you'll need a few resources installed:

1. Install the [Microsoft QDK for VS Code extension](#).
2. Install the [Azure CLI](#).
3. Install the `quantum` CLI extension for the Azure CLI.

```
az extension add -n quantum
```

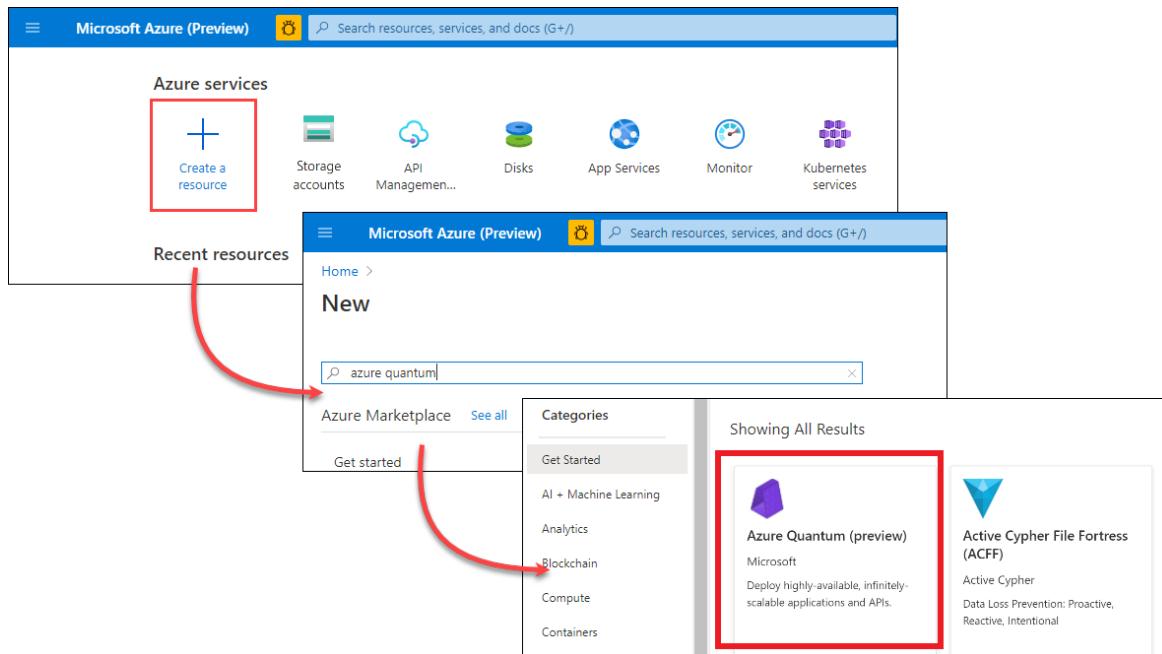
## Create an Azure Quantum workspace

You use the Azure Quantum service by adding an Azure Quantum workspace resource to your Azure subscription in the Azure portal. An Azure Quantum workspace resource, or workspace for short, is a collection of assets associated with running quantum or optimization applications.

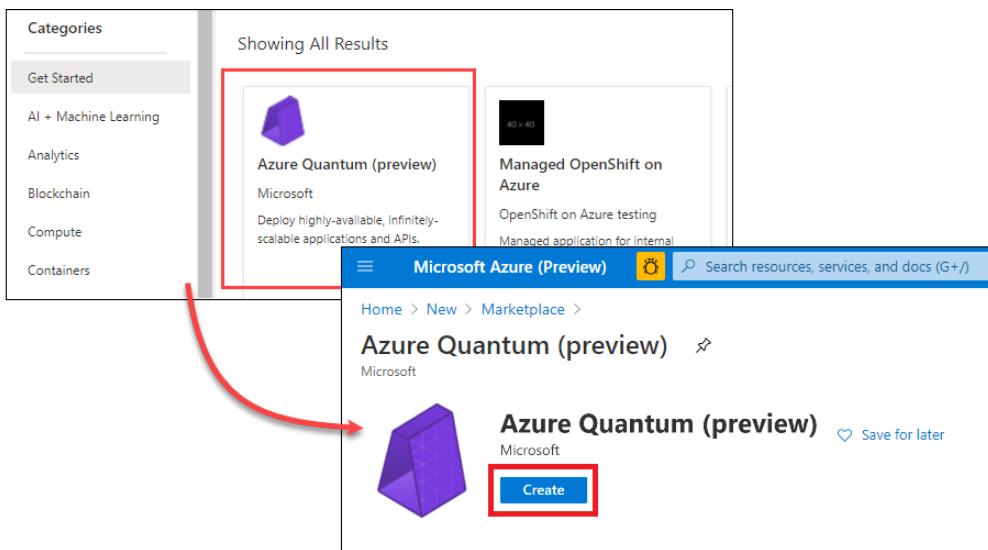
To open the Azure Portal, go to <https://portal.azure.com> and then follow these steps:

Note: This is a special link that allows you to create a workspace in the Azure Portal. Without using the link you will be able to see existing workspaces but not create new ones.

1. Click **Create a resource** and then search for **Azure Quantum**. On the results page, you should see a tile for the **Azure Quantum (preview)** service.



2. Click **Azure Quantum (preview)** and then click **Create**. This opens a form to create a workspace.



3. Fill out the details of your workspace:

- **Subscription:** The subscription that you want to associate with this workspace.
- **Resource group:** The resource group that you want to assign this workspace to.
- **Name:** The name of your workspace.
- **Region:** The region for the workspace.
- **Storage Account:** The Azure storage account to store your jobs and results. If you don't have an existing storage account, click **Create a new storage account** and complete the necessary fields. For this preview, we recommend using the default values.

**Project details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

<b>Subscription *</b> ⓘ └── Resource group * ⓘ	Azure Quantum PM and Outreach
Documentation	
<a href="#">Create new</a>	
<b>Instance details</b>	
Name * ⓘ	bbQuantumWksp3
Region * ⓘ	(US) West US
Storage Account ⓘ	quantumdoc
<a href="#">Create a new storage account</a>	

**Review + create**    Previous    [Providers >](#)

#### NOTE

You must be an Owner of the selected resource group to create a new storage account. For more information about how resource groups work in Azure, see [Control and organize Azure resources with Azure Resource Manager](#).

- After completing the information, click the **Providers** tab to add providers to your workspace. A provider gives you access to a quantum service, which can be quantum hardware, a quantum simulator, or an optimization service.

#### NOTE

If you do not see the IonQ provider, you may not have access to their preview yet. If you have received an email welcoming you to the IonQ Preview but you can't see their provider, please [create a ticket with Azure Support](#).

#### Available providers

 <b>1Qcloud Optimization Platform</b> Optimization  1QBit 1Qcloud Optimization Platform with Quantum Inspired Solutions	 <b>Honeywell Quantum Solutions</b> Quantum Computing  Access to Honeywell Quantum Solutions' trapped-ion systems	 <b>IonQ</b> Quantum Computing  IonQ's trapped ion quantum computers perform calculations by manipulating charged atoms of Ytterbium held in a vacuum with lasers.	 <b>Microsoft QIO</b> Optimization  Ground-breaking optimization algorithms inspired by decades of quantum research.
<a href="#">+ Add</a>	<a href="#">+ Add</a>	<a href="#">+ Add</a>	✓ Added

#### Providers added to this workspace

Name ↑↓	Provider type ↑↓	SKU	 Modify	 Remove
Microsoft QIO Microsoft	Optimization	Learn & Develop		

**Review + create**    Previous    [Tags >](#)

#### NOTE

By default, the Azure Quantum service adds the Microsoft QIO provider to every workspace.

5. Add at least the IonQ provider, then click **Review + create**.
6. Review the settings and approve the *Terms and Conditions of Use* of the selected providers. If everything is correct, click **Create** to create your workspace.

The screenshot shows the Microsoft Azure (Preview) portal with the URL [https://portal.azure.com/#create/Microsoft.IonQ/IonQ-Quantum-Workshop](#). The page title is "Create Quantum Workspace". The navigation bar includes "Dashboard", "Marketplace", "Azure Quantum (preview)", and "Create Quantum Workspace". Below the navigation is a "Quantum Workspace" section with tabs: "Basics", "Providers", "Tags", and "Review + create" (which is underlined). A red box highlights a warning message: "⚠️ Terms and conditions need to be accepted for the providers you selected. [Accept terms](#)". Below this, there's a "Terms and conditions" section with the heading "Please read and accept the terms and condition before enabling the following providers:" followed by a list of providers, with "IonQ" expanded. Under "IonQ", there is a checkbox labeled "I accept the terms and conditions." which is checked.

#### NOTE

Pricing for Azure Quantum varies by provider. Please consult the information in the Providers tab of your Azure Quantum workspace in the Azure portal for the most up-to-date pricing information.

## Setup your project and write your program

Next, we'll open up Visual Studio Code and get create a Q# Project.

1. In VS Code open the **View** menu and select **Command Palette**.
2. Type **Q#: Create New Project**.
3. Select **Standalone console application**.
4. Select a directory to hold your project, such as your home directory. Enter **QuantumRNG** as the project name, then select **Create Project**.
5. From the window that appears at the bottom, select **Open new project**.
6. You should see two files: the project file and **Program.qs**, which contains starter code. Open **Program.qs**.
7. Start by opening the **QuantumRNG.csproj** file and adding the **ExecutionTarget** property, which will give you design-time feedback on the compatibility of your program for IonQ's hardware.

```
<Project Sdk="Microsoft.Quantum.Sdk/0.17.2105143879">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <ExecutionTarget>ionq.qpu</ExecutionTarget>
  </PropertyGroup>
</Project>
```

1. Replace the contents of `Program.qs` with the program:

```
namespace QuantumRNG {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Canon;

    @EntryPoint()
    operation GenerateRandomBits() : Result[] {
        use qubits = Qubit[4];
        ApplyToEach(H, qubits);
        return MultiM(qubits);
    }
}
```

**NOTE**

If you would like to learn more about this program code, we recommend [Create your first Q# program by using the Quantum Development Kit](#).

## Prepare the AZ CLI

Next, we'll prepare your environment to run the program against the workspace you created.

1. Log in to Azure using your credentials. You'll get a list of subscriptions associated with your account.

```
az login
```

2. Specify the subscription you want to use from those associated with your Azure account. You can also find your subscription ID in the overview of your workspace in Azure Portal.

```
az account set -s <Your subscription ID>
```

3. Use `quantum workspace set` to select the workspace you created above as the default Workspace. Note that you also need to specify the resource group and the location you created it in:

```
az quantum workspace set -g MyResourceGroup -w MyWorkspace -l MyLocation -o table
```

Location	Name	ProvisioningState	ResourceGroup	StorageAccount	Usable
MyLocation	MyWorkspace	Succeeded	MyResourceGroup	/subscriptions/...	Yes

4. In your workspace, there are different targets available from the providers that you added when you created the workspace. You can display a list of all the available targets with the command

```
az quantum target list -o table
```

```
az quantum target list -o table
```

Provider	Target-id	Status	Average Queue Time
ionq	ionq.qpu	Available	0
ionq	ionq.simulator	Available	0

#### NOTE

When you submit a job in Azure Quantum it will wait in a queue until the provider is ready to run your program. The **Average Queue Time** column of the target list command shows you how many seconds recently run jobs waited in the queue. This can give you an idea of how long you might have to wait.

## Simulate the program

Before you run a program against real hardware, we recommend simulating it first (if possible, based on the number of qubits required) to help ensure that your algorithm is doing what you want. Fortunately, IonQ provides an idealized simulator that you can use.

#### NOTE

You can also simulate Q# programs locally using the [Full State Simulator](#).

Run your program with `az quantum execute --target-id ionq.simulator -o table`. This command will compile your program, submit it to Azure Quantum, and wait until IonQ has finished simulating the program. Once it's done it will output a histogram which should look like the one below:

```
az quantum execute --target-id ionq.simulator -o table
```

Result	Frequency
[0,0,0,0]	0.06250000
[1,0,0,0]	0.06250000
[0,1,0,0]	0.06250000
[1,1,0,0]	0.06250000
[0,0,1,0]	0.06250000
[1,0,1,0]	0.06250000
[0,1,1,0]	0.06250000
[1,1,1,0]	0.06250000
[0,0,0,1]	0.06250000
[1,0,0,1]	0.06250000
[0,1,0,1]	0.06250000
[1,1,0,1]	0.06250000
[0,0,1,1]	0.06250000
[1,0,1,1]	0.06250000
[0,1,1,1]	0.06250000
[1,1,1,1]	0.06250000

This shows an equal frequency for each of the 16 possible states for measuring 4 qubits, which is what we expect from an idealized simulator! This means we're ready to run it on the QPU.

## Run the program on hardware

To run the program on hardware, we'll use the asynchronous job submission command `az quantum job submit`. Like the `execute` command this will compile and submit your program, but it won't wait until the execution is complete. We recommend this pattern for running against hardware, because you may need to wait a while for

your job to finish. To get an idea of how long you can run `az quantum target list -o table` as described above.

```
az quantum job submit --target-id ionq.qpu -o table
```

Name	Id	Status	Target	Submission time
QuantumRNG	5aa8ce7a-25d2-44db-bbc3-87e48a97249c	Waiting	ionq.qpu	2020-10-22T22:41:27.8855301+00:00

The table above shows that your job has been submitted and is waiting for its turn to run. To check on the status, use the `az quantum job show` command, being sure to replace the `job-id` parameter with the Id output by the previous command:

```
az quantum job show -o table --job-id 5aa8ce7a-25d2-44db-bbc3-87e48a97249c
```

Name	Id	Status	Target	Submission time
QuantumRNG	5aa8ce7a-25d2-44db-bbc3-87e48a97249c	Waiting	ionq.qpu	2020-10-22T22:41:27.8855301+00:00

Eventually, you will see the `Status` in the above table change to `Succeeded`. Once that's done you can get the results from the job by running `az quantum job output`:

```
az quantum job output -o table --job-id 5aa8ce7a-25d2-44db-bbc3-87e48a97249c
```

Result	Frequency
[0,0,0,0]	0.05200000
[1,0,0,0]	0.07200000
[0,1,0,0]	0.05000000
[1,1,0,0]	0.06800000
[0,0,1,0]	0.04600000
[1,0,1,0]	0.06000000
[0,1,1,0]	0.06400000
[1,1,1,0]	0.07600000
[0,0,0,1]	0.04800000
[1,0,0,1]	0.06200000
[0,1,0,1]	0.07400000
[1,1,0,1]	0.08000000
[0,0,1,1]	0.05800000
[1,0,1,1]	0.06800000
[0,1,1,1]	0.05200000
[1,1,1,1]	0.07000000

The histogram you receive may be slightly different than the one above, but you should find that the states generally are observed with equal frequency.

#### NOTE

If you run into an error while working with Azure Quantum, you can check our [list of common issues](#).

## Next steps

This quickstart guide demonstrated how to get started running Q# programs against IonQ's simulator and QPU. For more information on the IonQ offering, please see the [IonQ Provider](#).

We recommend you continue your journey by learning more about the [different types of targets in Azure Quantum](#), which will dictate which Q# programs you may run against a given provider. You might also be interested in learning how to submit Q# jobs with [Jupyter Notebooks](#) or with [Python](#).

Looking for more samples to run? Check out the [samples directory](#) for Azure Quantum.

Lastly, if you would like to learn more about writing Q# programs please see the [Microsoft Quantum Documentation](#).

# Honeywell quickstart for Azure Quantum

7/9/2021 • 6 minutes to read • [Edit Online](#)

Learn how to use Azure Quantum to run Q# problems against the Honeywell simulator or QPU.

## Prerequisites

- To complete this tutorial you need an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- In this guide we'll use [Visual Studio Code](#) which you can download and use for free.

## Install the Quantum Development Kit and other resources

Before you can write a Q# program and run it with Honeywell, you'll need a few resources installed:

1. Install the [Microsoft QDK for VS Code extension](#).
2. Install the [Azure CLI](#).
3. Install the `quantum` CLI extension for the Azure CLI.

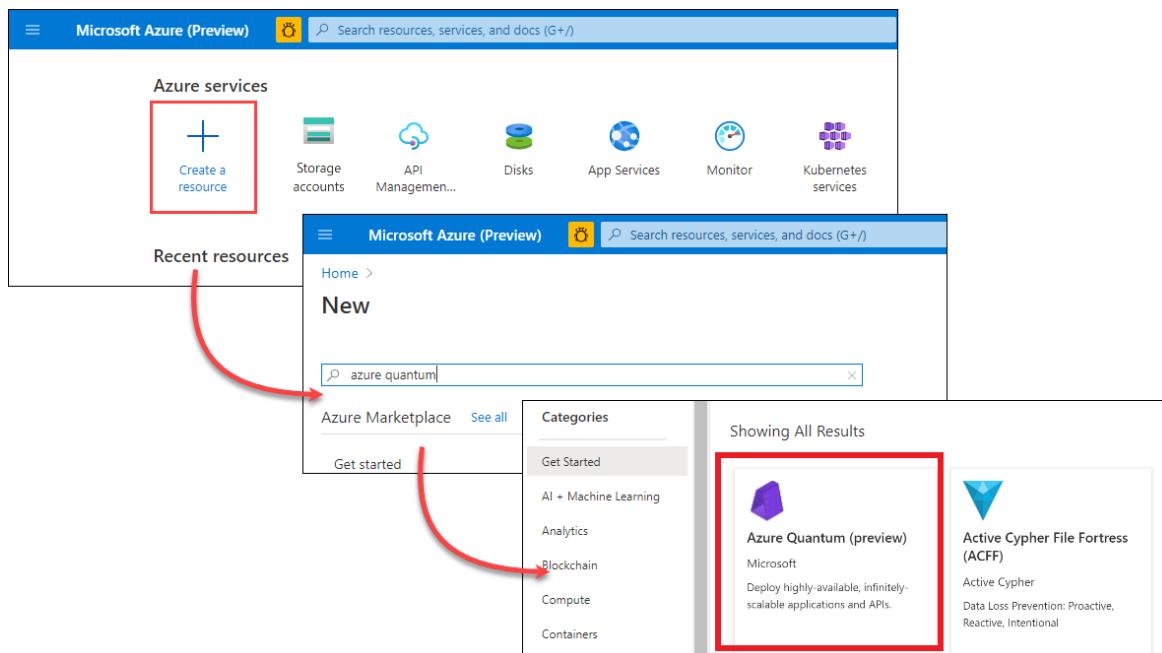
```
az extension add -n quantum
```

## Create an Azure Quantum workspace

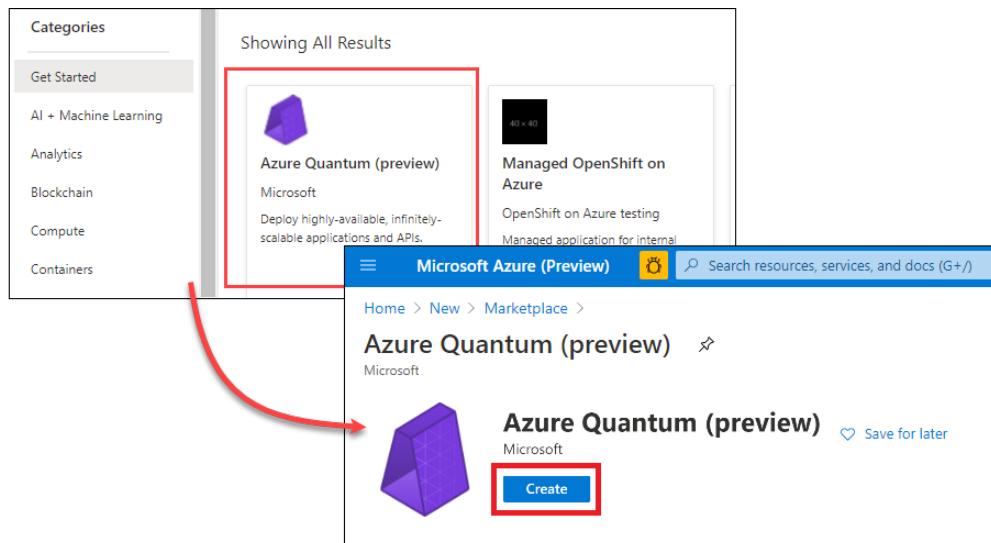
You use the Azure Quantum service by adding an Azure Quantum workspace resource to your Azure subscription in the Azure portal. An Azure Quantum workspace resource, or workspace for short, is a collection of assets associated with running quantum or optimization applications.

To open the Azure Portal, go to <https://portal.azure.com> and then follow these steps:

1. Click **Create a resource** and then search for **Azure Quantum**. On the results page, you should see a tile for the **Azure Quantum (preview)** service.



2. Click **Azure Quantum (preview)** and then click **Create**. This opens a form to create a workspace.



3. Fill out the details of your workspace:

- **Subscription:** The subscription that you want to associate with this workspace.
- **Resource group:** The resource group that you want to assign this workspace to.
- **Name:** The name of your workspace.
- **Region:** The region for the workspace.
- **Storage Account:** The Azure storage account to store your jobs and results. If you don't have an existing storage account, click **Create a new storage account** and complete the necessary fields. For this preview, we recommend using the default values.

**Project details**  
Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

**Subscription \*** ⓘ  
Azure Quantum PM and Outreach

**Resource group \*** ⓘ  
Documentation  
Create new

**Instance details**

**Name \*** ⓘ bbQuantumWksp3

**Region \*** ⓘ (US) West US

**Storage Account** ⓘ quantumdoc  
Create a new storage account

**Review + create**   **Previous**   **Providers >**

#### NOTE

You must be an Owner of the selected resource group to create a new storage account. For more information about how resource groups work in Azure, see [Control and organize Azure resources with Azure Resource Manager](#).

4. After completing the information, click the **Providers** tab to add providers to your workspace. A provider gives you access to a quantum service, which can be quantum hardware, a quantum simulator, or an

optimization service.

#### NOTE

If you do not see the Honeywell provider, you may not have access to their preview yet. If you have received an email welcoming you to the Honeywell Preview but you can't see their provider, please [create a ticket with Azure Support](#).

#### Available providers

 IQloud Optimization Platform Optimization  1QBit IQloud Optimization Platform with Quantum Inspired Solutions	 Honeywell Honeywell Quantum Solutions Quantum Computing  Access to Honeywell Quantum Solutions' trapped-ion systems	 IonQ Quantum Computing  IonQ's trapped ion quantum computers perform calculations by manipulating charged atoms of Ytterbium held in a vacuum with lasers.	 Microsoft QIO Optimization  Ground-breaking optimization algorithms inspired by decades of quantum research.
<a href="#">+ Add</a>	<a href="#">+ Add</a>	<a href="#">+ Add</a>	<a href="#">✓ Added</a>

#### Providers added to this workspace

Name ↑↓	Provider type ↑↓	SKU	Actions
 Microsoft QIO Microsoft	Optimization	Learn & Develop	<a href="#">Modify</a> <a href="#">Remove</a>

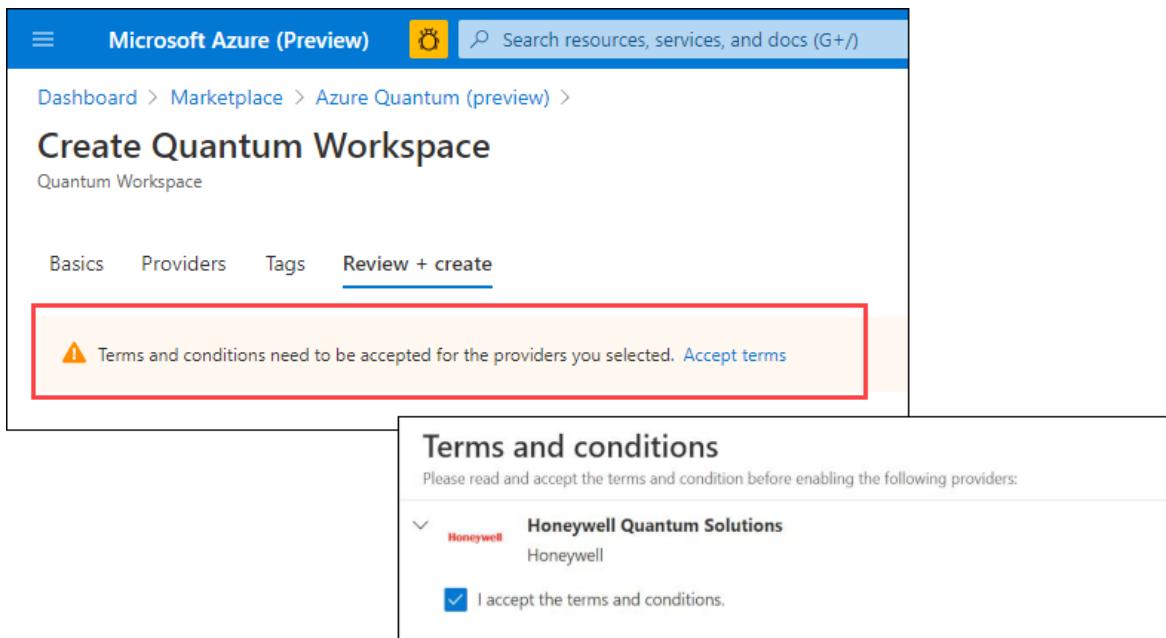
[Review + create](#) [Previous](#) [Tags >](#)

#### NOTE

By default, the Azure Quantum service adds the Microsoft QIO provider to every workspace.

5. Add at least the Honeywell provider, then click **Review + create**.

6. Review the settings and approve the *Terms and Conditions of Use* of the selected providers. If everything is correct, click **Create** to create your workspace.



**Microsoft Azure (Preview)** Search resources, services, and docs (G+ /)

Dashboard > Marketplace > Azure Quantum (preview) >

## Create Quantum Workspace

Quantum Workspace

Basics Providers Tags **Review + create**

**⚠️ Terms and conditions need to be accepted for the providers you selected. [Accept terms](#)**

**Terms and conditions**  
Please read and accept the terms and condition before enabling the following providers:

Honeywell Quantum Solutions  
Honeywell

I accept the terms and conditions.

#### NOTE

Pricing for Azure Quantum varies by provider. Please consult the information in the Providers tab of your Azure Quantum workspace in the Azure portal for the most up-to-date pricing information.

# Setup your project and write your program

Next, we'll open up Visual Studio Code and get create a Q# Project.

1. In VS Code open the **View** menu, and select **Command Palette**.
2. Type **Q#: Create New Project**.
3. Select **Standalone console application**.
4. Select a directory to hold your project, such as your home directory. Enter **QuantumRNG** as the project name, then select *Create Project*.
5. From the window that appears at the bottom, select **Open new project**.
6. You should see two files: the project file and **Program.qs**, which contains starter code. Open **Program.qs**.
7. Start by opening the **QuantumRNG.csproj** file and adding the `ExecutionTarget` property, which will give you design-time feedback on the compatibility of your program for Honeywell's hardware.

```
<Project Sdk="Microsoft.Quantum.Sdk/0.17.2105143879">
<PropertyGroup>
<OutputType>Exe</OutputType>
<TargetFramework>netcoreapp3.1</TargetFramework>
<ExecutionTarget>honeywell.hqs-lt-s1</ExecutionTarget>
</PropertyGroup>
</Project>
```

1. Replace the contents of `Program.qs` with the program:

```
namespace QuantumRNG {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Canon;

    @EntryPoint()
    operation GenerateRandomBits() : Result[] {
        use qubits = Qubit[4];
        ApplyToEach(H, qubits);
        return MultiM(qubits);
    }
}
```

## NOTE

If you would like to learn more about this program code, we recommend checking out how to [Create your first Q# program by using the Quantum Development Kit](#).

# Prepare the AZ CLI

Next, we'll prepare your environment to run the program against the workspace you created.

1. Log in to Azure using your credentials. You'll get a list of subscriptions associated with your account.

```
az login
```

2. Specify the subscription you want to use from those associated with your Azure account. You can also

find your subscription ID in the overview of your workspace in Azure Portal.

```
az account set -s <Your subscription ID>
```

3. Use `quantum workspace set` to select the workspace you created above as the default Workspace. Note that you also need to specify the resource group and the location you created it in:

```
az quantum workspace set -g MyResourceGroup -w MyWorkspace -l MyLocation -o table
```

Location	Name	ProvisioningState	ResourceGroup	StorageAccount	Usable
MyLocation	MyWorkspace	Succeeded	MyResourceGroup	/subscriptions/...	Yes

4. In your workspace, there are different targets available from the providers that you added when you created the workspace. You can display a list of all the available targets with the command

```
az quantum target list -o table
```

```
az quantum target list -o table
```

Provider	Target-id	Current Availability	Average Queue Time
-			
honeywell	honeywell.hqs-lt-s1	Available	0
honeywell	honeywell.hqs-lt-s1-apival	Available	0

#### NOTE

When you submit a job in Azure Quantum it will wait in a queue until the provider is ready to run your program.

The **Average Queue Time** column of the target list command shows you how many seconds recently run jobs waited in the queue. This can give you an idea of how long you might have to wait.

## Run the program on hardware

To run the program on hardware, we'll use the asynchronous job submission command `az quantum job submit`. Like the `execute` command this will compile and submit your program, but it won't wait until the execution is complete. We recommend this pattern for running against hardware, because you may need to wait a while for your job to finish. To get an idea of how long you can run `az quantum target list -o table` as described above.

```
az quantum job submit --target-id honeywell.hqs-lt-s1 -o table
```

Name	Id	Status	Target	Submission time
QuantumRNG	b4d17c63-2119-4d92-91d9-c18d1a07e08f	Waiting	honeywell.hqs-lt-s1	2020-01-12T22:41:27.8855301+00:00

The table above shows that your job has been submitted and is waiting for its turn to run. To check on the status, use the `az quantum job show` command, being sure to replace the `job-id` parameter with the Id output by the previous command:

```
az quantum job show -o table --job-id b4d17c63-2119-4d92-91d9-c18d1a07e08f
```

Name	Id	Status	Target	Submission time
QuantumRNG	b4d17c63-2119-4d92-91d9-c18d1a07e08f	Waiting	honeywell.hqs-lt-s1	2020-10-22T22:41:27.8855301+00:00

Eventually, you will see the `Status` in the above table change to `Succeeded`. Once that's done you can get the results from the job by running `az quantum job output`:

```
az quantum job output -o table --job-id b4d17c63-2119-4d92-91d9-c18d1a07e08f
```

Result	Frequency
[0,0,0,0]	0.05200000
[1,0,0,0]	0.07200000
[0,1,0,0]	0.05000000
[1,1,0,0]	0.06800000
[0,0,1,0]	0.04600000
[1,0,1,0]	0.06000000
[0,1,1,0]	0.06400000
[1,1,1,0]	0.07600000
[0,0,0,1]	0.04800000
[1,0,0,1]	0.06200000
[0,1,0,1]	0.07400000
[1,1,0,1]	0.08000000
[0,0,1,1]	0.05800000
[1,0,1,1]	0.06800000
[0,1,1,1]	0.05200000
[1,1,1,1]	0.07000000

The histogram you receive may be slightly different than the one above, but you should find that the states generally are observed with equal frequency.

#### NOTE

If you run into an error while working with Azure Quantum, you can check our [list of common issues](#).

## Next steps

This quickstart guide demonstrated how to get started running Q# programs against Honeywell's simulator and QPU. For more information on Honeywell's offerings, please see the [Honeywell Provider](#) documentation.

We recommend you continue your journey by learning more about the [different types of targets in Azure Quantum](#), which will dictate which Q# programs you may run against a given provider. You might also be interested in learning how to submit Q# jobs with [Jupyter Notebooks](#) or with [Python](#).

Looking for more samples to run? Check out the [samples directory](#) for Azure Quantum.

Lastly, if you would like to learn more about writing Q# programs please see the [Microsoft Quantum Documentation](#).

# Create and run Q# applications in Azure Quantum

6/17/2021 • 4 minutes to read • [Edit Online](#)

This guide outlines the process to create a Q# application and run it on the different quantum computing targets available in Azure Quantum. To see what types of quantum computing targets Azure Quantum offers, you can check our article [Targets in Azure Quantum](#).

## Create and run applications for Full profile targets

Full profile targets can run any Q# program, meaning you can write programs without functionality restrictions. Azure Quantum does not provide any target with this profile yet, but you can try any Q# program locally using the [full state simulator](#) or the [resources estimator](#) from the QDK.

If you need help setting up your environment to run Q# programs locally, see [Getting started with the QDK](#).

You can also explore different [Q# code samples](#) to run locally with the QDK.

## Create and run applications for No Control Flow profile targets

No Control Flow profile targets can run a wide variety of Q# applications, with the constraint that they can't use results from qubit measurements to control the program flow. More specifically, values of type `Result` do not support equality comparison.

For example, this operation can NOT be run on a No Control Flow target:

```
operation SetQubitState(desired : Result, q : Qubit) : Result {
    if (desired != M(q)) {
        X(q);
    }
}
```

Trying to run this operation on a No Control Flow target will fail because it evaluates a comparison between two results (`desired != M(q)`) to control the computation flow with an `if` statement.

### NOTE

Currently, there is an additional limitation for this type of profile target: *you can't apply operations on qubits that have been measured, even if you don't use the results to control the program flow*. This limitation is not inherent to this profile type but is circumstantial to the situation of the Public Preview.

Presently, these No Control Flow targets are available for Azure Quantum:

- **Provider:** IonQ
  - IonQ simulator (`ionq.simulator`)
  - IonQ QPU: (`ionq.qpu`)

## Create and run applications for Basic Measurement Feedback targets

Basic Measurement Feedback profile targets can run a wide variety of Q# applications, with the constraint that you can only compare values of type `Result` as part of conditions within `if` statements in operations. The corresponding conditional blocks may not contain `return` or `set` statements. This profile type supposes an

improvement over No Control Flow profiles, but still is subject to some limitations.

Presently, these Basic Measurement Feedback targets are available for Azure Quantum:

- **Provider:** Honeywell
  - Honeywell System Model H0 (`honeywell.hqs-lt-1.0`)
  - Honeywell System Model H1 (`honeywell.hqs-lt-s1`)

## Step-by-step guide to creating applications for hardware targets

Follow the steps in this section to create an application to run on available quantum computing targets.

### Prerequisites

- Install the [QDK](#).
- An Azure Quantum workspace with an appropriate provider subscription for your selected target. To create a workspace, see [Create an Azure Quantum workspace](#).

### Steps

1. [Create a Q# application using the Q# project template](#).
2. Open the `*.csproj` file in a text editor (for example, VS Code) and edit the file to:
  - Make sure the project points to the latest version of the QDK. You can verify the latest version in the official [QDK Release Notes](#).
  - Add a line specifying the preferred target inside the `<PropertyGroup>`. For example by picking one of the targets below:
    - IonQ simulator: `<ExecutionTarget>ionq.simulator</ExecutionTarget>`
    - IonQ QPU: `<ExecutionTarget>ionq.qpu</ExecutionTarget>`
    - Honeywell API validator: `<ExecutionTarget>honeywell.hqs-lt-1.0-apival</ExecutionTarget>`
    - Honeywell system model H0: `<ExecutionTarget>honeywell.hqs-lt-1.0</ExecutionTarget>`
    - Honeywell system model H1: `<ExecutionTarget>honeywell.hqs-lt-s1</ExecutionTarget>`

Your `*.csproj` file should look something like this:

```
<Project Sdk="Microsoft.Quantum.Sdk/X.XX.XXXX.XXXXXX">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <ExecutionTarget>my.target</ExecutionTarget>
  </PropertyGroup>

</Project>
```

where `X.XX.XXXX.XXXXXX` is a place holder for the latest version number of the QDK, and `my.target` a placeholder for your chosen target.

3. Write your Q# program, keeping in mind the restrictions applying to the computation profile of your particular target. You cannot compare measurement results to control the program flow.
4. Build and run your program locally using the targets supplied by the QDK. This will let you know if your Q# application can run on the specified Azure Quantum target by checking the fulfillment of the computation profile restrictions and calculating the needed resources.
  - You can run your Q# program locally using the QDK full state simulator by using the command `dotnet run`. Since you selected the `ExecutionTarget` in the `*.csproj` file, the console output will warn you if you created a file that is not compatible with the selected computation profile.

- You can use the [resources estimator](#) to estimate what resources your Q# program requires to run. You invoke the resources estimator with the command: `dotnet run --simulator ResourcesEstimator`.
5. Once your Q# program is ready, submit the job to Azure Quantum using your preferred environment by specifying the target ID, for example `ionq.qpu` or `honeywell.hqs-lt-1.0`.

For more information on how to submit jobs to Azure Quantum, see:

- [Submit jobs to Azure Quantum using the Azure CLI.](#)
- [Submit jobs to Azure Quantum using Python.](#)
- [Submit jobs to Azure Quantum using Q# Jupyter Notebooks](#)

The full list of targets can be found on the [Azure Quantum providers page](#).

**NOTE**

If you run into an error while working with Azure Quantum, you can check our [list of common issues](#).

## Next steps

- Now that you know how to create Q# applications, you can learn more details about [how to submit jobs to Azure Quantum](#).
- You can also try the different [samples](#) we have available or try to submit your own projects.

# Submit jobs to Azure Quantum with the command-line interface

5/27/2021 • 4 minutes to read • [Edit Online](#)

This guide shows you how to submit jobs to Azure Quantum using the command-line interface (CLI).

## Prerequisites

Ensure that the following items are installed on your computer:

- An Azure Quantum workspace in your Azure subscription. To create a workspace, see [Create an Azure Quantum workspace](#).
- The latest version of the [Quantum Development Kit](#).
- The [Azure CLI](#).
- The [necessary utilities to use Azure Quantum](#) (includes the `quantum` extension for the Azure CLI).

## Submit a job to Azure Quantum with the Azure CLI

These steps show how to use the Azure CLI to run a Q# application and select a target from the different providers of your Azure Quantum workspace.

### NOTE

A provider is a partner quantum service consisting of quantum hardware, a simulator, or an optimization service.

1. Log in to Azure using your credentials. You'll get a list of subscriptions associated with your account.

```
az login
```

2. Specify the subscription you want to use from those associated with your Azure account. You can also find your subscription ID in the overview of your workspace in Azure Portal.

```
az account set -s <Your subscription ID>
```

3. You can see all the Azure Quantum workspaces in your subscription with the following command:

```
az quantum workspace list
```

4. You can use `quantum workspace set` to select a default workspace that you want to use to list and submit jobs. Note that you also need to specify the resource group and the location:

```
az quantum workspace set -g MyResourceGroup -w MyWorkspace -l MyLocation -o table
```

Location	Name	ResourceGroup
MyLocation	ws-yyyyyyy	rg-yyyyyyyyyy

### TIP

You can check the current workspace with the command `az quantum workspace show -o table`.

1. In your Azure Quantum workspace, there are different targets available from the providers that you added when you created the workspace. You can display a list of all the available targets with the command `az quantum target list -o table`:

```
az quantum target list -o table
```

Provider	Target-id	Status	Average Queue Time
Microsoft	microsoft.paralleltempering-parameterfree.cpu	Available	0
Microsoft	microsoft.paralleltempering.cpu	Available	0
Microsoft	microsoft.simulatedannealing-parameterfree.cpu	Available	0
Microsoft	microsoft.simulatedannealing.cpu	Available	0
Microsoft	microsoft.paralleltempering.fpga	Available	0
Microsoft	microsoft.simulatedannealing.fpga	Available	0
ionq	ionq.qpu	Available	0
ionq	ionq.simulator	Available	0

2. To submit a new job, navigate to the directory containing your project using the command line and submit your job. Use the command `az quantum job submit`. You need to specify the target where you want to run your job, for example:

```
az quantum job submit --target-id MyProvider.MyTarget
```

Id	State	Target	Submission time
yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyy	Waiting	MyProvider.MyTarget	2020-06-12T14:20:18.6109317+00:00

The console will output the job information, including the job ID.

3. You can use the job ID to track its status:

```
az quantum job show -j yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyy -o table
```

Id	State	Target	Submission time
-- yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyy	Succeeded	MyProvider.MyTarget	2020-06-12T14:20:19.819981+00:00

**TIP**

To see all the jobs in the workspace, use the command `az quantum job list -o table`.

- Once the job finishes, display the job's results with the command `az quantum job output`:

```
az quantum job output -j yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyy -o table
```

Result	Frequency	
[0,0]	0.0000000	
[1,0]	0.5000000	
[0,1]	0.2500000	
[1,1]	0.2500000	

**TIP**

To submit a job synchronously, for example, waiting for the job to complete and showing results, use the command

```
az quantum execute --target-id MyProvider.MyTarget .
```

## Example

### Write your quantum application

First, you need the Q# quantum application that you want to run in Azure Quantum.

**TIP**

If this is your first time creating Q# quantum applications, see our [Microsoft Learn module](#).

In this case, we will use a simple quantum random bit generator. Create a Q# project and substitute the content of `Program.qs` with the following code:

```
namespace RandomBit {

    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Measurement;

    @EntryPoint()
    operation GenerateRandomBit() : Result {
        use q = Qubit();
        H(q);
        return MResetZ(q);
    }
}
```

Note that the `@EntryPoint` attribute tells Q# which operation to run when the program starts.

### Submit the job

In this example, we are going to use IonQ as the provider and the `ionq.simulator` as the target. To submit the job to the currently selected default workspace, use the command `az quantum job submit`:

## IMPORTANT

Verify that the Quantum SDK version of the `*.csproj` file is `0.11.2006.403` or higher. If not, it could cause a compilation error.

```
az quantum job submit --target-id ionq.simulator --job-name ExampleJob -o table
```

Name	Id	Status	Target	Submission time
ExampleJob	yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyyyy	Waiting	ionq.simulator	2020-06-17T17:07:07.3484901+00:00

Once the job completes (that is, when it's in a **Successful** state), use the command `az quantum job output` to view the results:

```
az quantum job output -j yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyyyy -o table
```

Result	Frequency
[0,0]	0.50000000
[0,1]	0.50000000

The output displays a histogram with the frequency that a specific result was measured. In the example above, the result `[0,1]` was observed 50% of the times.

Optionally, you can use the command `az quantum execute` as a shortcut for both submitting and returning the results of a run.

```
az quantum execute --target-id ionq.simulator --job-name ExampleJob2 -o table
```

Result	Frequency
[0,0]	0.50000000
[0,1]	0.50000000

Note that the IonQ simulator gives the probabilities of obtaining each output if we run the algorithm an infinite number of times. In this case, we see that each state has a 50% probability of being measured.

## TIP

You can also check the status of your jobs from your Azure portal.

## Next steps

Now that you know how to submit jobs to Azure quantum, you can try to run the different [samples](#) we have available or try to submit your own projects.

# Submit jobs to Azure Quantum with Q# Jupyter Notebooks

6/25/2021 • 3 minutes to read • [Edit Online](#)

This document provides a basic guide to submit and run Q# applications in Azure Quantum using Q# Jupyter Notebooks.

## Prerequisites

- An Azure Quantum workspace in your Azure subscription. To create a workspace, see [Create an Azure Quantum workspace](#).
- The latest version of the [Quantum Development Kit for Jupyter Notebooks](#).

Follow the installation steps in the provided link to install Jupyter Notebook and the current version of the IQ# kernel, which powers the Q# Jupyter Notebook and Python experiences.

## Quantum computing with Q# Jupyter Notebooks

1. Run `jupyter notebook` from the terminal where your conda environment is activated. This starts the notebook server and opens Jupyter in a browser.
2. Create your Q# notebook (via **New** → **Q#**) and write your Q# program.
3. If you've never used Q# with Jupyter, follow the steps in [Create your first Q# notebook](#).
4. Write your Q# operations directly in the notebook. Running the cells will compile the Q# code and report whether there are any errors.
  - For example, you could write a Q# operation that looks like this:

```
operation GenerateRandomBit() : Result {
    use q = Qubit();
    H(q);
    let r = M(q);
    Reset(q);
    return r;
}
```

5. Once you have your Q# operations defined, use the `%azure.*` magic commands to connect and submit jobs to Azure Quantum. You'll use the resource ID of your Azure Quantum workspace in order to connect. (The resource ID can be found on your workspace page in the Azure Portal.)

If your workspace was created in an Azure region other than "West US", you also need to specify this as the `location` parameter to `%azure.connect`.

- For example, the following commands will connect to an Azure Quantum workspace and run an operation on the `ionq.simulator` target:

```
%azure.connect "/subscriptions/.../Microsoft.Quantum/Workspaces/WORKSPACE_NAME" location="West US"

%azure.target ionq.simulator

%azure.execute GenerateRandomBit
```

where `GenerateRandomBit` is the Q# operation that you have already defined in the notebook.

- After submitting a job, you can check its status with the command `%azure.status` or view its results with the command `%azure.output`. You can view a list of all your jobs with the command `%azure.jobs`.

## Authentication

The `%azure.connect` magic takes an optional `credential` parameter that allows you to specify what type of credentials to use to authenticate with Azure. The possible values for this parameter are:

- **Environment**: Authenticates a service principal or user via credential information specified in environment variables. For information about the specific environment variable needed see the [EnvironmentCredential online documentation](#)
- **ManagedIdentity**: Authenticates the managed identity of an Azure resource.
- **CLI**: Authenticates in a development environment using data from the Azure CLI.
- **SharedToken**: Authenticates in a development environment using tokens in the local cache shared between Microsoft applications.
- **VisualStudio**: Authenticates in a development environment using data from Visual Studio.
- **VisualStudioCode**: Authenticates in a development environment using data from Visual Studio Code.
- **Interactive**: Opens a new browser window to interactively authenticate and obtain an access token.
- **DeviceCode**: Authenticates using the device code flow.

If the `credential` parameter is not provided, then by default all the different mechanisms listed above are tried in order until one succeeds.

The following example shows how to use the `credential` parameter to explicitly open a new browser window to login to Azure:

```
%azure.connect
/subscriptions/SUBSCRIPTION_ID/resourceGroups/RESOURCE_GROUP/providers/Microsoft.Quantum/Workspaces/WORKSPACE
location=LOCATION
credential=interactive
```

## Getting inline help

Some helpful tips while using Q# Jupyter Notebooks:

- Use the command `%lsmagic` to see all of the available magic commands, including the ones for Azure Quantum.
- Detailed usage information for any magic command can be displayed by simply appending a `?` to the command, for example, `%azure.connect?`.
- Documentation for magic commands is also available online: [%azure.connect](#), [%azure.target](#), [%azure.submit](#), [%azure.execute](#), [%azure.status](#), [%azure.output](#), [%azure.jobs](#)

## Next steps

Now that you know how to submit jobs to Azure Quantum, you can try to run the different [samples](#) we have available or try to submit your own projects. In particular, you can view a sample written entirely in a Q# Jupyter notebook.

# Submit jobs to Azure Quantum with Python

6/25/2021 • 3 minutes to read • [Edit Online](#)

This document provides a basic guide to submit and run Q# applications in Azure Quantum using Python.

## Prerequisites

- An Azure Quantum workspace in your Azure subscription. To create a workspace, see [Create an Azure Quantum workspace](#).
- The latest version of the [Quantum Development Kit for Python](#).

Follow the installation steps in the provided link to install Python and the current version of the IQ# kernel, which powers the Q# Jupyter Notebook and Python experiences.

## Quantum computing with Q# and Python

1. The Python environment in the conda environment that you created earlier already includes the `qsharp` Python package. Make sure you are running your Python script from a terminal where this conda environment is activated.
2. If you've never used Q# with Python, follow the steps in [Create your first Q# program with Python](#).
3. Write your Q# operations in a `*.qs` file. When running `import qsharp` in Python, the IQ# kernel will automatically detect any .qs files in the same folder, compile them, and report any errors. If compilation is successful, those Q# operations will become available for use directly from within Python.
  - For example, the contents of your .qs file could look something like this:

```
namespace Test {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Canon;

    operation GenerateRandomBits(n : Int) : Result[] {
        use qubits = Qubit[n];
        ApplyToEach(H, qubits);
        return MultiM(qubits);
    }
}
```

4. Create a Python script in the same folder as your `*.qs` file. Azure Quantum functionality is available by running `import qsharp.azure` and then calling the Python commands to interact with Azure Quantum. For reference, see the [complete list of `qsharp.azure` Python commands](#). You'll need the resource ID of your Azure Quantum workspace in order to connect. (The resource ID can be found on your workspace page in the Azure Portal.)

If your workspace was created in an Azure region other than "West US", you also need to specify this as the `location` parameter to `qsharp.azure.connect()`.

For example, your Python script could look like this:

```

import qsharp
import qsharp.azure
from Test import GenerateRandomBits

qsharp.azure.connect(
    resourceId="/subscriptions/.../Microsoft.Quantum/Workspaces/WORKSPACE_NAME",
    location="West US")
qsharp.azure.target("ionq.simulator")
result = qsharp.azure.execute(GenerateRandomBits, n=3, shots=1000, jobName="Generate three random
bits")
print(result)

```

where `GenerateRandomBits` is the Q# operation in a namespace `Test` that is defined in your `*.qs` file, `n=3` is the parameter to be passed to that operation, `shots=1000` (optional) specifies the number of repetitions to perform, and `jobName="Generate three random bits"` (optional) is a custom job name to identify the job in the Azure Quantum workspace.

- Run your Python script by running the command `python test.py`, where `test.py` is the name of your Python file. If successful, you should see your job results displayed to the terminal. For example:

```
{'[0,0,0]': 0.125, '[1,0,0]': 0.125, '[0,1,0]': 0.125, '[1,1,0]': 0.125, '[0,0,1]': 0.125, '[1,0,1]':
0.125, '[0,1,1]': 0.125, '[1,1,1]': 0.125}
```

- To view the details of all jobs in your Azure Quantum workspace, run the command `qsharp.azure.jobs()`:

```

>>> qsharp.azure.jobs()
[{'id': 'f4781db6-c41b-4402-8d7c-5cfce7f3cde4', 'name': 'GenerateRandomNumber 3 qubits', 'status':
'Succeeded', 'provider': 'ionq', 'target': 'ionq.simulator', 'creation_time': '2020-07-
17T21:45:43.4405253Z', 'begin_execution_time': '2020-07-17T21:45:54.09Z', 'end_execution_time':
'2020-07-17T21:45:54.101Z'}, {'id': '1b03cc74-b5d5-4ffa-81db-465f08ae6cd0', 'name':
'GenerateRandomBit', 'status': 'Succeeded', 'provider': 'ionq', 'target': 'ionq.simulator',
'creation_time': '2020-07-21T19:44:17.1065156Z', 'begin_execution_time': '2020-07-21T19:44:25.85Z',
'end_execution_time': '2020-07-21T19:44:25.858Z'}]

```

- To view the detailed status of a particular job, pass the job ID to `qsharp.azure.status()` or `qsharp.azure.output()`, for example:

```

>>> qsharp.azure.status('1b03cc74-b5d5-4ffa-81db-465f08ae6cd0')
{'id': '1b03cc74-b5d5-4ffa-81db-465f08ae6cd0', 'name': 'GenerateRandomBit', 'status': 'Succeeded',
'provider': 'ionq', 'target': 'ionq.simulator',
'creation_time': '2020-07-21T19:44:17.1065156Z', 'begin_execution_time': '2020-07-21T19:44:25.85Z',
'end_execution_time': '2020-07-21T19:44:25.858Z'}

>>> qsharp.azure.output('1b03cc74-b5d5-4ffa-81db-465f08ae6cd0')
{'0': 0.5, '1': 0.5}

```

## Authentication

The `qsharp.azure.connect` method takes an optional `credential` parameter that allows you to specify what type of credentials to use to authenticate with Azure. The possible values for this parameter are:

- Environment**: Authenticates a service principal or user via credential information specified in environment variables. For information about the specific environment variable needed see the [EnvironmentCredential online documentation](#).
- ManagedIdentity**: Authenticates the managed identity of an Azure resource.
- CLI**: Authenticates in a development environment using data from the Azure CLI.

- **SharedToken**: Authenticates in a development environment using tokens in the local cache shared between Microsoft applications.
- **VisualStudio**: Authenticates in a development environment using data from Visual Studio.
- **VisualStudioCode**: Authenticates in a development environment using data from Visual Studio Code.
- **Interactive**: Opens a new browser window to interactively authenticate and obtain an access token.
- **DeviceCode**: Authenticates using the device code flow.

If the `credential` parameter is not provided, then by default all the different mechanisms listed above are tried in order until one succeeds.

The following example shows how to use the `credential` parameter to explicitly open a new browser window to login to Azure:

```
import qsharp
import qsharp.azure
from Test import GenerateRandomBits

qsharp.azure.connect(
    resourceId="/subscriptions/.../Microsoft.Quantum/Workspaces/WORKSPACE_NAME",
    location="West US",
    credential="interactive")
```

## Next steps

Now that you know how to submit jobs to Azure Quantum, you can try to run the different [samples](#) we have available or try to submit your own projects.

# List of quantum computing targets on Azure Quantum

4/29/2021 • 2 minutes to read • [Edit Online](#)

Azure Quantum also offers a variety of quantum solutions, such as different hardware devices and quantum simulators that you can use to run Q# quantum computing programs.

## Provider: IonQ



### **IonQ Quantum Simulator**

GPU-accelerated idealized simulator supporting up to 29 qubits, using the same set of gates IonQ provide on its quantum hardware—a great place to preflight jobs before running them on an actual quantum computer. For more information, go to the [IonQ provider reference page](#).

### **IonQ Quantum Computer**

Trapped ion quantum computer. Dynamically reconfigurable in software to use up to 11 qubits. All qubits are fully connected, meaning you can run a two-qubit gate between any pair. For more information, go to the [IonQ provider reference page](#).

## Provider: Honeywell



### **API Validator**

Tool to verify proper syntax and compilation completion. Full stack is exercised with the exception of the actual quantum operations. Assuming no bugs, all zeros are returned in the proper data structure. For more information, go to the [Honeywell provider reference page](#).

### **Honeywell System Model H0**

Trapped ion quantum computer with 6 physical fully connected qubits and laser based gates. It uses a QCDD architecture with linear trap and two parallel operation zones. For more information, go to the [Honeywell provider reference page](#).

### **Honeywell System Model H1**

Trapped ion quantum computer with 10 physical fully connected qubits and laser based gates. It uses a QCDD architecture with linear trap and two parallel operation zones. For more information, go to the [Honeywell provider reference page](#)

# IonQ provider

6/2/2021 • 2 minutes to read • [Edit Online](#)

IonQ's quantum computers perform calculations by manipulating the hyperfine energy states of Ytterbium ions with lasers. Atoms are nature's qubits — every qubit is identical within and between programs. Logical operations can also be performed on any arbitrary pair of qubits, enabling complex quantum programs unhindered by physical connectivity. Want to learn more? Read IonQ's [trapped ion quantum computer technology overview](#).

- Publisher: [IonQ](#)
- Provider ID: `ionq`

## Targets

The IonQ provider makes the following targets available:

- [IonQ provider](#)
  - [Targets](#)
  - [IonQ Quantum Simulator](#)
  - [IonQ Quantum Computer](#)

## Quantum Simulator

GPU-accelerated idealized simulator supporting up to 29 qubits, using the same set of gates IonQ provide on its quantum hardware—a great place to preflight jobs before running them on an actual quantum computer.

- Job type: `Simulation`
- Data Format: `ionq.circuit.v1`
- Target ID: `ionq.simulator`
- Q# Profile: `No Control Flow`

## Quantum Computer

Trapped ion quantum computer. Dynamically reconfigurable in software to use up to 11 qubits. All qubits are fully connected, meaning you can run a two-qubit gate between any pair.

- Job type: `Quantum Program`
- Data Format: `ionq.circuit.v1`
- Target ID: `ionq.qpu`
- Q# Profile: `No Control Flow`

PARAMETER NAME	TYPE	REQUIRED	DESCRIPTION
<code>shots</code>	int	No	Number of experimental shots. Defaults to 500.

## System timing

MEASURE	AVERAGE TIME DURATION (MS)
T1	>10^7
T2	200,000
Single-qubit gate	10
Two-qubit gate	210
Readout	100
Register reset	25
Coherence time / gate duration	1667

## System fidelity

OPERATION	AVERAGE FIDELITY
Single-qubit gate	99.35% (SPAM corrected)
Two-qubit gate	96.02% (not SPAM corrected)
SPAM	99.3 - 99.8%
Geometric mean op	98.34%

## Pricing

IonQ charges per **gate-shot**: the number of gates in your circuit, multiplied by the number of shots.

Multi-controlled two-qubit gates are billed as  $6 * (N - 2)$  two-qubit gates, where N is the number of qubits involved in the gate. For example, a NOT gate with three controls would be billed as  $(6 * (4 - 2))$  or 12 two-qubit gates.

To see the pricing options:

1. Go to the Azure Portal and create a new workspace.
2. In the **Providers** pane, click in the **Add** button of the IonQ tile and in the description you will find the current pricing options.

### IMPORTANT

Note that there is a \$1 USD minimum cost to run a job on the IonQ QPU.

## IonQ Best Practices & Connectivity Graph

To see recommended best practices for the IonQ QPU, we recommend reading their [best practices](#).

# Support Policy for IonQ in Azure Quantum

5/27/2021 • 2 minutes to read • [Edit Online](#)

This article describes the Microsoft support policy that applies when you use the IonQ provider in Azure Quantum. The article applies to any of the targets under this provider.

If you are using the IonQ provider and hit any unexpected issues that you cannot troubleshoot yourself, you can usually contact the Azure Support team for help by [creating an Azure support case](#).

There are however some situations, where the Azure Support team will need to redirect you to IonQ's support team, or where you may receive a quicker response by reaching out to IonQ directly. This article aims to provide more detail on this based on some frequently asked questions.

## Frequently asked questions

### **Q: What is the support policy for using IonQ offerings through Azure Quantum?**

Microsoft will provide support for the Azure Platform and the Azure Quantum service only. Support for IonQ hardware, simulators, and other products and targets will be provided directly by IonQ. For more information about Azure support, [see Azure support plans](#). For information about IonQ support, please [visit the IonQ Support website](#).

### **Q: What happens if I raise a support issue with the Azure support team and it turns out that a third party provider (like IonQ) needs to troubleshoot the issue further?**

The support engineer will create a redirection package for you. This is a PDF document that contains information about your case which you can provide to the IonQ support team. The support engineer will also give you advice and guidance on how to reach out to IonQ to continue troubleshooting.

### **Q: What happens if I raise a support issue with the IonQ team and it turns out that there is an issue with the Azure Quantum service?**

The IonQ support team will help you reach out to Microsoft and provide you with a redirection package. This is a PDF document that you can use when continuing your support enquiry with the Azure support team.

### **Third-party information disclaimer**

The third-party products that this article discusses are manufactured by companies that are independent of Microsoft. Microsoft makes no warranty, implied or otherwise, about the performance or reliability of these products.

### **Third-party contact disclaimer**

Microsoft provides third-party contact information to help you find additional information about this topic. This contact information may change without notice. Microsoft does not guarantee the accuracy of third-party contact information.

# Honeywell provider

7/9/2021 • 2 minutes to read • [Edit Online](#)

- Publisher: [Honeywell](#)
- Provider ID: `honeywell`

## Targets

The following targets are available from this provider:

- [API Validator](#)
- [Honeywell System Model H1](#)

### Target Availability

A target's status indicates its current ability to process jobs. The possible states of a target include:

- **Available:** The target is processing jobs at a normal rate.
- **Degraded:** The target is currently processing jobs at a slower rate than usual.
- **Unavailable:** The target currently does not process jobs.

Current status information may be retrieved from the *Providers* tab of a workspace on the [Azure Portal](#).

### API Validator

Tool to verify proper syntax and compilation completion. Full stack is exercised with the exception of the actual quantum operations. Assuming no bugs, all zeros are returned in the proper data structure.

- Job type: `Simulation`
- Data Format: `honeywell.openqasm.v1`
- Target ID: `honeywell.hqs-lt-1.0-apival`
- Target Execution Profile: [Basic Measurement Feedback](#)

Billing information: No charge for usage.

### Honeywell System Model H1

Honeywell Quantum Solutions' Quantum Computer, System Model H1

- Job type: `Quantum Program`
- Data Format: `honeywell.openqasm.v1`
- Target ID: `honeywell.hqs-lt-s1`
- Target Execution Profile: [Basic Measurement Feedback](#)

Billing information:

- **Standard Subscription:** Monthly subscription plan with 10K Honeywell quantum credits (HQCs) / month, available through queue.
- **Premium Subscription:** Monthly subscription plan with 17K Honeywell quantum credits (HQCs) / month, available through queue.

The following equation defines how circuits are translated into Honeywell Quantum Credits (HQCs):

$$\$ \text{ HQC} = 5 + C(N_{\{1q\}} + 10 N_{\{2q\}} + 5 N_m)/5000 \$$$

where:

- $N_{\{1q\}}$  is the number of one-qubit operations in a circuit.
- $N_{\{2q\}}$  is the number of native two-qubit operations in a circuit. Native gate is equivalent to CNOT up to several one-qubit gates.
- $N_{\{m\}}$  is the number of state preparation and measurement (SPAM) operations in a circuit including initial implicit state preparation and any intermediate and final measurements and state resets.
- $C$  is the shot count.

#### **Technical Specifications**

- Trapped-ion based quantum computer with laser based gates
- QCCD architecture with linear trap and three parallel operational zones
- 10 physical qubits, fully connected
- Typical limiting fidelity >99.5% (two-qubit fidelity)
- Coherence Time ( $T_2$ ) ~3 sec
- Ability to perform mid-circuit measurement and qubit reuse
- High-resolution rotations ( $> \pi/500$ )
- Native Gate set:
  - single-qubit rotations
  - two-qubit ZZ-gates

More details available under NDA.

# Support Policy for Honeywell in Azure Quantum

5/27/2021 • 5 minutes to read • [Edit Online](#)

This article describes the Microsoft support policy that applies when you use the Honeywell provider in Azure Quantum. The article applies to any of the targets under this provider.

If you are using the Honeywell provider and hit any unexpected issues that you cannot troubleshoot yourself, you can usually contact the Azure Support team for help by [creating an Azure support case](#).

There are however some situations, where the Azure Support team will need to redirect you to Honeywell's support team, or where you may receive a quicker response by reaching out to Honeywell directly. This article aims to provide more detail on this based on some frequently asked questions.

## Honeywell Support Policy - Public Preview

The goal of Support is to identify and remedy defects or malfunctions causing the Quantum Computer to fail to perform in accordance with the agreed specifications and documentation ("Problems"). Support only covers the current released version generally available to customers. Although our quantum experts can help you troubleshoot algorithm issues, we are not responsible for any issues, problems, or defects with your algorithm. To minimize programming issues, users are able to run their algorithms through the syntax validator prior to running on hardware.

### 1.1 CONTACTING TECHNICAL SUPPORT WITHIN HONEYWELL QUANTUM SOLUTIONS

To create a support request with Honeywell Quantum Solutions, submit an incident report via email to [HoneywellAzureQuantumSupport@Honeywell.com](mailto:HoneywellAzureQuantumSupport@Honeywell.com). The on-call engineer will respond within the appropriate SLA (Table 2). Note that only incident reports that contain all of the below information will notify the Honeywell Quantum Solutions team and have the appropriate SLA resolution window. Reports without the necessary information will not trigger this response. All requests must be provided in the English language and will be answered in the English language.

*Table 1: Support Contact information and Standard Operating Hours | Email | Standard Operating Hours || - | - || HoneywellAzureQuantumSupport@Honeywell.com | 8am – 5pm MST on Business Days<sup>1</sup> |*

<sup>1</sup> "Business Days" means Monday to Friday but excluding national, legal, or bank holidays.

The incident report will require the following information:

- Description: A detailed description of the issue you are facing, what you are observing, and any steps you have already taken to triage/understand the issue
- Primary contact: A primary contact name and phone number in case we require more information
- Incident Severity: the severity of the incident, according to our severity definitions

Once a report is filed, the Honeywell Quantum Solutions team will be notified and will respond within the required SLA window. You will receive an acknowledgement of the issue once it has been received and the on-call engineer has been notified. The on-call engineer may request more information, or may call you at the contact number listed in the report depending on the severity.

During Reserved (dedicated) Sessions, Honeywell Quantum Solutions will provide at least one quantum specialist available to you for customer support for the entirety of your dedicated session. Honeywell reserves the right to change out the specialist during your session, but will use commercially reasonable efforts to ensure the new specialist is prepared to continue providing support.

During Queued Runs, if your job is running in the general queue and you encounter an issue, you may submit your issue request online at: [HoneywellAzureQuantumSupport@Honeywell.com](mailto:HoneywellAzureQuantumSupport@Honeywell.com). You can expect to receive an answer within three (3) business days.

## 1.2 HONEYWELL QUANTUM SOLUTIONS SEVERITY INDEX AND RESPONSE SLA

If the Honeywell Quantum Solutions engineer determines that the issue does not fall within the bounds of our incident response commitments, or does not have the right severity classification, they may opt at their sole discretion to adjust the severity and/or end support if the new severity does not require it.

Certain problems may be easier to address than others, and we may not be able to completely resolve the problem with our initial response. If we are unable to reasonably resolve the problem, we will make a good faith effort to give an assessment of the issue and an estimated time for resolution.

*Table 2: Definition of Severity Index*

SEVERITY	MEANING	DESCRIPTION
Severity 1	Significant Impact	System performance degraded below minimum accepted level
Severity 2	Urgent or high impact	Job status has not updated; issues retrieving data
Severity 3	Not urgent	Miscellaneous

The table below shows the Response SLA that will be adhered to by Honeywell Quantum Solutions for incident reports of the corresponding severity. Note that these define how fast we will respond to issues – resolution times are not guaranteed.

*Table 3: Response SLA for Various Severity | Severity | Response SLA || - | - | Severity 1 | Business hours response within 1 hour | | Severity 2 | Business hours response within 1 business day | | Severity 3 | Business hours response within 3 business days |*

## Frequently asked questions

**Q: What happens if I raise a support issue with the Azure support team and it turns out that a third party provider (like Honeywell) needs to troubleshoot the issue further?**

The support engineer will create a redirection package for you. This is a PDF document that contains information about your case which you can provide to the Honeywell support team. The support engineer will also give you advice and guidance on how to reach out to Honeywell to continue troubleshooting.

**Q: What happens if I raise a support issue with the Honeywell team and it turns out that there is an issue with the Azure Quantum service?**

The Honeywell support team will help you reach out to Microsoft and provide you with a redirection package. This is a PDF document that you can use when continuing your support enquiry with the Azure support team.

### Third-party information disclaimer

The third-party products that this article discusses are manufactured by companies that are independent of Microsoft. Microsoft makes no warranty, implied or otherwise, about the performance or reliability of these products.

### Third-party contact disclaimer

Microsoft provides third-party contact information to help you find additional information about this topic. This contact information may change without notice. Microsoft does not guarantee the accuracy of third-party contact information.

# Microsoft QIO optimization quickstart for Azure Quantum

6/8/2021 • 8 minutes to read • [Edit Online](#)

Learn how to use Microsoft QIO in Azure Quantum to solve a simple binary optimization problem.

## Prerequisites

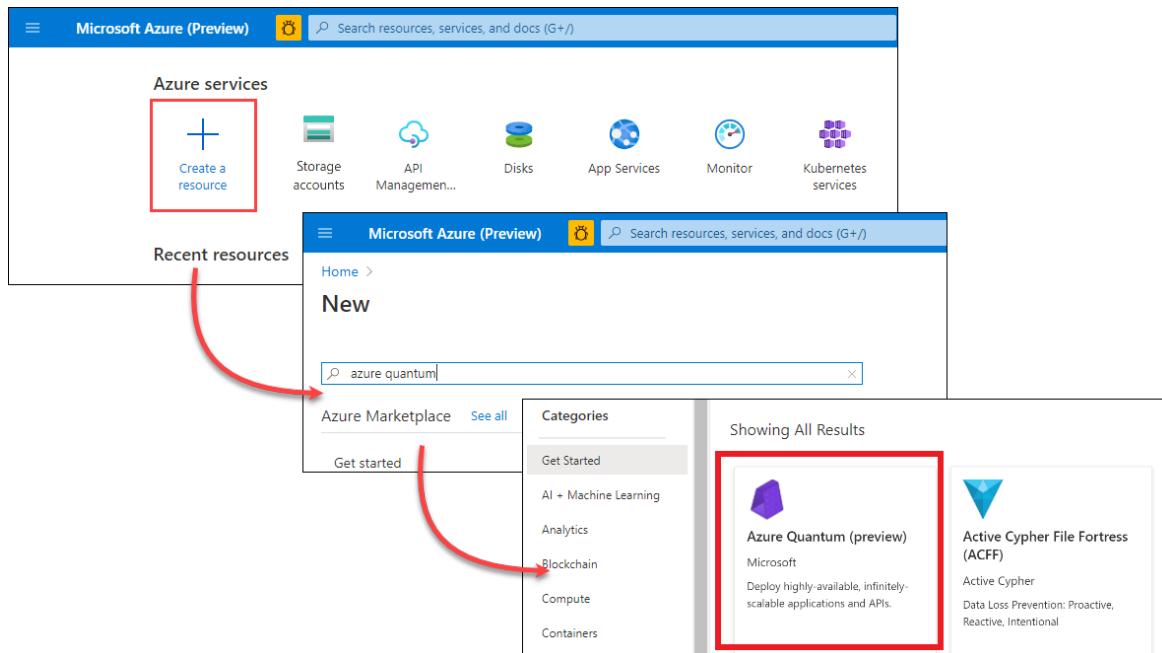
- To complete this tutorial you need an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.

## Create an Azure Quantum workspace

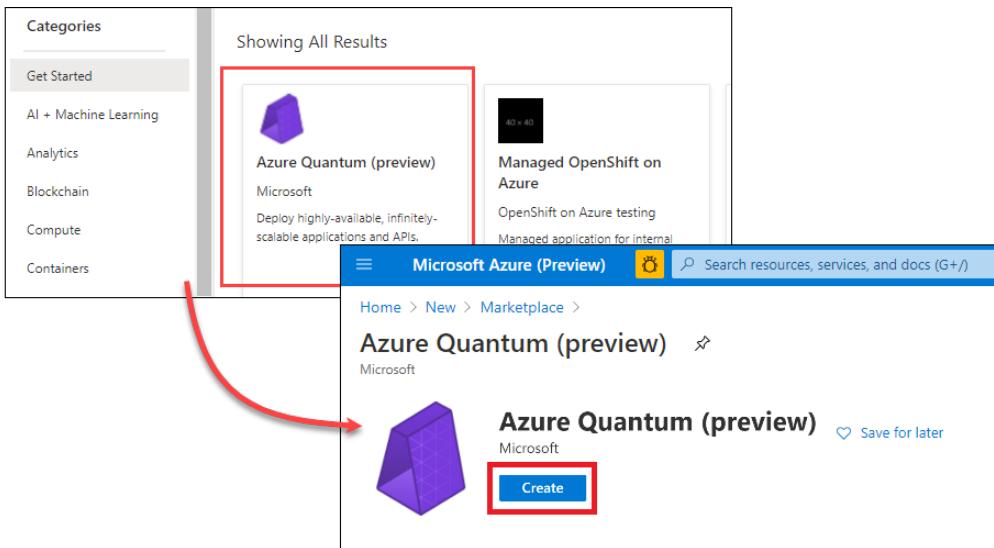
You use the Azure Quantum service by adding an Azure Quantum workspace resource to your Azure subscription in the Azure portal. An Azure Quantum workspace resource, or workspace for short, is a collection of assets associated with running quantum or optimization applications.

To open the Azure Portal, go to <https://portal.azure.com> and then follow these steps:

1. Click **Create a resource** and then search for **Azure Quantum**. On the results page, you should see a tile for the **Azure Quantum (preview)** service.



2. Click **Azure Quantum (preview)** and then click **Create**. This opens a form to create a workspace.



3. Fill out the details of your workspace:

- **Subscription:** The subscription that you want to associate with this workspace.
- **Resource group:** The resource group that you want to assign this workspace to.
- **Name:** The name of your workspace.
- **Region:** The region for the workspace.
- **Storage Account:** The Azure storage account to store your jobs and results. If you don't have an existing storage account, click **Create a new storage account** and complete the necessary fields. For this preview, we recommend using the default values.

**Project details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

<b>Subscription *</b> ⓘ Azure Quantum PM and Outreach	<b>Resource group *</b> ⓘ Documentation <a href="#">Create new</a>
<b>Instance details</b>	
<b>Name *</b> ⓘ bbQuantumWksp3	<b>Region *</b> ⓘ (US) West US
<b>Storage Account</b> ⓘ quantumdoc <a href="#">Create a new storage account</a>	

[Review + create](#)   [Previous](#)   [Providers >](#)

#### NOTE

You must be an Owner of the selected resource group to create a new storage account. For more information about how resource groups work in Azure, see [Control and organize Azure resources with Azure Resource Manager](#).

4. After completing the information, click the **Providers** tab to add providers to your workspace. A provider gives you access to a quantum service, which can be quantum hardware, a quantum simulator, or an optimization service.

5. Ensure the Microsoft QIO provider is enabled (it is by default), then click **Review + create**.
6. Review the setting you've selected and if everything is correct, click **Create** to create your workspace.

The screenshot shows the Microsoft Azure (Preview) interface for creating a Quantum Workspace. At the top, there's a navigation bar with 'Microsoft Azure (Preview)', a search bar, and a dashboard link. Below that, the 'Marketplace' and 'Azure Quantum (preview)' sections are visible. The main title is 'Create Quantum Workspace' under the 'Quantum Workspace' category. Below the title, there are tabs for 'Basics', 'Providers', 'Tags', and 'Review + create', with 'Review + create' being the active tab. A prominent red box highlights a warning message: '⚠️ Terms and conditions need to be accepted for the providers you selected. [Accept terms](#)'. Below this, a modal window titled 'Terms and conditions' appears. It contains a note: 'Please read and accept the terms and condition before enabling the following providers:' followed by a list of providers, with 'IonQ' currently expanded. A checkbox labeled 'I accept the terms and conditions.' is checked.

#### NOTE

Pricing for Azure Quantum varies by provider. Please consult the information in the Providers tab of your Azure Quantum workspace in the Azure portal for the most up-to-date pricing information, or visit the [Azure Quantum pricing page](#).

## Define your optimization problem

In this guide, you will solve a simple optimization example to get started with the optimization services of Azure Quantum. This quickstart is based on the [ship loading sample](#).

Suppose there are two ships ready to be loaded with containers and a list of containers of varying weights to be assigned to each ship. The aim of the optimization problem is to assign containers to each ship in such a way that the weight is distributed as evenly as possible between both ships.

The cost function for this optimization problem looks like the following:

```
$$ H^2 = \text{Large}(\sum_{i \in A \cup B} w_i x_i)^2 $$
```

This cost function has the following properties:

- If all the containers are on one ship, the function is at its highest value - reflecting that this is the least optimal solution
- If the containers are perfectly balanced, the value of the summation inside the square is  $\$0\$$  - the function is at its lowest value. This solution is optimal.

The goal is to find the configuration that yields the lowest possible value of  $H^2$ .

#### NOTE

For a detailed walkthrough of the problem scenario and how the cost function is constructed, please refer to the [sample](#) and/or the associated [Microsoft Learn module](#).

## Install the Python SDK for Azure Quantum

To implement a solution, first ensure that you have the Python SDK for Azure Quantum installed on your machine. If you don't have it installed yet, follow these steps:

1. Install [Python](#) 3.6 or later in case you haven't already.
2. Install [PIP](#) and ensure you have **version 19.2 or higher**.
3. Install the `azure-quantum` python package.

```
pip install --upgrade azure-quantum
```

## Create a `Workspace` object in your Python code and log in

Now create a Python file or Jupyter Notebook, import the `Workspace` module from `azure.quantum`, and create a `Workspace` object. This is what you will use to submit our optimization problem to Azure Quantum. The value for `resource_id` and `location` can be found on the Azure Portal page for the [workspace you created](#).

```
from azure.quantum import Workspace

# Copy the settings for your workspace below
workspace = Workspace(
    resource_id = "", # add the Resource ID of the Azure Quantum workspace you created
    location = ""     # add the location of your Azure Quantum workspace (e.g. "westus")
)
```

The first time you run a method which interacts with the Azure service, a window might prompt in your default browser asking for your credentials. You can optionally pass a credential to be used in the authentication in the construction of the `Workspace` object or via its `credentials` property. See more at [Azure.Quantum.Workspace](#)

### NOTE

The `workspace.login()` method has been deprecated and is no longer necessary. The first time there is a call to the service, an authentication will be attempted using the credentials passed in the `Workspace` constructor or its `credentials` property. If no credentials were passed, several authentication methods will be attempted by the [DefaultAzureCredential](#).

## Generate the terms for the problem

Next, you need to transform the mathematical representation of the problem into code. As a reminder, this is what the cost function looks like:

$$\$ \$ H^2 = \text{Large}(\sum_{i \in A \cup B} w_i x_i)^2 \$ \$$$

Below, you can see the code required to generate the terms (`Term`) of the cost function:

```

from typing import List
from azure.quantum.optimization import Problem, ProblemType, Term

def createProblemForContainerWeights(containerWeights: List[int]) -> List[Term]:
    terms: List[Term] = []

    # Expand the squared summation
    for i in range(len(containerWeights)):
        for j in range(len(containerWeights)):
            if i == j:
                # Skip the terms where i == j as they can be disregarded:
                # w_i*w_j*x_i*x_j = w_i*w_j*(x_i)^2 = w_i*w_j
                # for x_i = x_j, x_i ∈ {1, -1}
                continue

            terms.append(
                Term(
                    c = containerWeights[i] * containerWeights[j],
                    indices = [i, j]
                )
            )

    return terms

```

#### NOTE

For a detailed explanation of how this function is derived, please refer to the [shipping sample](#) or the [Microsoft Learn module for optimization](#).

## Create a `Problem` instance

Now that you have a way to generate the terms for the problem, let's provide a specific example and build out the cost function:

```

# This array contains a list of the weights of the containers:
containerWeights = [1, 5, 9, 21, 35, 5, 3, 5, 10, 11]

# Create the Terms for this list of containers:
terms = createProblemForContainerWeights(containerWeights)

```

The next step is to create an instance of a `Problem` to submit to the Azure Quantum solver:

```

# Create the Problem to submit to the solver:
problem = Problem(name="Ship Loading Problem", problem_type=ProblemType.ising, terms=terms)

```

Above, you can see that you have provided the following parameters:

- `name` : The name of the problem, used to identify the job in the Azure portal later on
- `problem_type` : In this instance, you have chosen an `ising` representation for the problem due to the way we defined the cost function, however you could alternatively have chosen a `pulse` representation.
- `terms` : These are the terms defining the cost function that you generated previously.

## Submit your problem to Azure Quantum

Next, you will submit the `Problem` instance defined to Azure Quantum.

```
from azure.quantum.optimization import ParallelTempering

# Instantiate a solver instance to solve the problem
solver = ParallelTempering(workspace, timeout=100) # timeout in seconds

# Optimize the problem
result = solver.optimize(problem)
```

Here you created an instance of a `ParallelTempering` solver for the problem. You could have chosen other Microsoft QIO optimization solvers (for example, `SimulatedAnnealing`) without needing to change more lines of code. To see a list of the available solvers, go to the [reference page](#).

The type `Problem` is the common parameter for all the solvers of Azure Quantum.

You then call `solver.optimize()` and supply the `problem` as the argument. This submits the problem synchronously to Azure Quantum and returns a Python dictionary of values to save the `result` variable for parsing in the next step.

You can also submit problems asynchronously. For more info, you can go to the guide for [solving long-running problems](#).

## Results readout

The final step is to transform the result returned by calling `solver.optimize()` to something human-readable. The following code takes the configuration `dict` returned by the service and prints out a list of container assignments:

```
def printResultSummary(result):
    # Print a summary of the result
    shipAWeight = 0
    shipBWeight = 0
    for container in result['configuration']:
        containerAssignment = result['configuration'][container]
        containerWeight = containerWeights[int(container)]
        ship = ''
        if containerAssignment == 1:
            ship = 'A'
            shipAWeight += containerWeight
        else:
            ship = 'B'
            shipBWeight += containerWeight

        print(f'Container {container} with weight {containerWeight} was placed on Ship {ship}')

    print(f'\nTotal weights: \n\tShip A: {shipAWeight} tonnes \n\tShip B: {shipBWeight} tonnes')

printResultSummary(result['solutions'][0])
```

The output should look something like this:

```
Container 0 with weight 1 was placed on Ship A
Container 1 with weight 5 was placed on Ship B
Container 2 with weight 9 was placed on Ship A
Container 3 with weight 21 was placed on Ship A
Container 4 with weight 35 was placed on Ship B
Container 5 with weight 5 was placed on Ship B
Container 6 with weight 3 was placed on Ship B
Container 7 with weight 5 was placed on Ship B
Container 8 with weight 10 was placed on Ship A
Container 9 with weight 11 was placed on Ship A
```

Total weights:

```
Ship A: 52 tonnes
Ship B: 53 tonnes
```

#### NOTE

If you run into an error while working with Azure Quantum, you can check our [list of common issues](#).

## Next steps

During this quick-start guide, you have seen an end-to-end example of how to take a mathematical cost function, represent it in code, submit it to Azure Quantum and parse the results. To learn more about the Microsoft QIO offering in Azure Quantum, please see the [Microsoft QIO Provider documentation](#).

For more detailed information on the shipping optimization problem, please refer to the following resources:

- [Ship loading sample](#)
- [Microsoft Learn module](#)

Once you have explored the ship loading sample in more detail, you may find it useful to tackle the more complex [job shop scheduling sample](#). The associated Microsoft Learn module can be found [here](#).

# 1QBit optimization quickstart for Azure Quantum

6/1/2021 • 8 minutes to read • [Edit Online](#)

Learn how to use 1QBit in Azure Quantum to solve complex optimization problems.

## Prerequisites

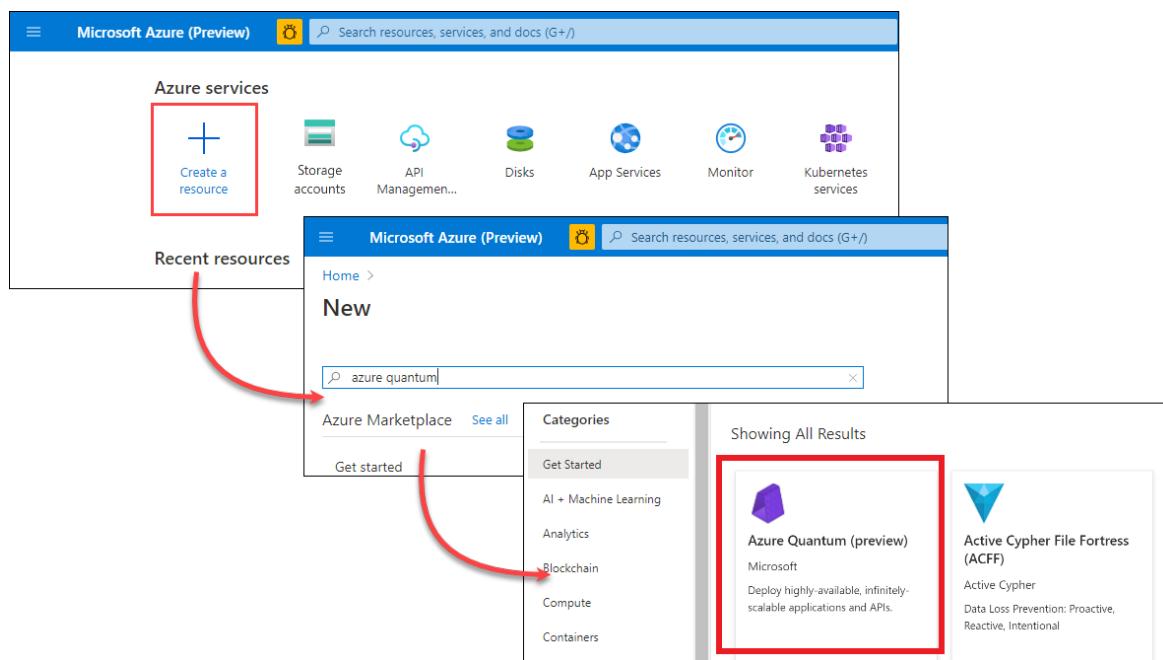
- To complete this tutorial you need an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.

## Create an Azure Quantum workspace

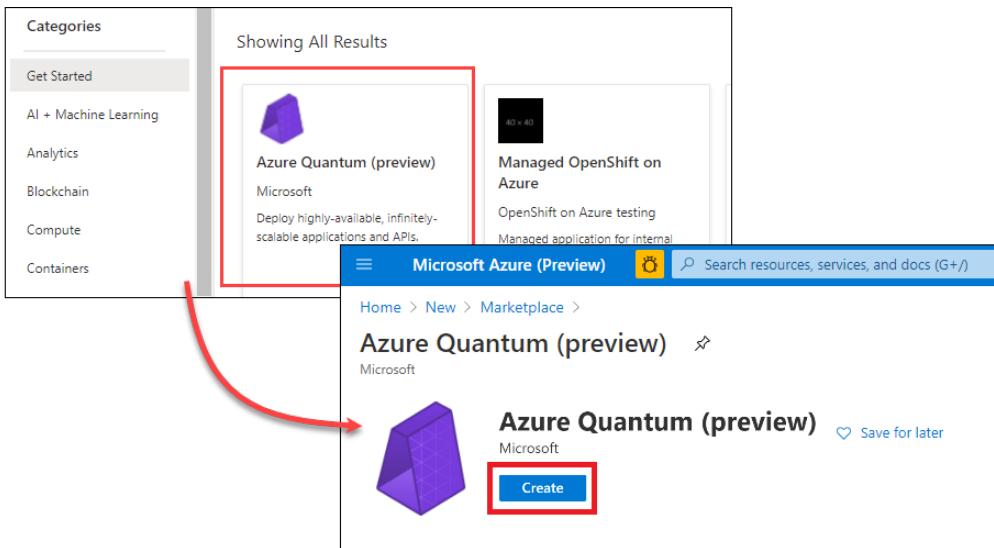
You use the Azure Quantum service by adding an Azure Quantum workspace resource to your Azure subscription in the Azure portal. An Azure Quantum workspace resource, or workspace for short, is a collection of assets associated with running quantum or optimization applications.

To open the Azure Portal, go to <https://portal.azure.com> and then follow these steps:

1. Click **Create a resource** and then search for **Azure Quantum**. On the results page, you should see a tile for the **Azure Quantum (preview)** service.



2. Click **Azure Quantum (preview)** and then click **Create**. This opens a form to create a workspace.



3. Fill out the details of your workspace:

- **Subscription:** The subscription that you want to associate with this workspace.
- **Resource group:** The resource group that you want to assign this workspace to.
- **Name:** The name of your workspace.
- **Region:** The region for the workspace.
- **Storage Account:** The Azure storage account to store your jobs and results. If you don't have an existing storage account, click **Create a new storage account** and complete the necessary fields. For this preview, we recommend using the default values.

**Project details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

<b>Subscription *</b> ⓘ Azure Quantum PM and Outreach	<b>Resource group *</b> ⓘ Documentation <a href="#">Create new</a>
<b>Instance details</b>	
<b>Name *</b> ⓘ bbQuantumWksp3	<b>Region *</b> ⓘ (US) West US
<b>Storage Account</b> ⓘ quantumdoc <a href="#">Create a new storage account</a>	

[Review + create](#)   [Previous](#)   [Providers >](#)

#### NOTE

You must be an Owner of the selected resource group to create a new storage account. For more information about how resource groups work in Azure, see [Control and organize Azure resources with Azure Resource Manager](#).

4. After completing the information, click the **Providers** tab to add providers to your workspace. A provider gives you access to a quantum service, which can be quantum hardware, a quantum simulator, or an optimization service.

#### Available providers

 1Qcloud Optimization Platform Optimization  1Bit 1Qcloud Optimization Platform with Quantum Inspired Solutions	 Honeywell Quantum Solutions Quantum Computing  Access to Honeywell Quantum Solutions' trapped-ion systems	 IonQ Quantum Computing  IonQ's trapped ion quantum computers perform calculations by manipulating charged atoms of Ytterbium held in a vacuum with lasers.	 Microsoft QIO Optimization  Ground-breaking optimization algorithms inspired by decades of quantum research.
<a href="#">+ Add</a>	<a href="#">+ Add</a>	<a href="#">+ Add</a>	<a href="#">✓ Added</a>

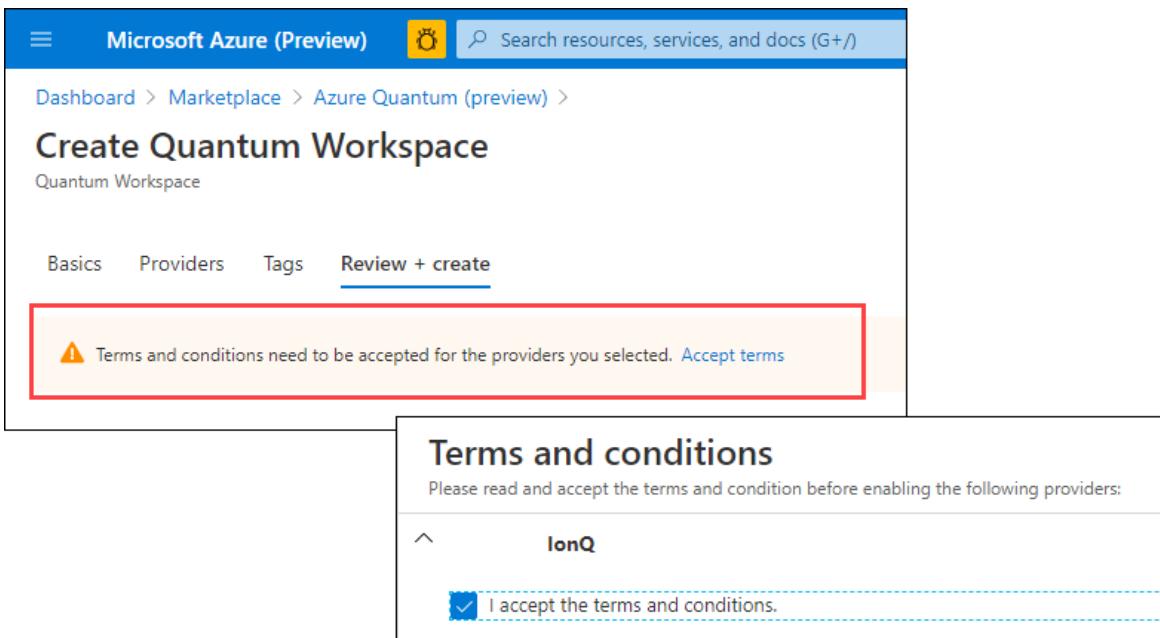
#### Providers added to this workspace

Name ↑↓	Provider type ↑↓	SKU	
 Microsoft QIO Microsoft	Optimization	Learn & Develop	<a href="#">Modify</a> <a href="#">Remove</a>
<a href="#">Review + create</a>		<a href="#">Previous</a>	<a href="#">Tags &gt;</a>

#### NOTE

By default, the Azure Quantum service adds the Microsoft QIO provider to every workspace.

5. Add at least the 1Qcloud Optimization Platform provider, then click **Review + create**.
6. Review the setting you've selected and if everything is correct, click **Create** to create your workspace.



The screenshot shows the Azure portal interface for creating a Quantum Workspace. The top navigation bar includes the Microsoft Azure logo, a search bar, and a dashboard menu. Below the header, the breadcrumb navigation shows: Dashboard > Marketplace > Azure Quantum (preview) > Create Quantum Workspace. The main content area is titled "Create Quantum Workspace" and has a sub-section for "Quantum Workspace". There are four tabs at the top of this section: Basics, Providers, Tags, and Review + create. The "Review + create" tab is currently selected and highlighted with a blue underline. A red box surrounds a warning message: "⚠ Terms and conditions need to be accepted for the providers you selected. [Accept terms](#)". A modal window titled "Terms and conditions" is displayed, containing the text "Please read and accept the terms and condition before enabling the following providers:" followed by a list of providers (IonQ) and a checkbox labeled "I accept the terms and conditions.".

#### NOTE

Pricing for Azure Quantum varies by provider. Please consult the information in the Providers tab of your Azure Quantum workspace in the Azure portal for the most up-to-date pricing information.

## Define your optimization problem

In this guide, you will solve a simple optimization example to get started with the optimization services of Azure Quantum. This quickstart is based on the [ship loading sample](#).

Suppose there are two ships ready to be loaded with containers and a list of containers of varying weights to be assigned to each ship. The aim of the optimization problem is to assign containers to each ship in such a way that the weight is distributed as evenly as possible between both ships.

The cost function for this optimization problem looks like the following:

$$\$ H^2 = \text{Large}(\sum_{i \in A \cup B} w_i x_i)^2 \$$$

This cost function has the following properties:

- If all the containers are on one ship, the function is at its highest value - reflecting that this is the least optimal solution
- If the containers are perfectly balanced, the value of the summation inside the square is  $\$0\$$  - the function is at its lowest value. This solution is optimal.

The goal is to find the configuration that yields the lowest possible value of  $H^2$ .

#### NOTE

For a detailed walkthrough of the problem scenario and how the cost function is constructed, please refer to the [sample](#) and/or the associated [Microsoft Learn module](#).

## Install the Python SDK for Azure Quantum

To implement a solution, first ensure that you have the Python SDK for Azure Quantum installed on your machine. If you don't have it installed yet, follow these steps:

1. Install [Python](#) 3.6 or later in case you haven't already.
2. Install [PIP](#) and ensure you have [version 19.2 or higher](#).
3. Install the `azure-quantum` python package.

```
pip install --upgrade azure-quantum
```

## Create a `Workspace` object in your Python code and log in

Now create a Python file or Jupyter Notebook, import the `Workspace` module from `azure.quantum`, and create a `Workspace` object. This is what you will use to submit our optimization problem to Azure Quantum. The value for `resource_id` and `location` can be found on the Azure Portal page for the [workspace you created](#).

```
from azure.quantum import Workspace

# Copy the settings for your workspace below
workspace = Workspace(
    resource_id = "", # add the Resource ID of the Azure Quantum workspace you created
    location = "" # add the location of your Azure Quantum workspace (e.g. "westus")
)
```

The first time you run a method which interacts with the Azure service, a window might prompt in your default browser asking for your credentials. You can optionally pass a credential to be used in the authentication in the construction of the `Workspace` object or via its `credentials` property. See more at [Azure.Quantum.Workspace](#)

#### NOTE

The `workspace.login()` method has been deprecated and is no longer necessary. The first time there is a call to the service, an authentication will be attempted using the credentials passed in the `Workspace` constructor or its `credentials` property. If no credentials were passed, several authentication methods will be attempted by the `DefaultAzureCredential`.

## Generate the terms for the problem

Next, you need to transform the mathematical representation of the problem into code. As a reminder, this is what the cost function looks like:

$$\text{H}^2 = \text{Large}(\sum_{i \in A \cup B} w_i x_i)^2$$

Below, you can see the code required to generate the terms (`Term`) of the cost function:

```
from typing import List
from azure.quantum.optimization import Problem, ProblemType, Term

def createProblemForContainerWeights(containerWeights: List[int]) -> List[Term]:

    terms: List[Term] = []

    # Expand the squared summation
    for i in range(len(containerWeights)):
        for j in range(len(containerWeights)):
            if i == j:
                # Skip the terms where i == j as they can be disregarded:
                # w_i*w_j*x_i*x_j = w_i*w_j*(x_i)^2 = w_i*w_j
                # for x_i = x_j, x_i ∈ {1, -1}
                continue

            terms.append(
                Term(
                    w = containerWeights[i] * containerWeights[j],
                    indices = [i, j]
                )
            )

    return terms
```

#### NOTE

For a detailed explanation of how this function is derived, please refer to the [shipping sample](#) or the [Microsoft Learn module for optimization](#).

## Create a `Problem` instance

Now that you have a way to generate the terms for the problem, let's provide a specific example and build out the cost function:

```
# This array contains a list of the weights of the containers:
containerWeights = [1, 5, 9, 21, 35, 5, 3, 5, 10, 11]

# Create the Terms for this list of containers:
terms = createProblemForContainerWeights(containerWeights)
```

The next step is to create an instance of a `Problem` to submit to the Azure Quantum solver:

```
# Create the Problem to submit to the solver:  
problem = Problem(name="Ship Loading Problem", problem_type=ProblemType.ising, terms=terms)
```

Above, you can see that you have provided the following parameters:

- `name` : The name of the problem, used to identify the job in the Azure portal later on
- `problem_type` : In this instance, you have chosen an `ising` representation for the problem due to the way we defined the cost function, however you could alternatively have chosen a `pobo` representation.
- `terms` : These are the terms defining the cost function that you generated previously.

## Submit your problem to Azure Quantum

Next, you will submit the `Problem` instance defined to Azure Quantum.

```
from azure.quantum.optimization.oneqbit import PathRelinkingSolver  
  
# Instantiate a solver instance to solve the problem  
solver = PathRelinkingSolver(workspace)  
  
# Optimize the problem  
result = solver.optimize(problem)
```

Here you created an instance of a `PathRelinkingSolver` solver for the problem. You could also have chosen other 1QBit solvers (for example, `TabuSearch`) without needing to change more lines of code. To see a list of the available solvers, go to the [reference page](#).

The type `Problem` is the common parameter for all the solvers of Azure Quantum.

You then call `solver.optimize()` and supply the `problem` as the argument. This submits the problem synchronously to Azure Quantum and returns a Python dictionary of values to save the `result` variable for parsing in the next step.

## Results readout

The final step is to transform the result returned by calling `solver.optimize()` to something human-readable. The following code takes the configuration `dict` returned by the service and prints out a list of container assignments:

```

def printResultSummary(result):
    # Print a summary of the result
    shipAWeight = 0
    shipBWeight = 0
    for container in result['configuration']:
        containerAssignment = result['configuration'][container]
        containerWeight = containerWeights[int(container)]
        ship = ''
        if containerAssignment == 1:
            ship = 'A'
            shipAWeight += containerWeight
        else:
            ship = 'B'
            shipBWeight += containerWeight

        print(f'Container {container} with weight {containerWeight} was placed on Ship {ship}')

    print(f'\nTotal weights: \n\tShip A: {shipAWeight} tonnes \n\tShip B: {shipBWeight} tonnes')

printResultSummary(result['solutions'][0])

```

The output should look something like this:

```

Container 0 with weight 1 was placed on Ship A
Container 1 with weight 5 was placed on Ship B
Container 2 with weight 9 was placed on Ship A
Container 3 with weight 21 was placed on Ship A
Container 4 with weight 35 was placed on Ship B
Container 5 with weight 5 was placed on Ship B
Container 6 with weight 3 was placed on Ship B
Container 7 with weight 5 was placed on Ship B
Container 8 with weight 10 was placed on Ship A
Container 9 with weight 11 was placed on Ship A

Total weights:
    Ship A: 52 tonnes
    Ship B: 53 tonnes

```

#### NOTE

If you run into an error while working with Azure Quantum, you can check our [list of common issues](#).

## Next steps

During this quick-start guide, you have seen an end-to-end example of how to take a mathematical cost function, represent it in code, submit it to Azure Quantum and parse the results. To learn more about the 1QBit offering in Azure Quantum, please see the [1QBit Provider documentation](#).

For more detailed information on the shipping optimization problem please refer to the following resources:

- [Ship loading sample](#)
- [Microsoft Learn module](#)

Once you have explored the ship loading sample in more detail, you may find it useful to tackle the more complex [job shop scheduling sample](#). The associated Microsoft Learn module can be found [here](#).

# Install and use the Python SDK for Azure Quantum

6/17/2021 • 3 minutes to read • [Edit Online](#)

This guide provides a basic overview of how to install and use the Python SDK for Azure Quantum.

## Prerequisites

- An Azure Quantum workspace created in your Azure subscription. To create a workspace, see [Create an Azure Quantum workspace](#).

## Python SDK for Azure Quantum installation

The Python SDK is distributed as the `azure-quantum` PyPI package. To install the package you will need to follow the steps below:

1. Install [Python](#) 3.6 or later.
2. Install [PIP](#), the Python Package Installer, and ensure you have **version 19.2 or higher**.
3. Install the latest `azure-quantum` Python package:

```
pip install --upgrade azure-quantum
```

## Jupyter Notebooks installation

You can also choose to interact with Azure Quantum optimization using Jupyter Notebooks. In order to do this, you will need to:

1. Install the Python SDK for Azure Quantum (as described in the previous section)
2. [Install Jupyter Notebooks](#)
3. In your terminal of choice, use the following command to launch a new Jupyter Notebook:

```
jupyter notebook
```

This will launch a new browser window (or a new tab) showing the Notebook Dashboard, a sort of control panel that allows you (among other things) to select which notebook to open.

4. In the browser view, select the dropdown button on the right hand top corner and select `Python 3` from the list. This should create a new notebook.

## Usage example

Whether you choose to solve optimization problems using Jupyter Notebooks or a Python script, once you have installed the prerequisites from the previous sections you can follow the instructions below to run a test problem.

## Connecting to an Azure Quantum workspace

A `Workspace` represents the Azure Quantum workspace you [previously created](#) and is the main interface for

interacting with the service.

```
from typing import List
from azure.quantum.optimization import Term
from azure.quantum import Workspace

workspace = Workspace (
    subscription_id = "", # Add your subscription_id
    resource_group = "", # Add your resource_group
    name = "", # Add your workspace name
    location = "" # Add your workspace location (for example, "westus")
)
```

The first time you run a method which interacts with the Azure service, a window might prompt in your default browser asking for your credentials. You can optionally pass a credential to be used in the authentication in the construction of the `Workspace` object or via its `credentials` property. See more at [Azure.Quantum.Workspace](#)

#### NOTE

The `workspace.login()` method has been deprecated and is no longer necessary. The first time there is a call to the service, an authentication will be attempted using the credentials passed in the `Workspace` constructor or its `credentials` property. If no credentials were passed, several authentication methods will be attempted by the [DefaultAzureCredential](#).

## Expressing and solving a simple problem

To express a simple problem to be solved, create an instance of a `Problem` and set the `problem_type` to either `ProblemType.ising` or `ProblemType.pujo`. For more information, see [ProblemType](#).

```
from azure.quantum.optimization import Problem, ProblemType, Term, ParallelTempering

problem = Problem(name="My First Problem", problem_type=ProblemType.ising)
```

Next, create an array of `Term` objects and add them to the `Problem`:

```
terms = [
    Term(c=-9, indices=[0]),
    Term(c=-3, indices=[1,0]),
    Term(c=5, indices=[2,0]),
    Term(c=9, indices=[2,1]),
    Term(c=2, indices=[3,0]),
    Term(c=-4, indices=[3,1]),
    Term(c=4, indices=[3,2])
]

problem.add_terms(terms=terms)
```

#### NOTE

There are [multiple ways](#) to supply terms to the problem, and not all terms must be added at once.

Next, we're ready to apply a `solver`. In this example we'll use a parameter-free version of parallel tempering. You can find documentation on this solver and the other available solvers in the [Microsoft QIO provider reference](#).

```
solver = ParallelTempering(workspace, timeout=100)

result = solver.optimize(problem)
print(result)
```

This method will submit the problem to Azure Quantum for optimization and synchronously wait for it to be solved. You'll see output like the following in your terminal window or Jupyter Notebook:

```
{'solutions': [{['configuration': {'0': 1, '1': 1, '2': -1, '3': 1}, 'cost': -32.0}]}]
```

#### NOTE

If you run into an error while working with Azure Quantum, you can check our [list of common issues](#). Also if you are using an optimization solver and you get an error in the form , you can check our [list of common user errors in optimization solvers](#).

## Next steps

### Documentation

- [Solver overview](#)
- [Expressing problems & supplying terms](#)
- [Interpreting solver results](#)
- [Job management](#)
- [Solve long-running problems \(async problem submission\)](#)
- [Reuse problem definitions](#)
- [Authenticating with a service principal](#)
- [Solvers reference for Microsoft QIO solvers](#)

### Samples and end-to-end learning

- [QIO samples repo](#)
- [Getting started](#)
  - [1QBit](#)
  - [Microsoft QIO](#)
- [Ship loading sample problem](#)
  - [End-to-end Microsoft Learn Module](#)
  - [Sample code](#)
- [Job shop scheduling sample problem](#)
  - [End-to-end Microsoft Learn Module](#)
  - [Sample code](#)

# Troubleshooting user errors in optimization solvers

5/26/2021 • 6 minutes to read • [Edit Online](#)

This document lists the common user errors return by the Azure Quantum optimization solvers, and troubleshooting steps for each error. The error messages returned by solvers should have an error code (in the form \_<code>) attached that the user should use to find their specific issue. The code is prefixed with "AZQ".

## Insufficient Resources (Range 001-100)

Errors in this category are due to lack of resources to carry out a specific job. This could be caused by factors such as problem size, or parameter settings.

### AZQ001 - Memory Limited

**Cause:** This error happens when the submitted problem is too large (usually, due to too many terms) and cannot fit into memory. Users can *estimate* how much memory their problem will use with the following formula (although not 100% precise, it is close to real usage):

```
memory_bytes = sum_coefficient_degrees_total*num_variables/8
```

Where

- `num_variables` is the number of variables in the problem
- `sum_coefficient_degrees_total` is the total sum of the number of variables in each term

An example with this formulation:

```
"terms": [
  {
    "c": 1.0,
    "ids": [0, 1, 2]
  },
  {
    "c": 1.0,
    "ids": [2, 3, 4]
  }
]
```

Here `num_variables = 5` ( $\{0,1,2,3,4\}$ ) and `sum_coefficient_degrees_total = 6` ( $3 + 3$ ). The total bytes of memory estimated is  $6 * 5 / 8 = 3.75$  \text{bytes}.

**Possible actions to take:** This error is hard to "fix" because some problems will unavoidably have large expanded term expressions, especially higher order problems. If you see this error then most likely at this time, our solvers are not capable of solving your problem. However, Azure Quantum is continuously developing support for more complex term expressions that can reduce the problem size in the future. Be sure to communicate the need for this feature to our support team!

In the mean time, consider:

- Removing constant terms (for example, terms without variables, Ising terms with even variable power, etc.)

### AZQ002 - Timeout Insufficient

**Cause:** This error happens when using parameter-free solvers specifically. It means that the "timeout"

parameter (in seconds) is set too low for any meaningful exploration. Each solver has a different search process and some solvers will take longer than others.

The table below shows the **bare minimum** timeout needed for a particular problem size to get a result (note: not necessarily a good one). Based on the size of your problem, you can adjust the numbers accordingly.

PROBLEM	SIMULATED ANNEALING	PARALLEL TEMPERING	TABU SEARCH
Variables: 1024, Terms: 195k	5s	100s	1s

Possible actions to take:

- Increase the timeout value. This will depend on the solver that is being called (see table above for starting point).
- If the problem is particularly large (10k+ variables) and/or with many terms, a larger timeout might be needed.

## Invalid Input Data (Range 101-200)

Errors in this category are due to mistakes in the user inputs - either there was an issue with the cost function expression, or parameters are invalid.

### AZQ101 - Duplicated Variable

**Cause:** This error happens when using the Ising cost function. Azure Quantum solvers will only accept single-degree variables so if the user is submitting higher power variables, an error will be thrown.

Possible actions to take:

- Condense all higher power variables into either a single variable, or a constant (1). See example below:

```
"terms": [
  {
    "c": 1.0,
    "ids": [0, 0, 0, 1, 1]
  },
  {
    "c": 1.0,
    "ids": [2, 2]
  }
]
```

If PUBO/HOBO, this should get condensed to:

```
"terms": [
  {
    "c": 1.0,
    "ids": [0, 1]
  },
  {
    "c": 1.0,
    "ids": [2]
  }
]
```

If Ising, even-powered terms are 1 so this should get condensed to:

```

"terms": [
  {
    "c": 1.0,
    "ids": [0]
  },
  {
    "c": 1.0,
    "ids": []
  }
]

```

## AZQ102 - Missing sections in Input Data

**Cause:** This error happens when there are missing fields in the input data. This usually only happens when submitting to the API directly, and not via the SDK. The SDK automatically formats the submitted terms in the correct structure.

The solver usually returns a more specific error with which fields are missing in the input.

**Possible actions to take:**

- Look at the specific error message and determine which field is missing from the input. Ensure field is present and has a value of the correct type.
- Ensure all fields are present and have values in the input expression. **Avoid empty strings "" and null values.**

```
{
  "cost_function": {
    "type": "pubo",
    "version": "1.0",
    "terms": [
      {
        "c": 1.0,
        "ids": [0, 1]
      },
      {
        "c": 0.36,
        "ids": [1, 2]
      }
    ]
  }
}
```

## AZQ103 - Invalid Types in Input Data

**Cause:** This error happens when there are fields in the data with invalid types. For more information, see [How to express problem terms correctly](#). The table below shows the expected types for each field.

FIELD NAME	EXPECTED TYPE	COMMON ERRORS
c	double	Submitting this field in string form, for example "2.0" or null form.
indices	list of integers	Submitting this field in string form "[0,1]" or individual items as string form ["0", "1"]. Empty lists are allowed and will be treated as constants.

This could also happen in the parameters supplied to the solver.

**Possible actions to take:**

- Ensure all types are converted correctly when using the SDK, especially if you are parsing these values from

strings or existing files.

- Ensure parameters that are lists are not supplied as strings of lists - e.g. [0,1] instead of "[0,1]" or ["0", "1"]

#### AZQ104 - Initial Config Error

**Cause:** This is a group of errors related to using the initial configuration setting. The error message returned from the solver should contain the specific message. Possible causes can include:

- Variable values are invalid and do not match the given problem type (Ising/PUBO). For example, this error will appear if the original problem type is Ising, and the initial configuration variables are found with values outside of 1 and -1.
- Variable dimensions supplied in the initial configuration do not match the variable size of the original problem.

**Possible actions to take:**

- Ensure that initial configuration settings for variables are valid and only take 2 values (either (0|1) or (-1|1)).
- Ensure the variables in the configuration map are part of the initial problem and that you did not include any new variable ids.

#### AZQ105 - Couldn't Parse Input

**Cause:** This error happens when the solver is unable to parse the input data. This usually only happens when submitting to the API directly, and not via the SDK. The SDK automatically formats the submitted terms in the correct json structure.

This happens when there's a syntax error in the input data json, or parameter file json.

**Possible actions to take:**

- Use a json lint or formatter tool on the invalid json and correct the syntax errors.
- Formulate problems with the SDK (which automatically submits problems in correct form)

#### AZQ106 - Feature Switch Error

**Cause:** This error happens when the feature switch functionality is used and an invalid feature id is supplied.

**Possible actions to take:**

- Reference the documentation and disable only the supported feature ids.

#### AZQ107 - Invalid Values in Input

**Cause:** This error happens when there are forbidden or invalid values being inputted. This mostly happens if the parameters you set for the solver are not valid.

**Possible actions to take:**

- Look at the specific error message and determine which field is invalid and the range of allowed values for that field.
- Run the parameter-free solver version to obtain a starting point for parameter values.
- Most parameters require a value of >0 (e.g. restarts, replicas, sweeps etc.).

# Introduction to optimization

4/29/2021 • 2 minutes to read • [Edit Online](#)

At its simplest, optimization is just the process of selecting the best choice from a set of possible options.

**Optimization**

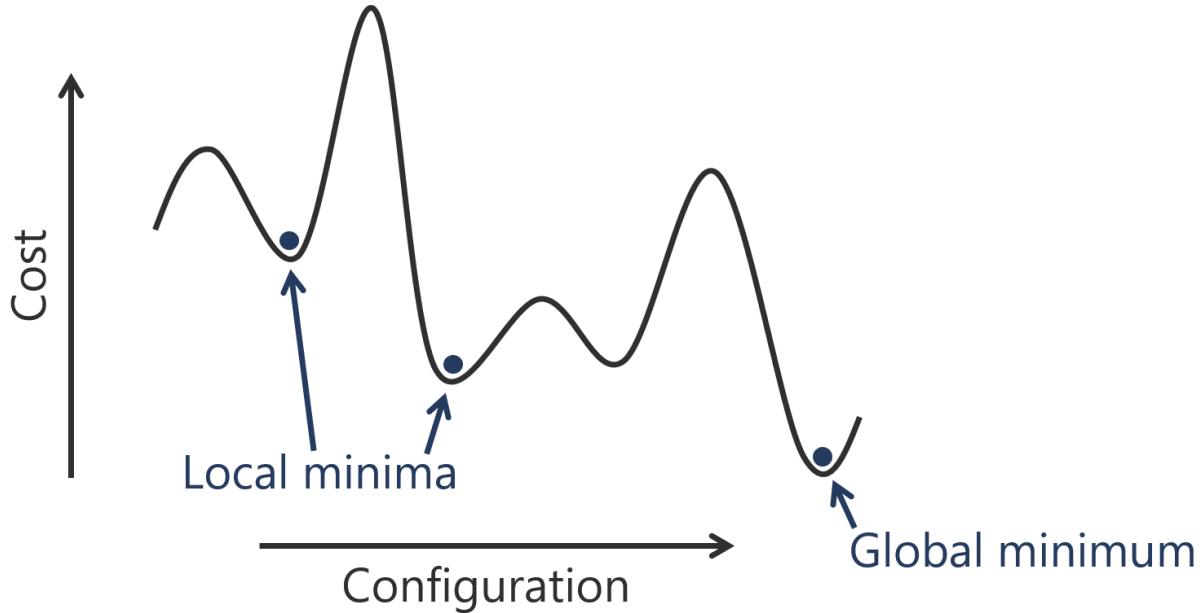
Optimization solvers that run on classical hardware.

LEARN	CODE	USE
Code Samples	Visual Studio	Python SDK
MS Learn	VS Code	Optimization API
	Jupyter Notebooks	
	Python IDE	
SOLVE	RUN	
Optimization solvers	CPUs, FPGAs, GPUs	

We can define *best* in many ways: it could be the option with the lowest cost, the quickest runtime or perhaps the lowest environmental impact. To keep things simple, we usually just refer to this definition of best as a cost to be minimized. If we wanted to maximize the cost instead (for example, if we wanted to maximize energy output from a solar cell), all we would need to do is multiply the cost by negative one and then minimize it.

Usually when we say we have an optimization problem, we mean we have a pretty complicated one, where lots of variables can interact in many ways to influence the final cost. We call a particular arrangement of the variables the *configuration* of the problem.

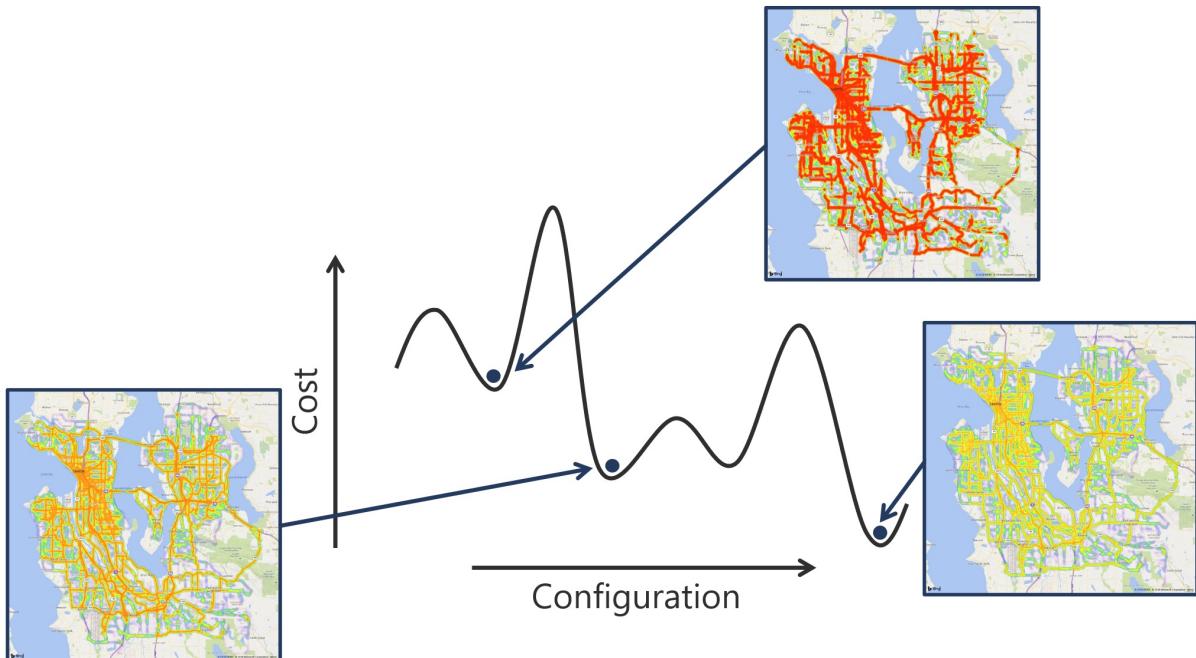
Because there are so many possible configurations to choose from, it is sometimes very difficult to identify the best solution, particularly when the problem space is very large. It can be easy to get stuck in a local optimum. Some examples of local optima are shown on the graph below, as well as the global optimum - the lowest cost configuration our system can adopt.



The way the cost varies as a function of the system configuration gives us what we call the *cost function* - the line traced on our graph. The goal of our optimization is to find the minimum point on this cost function (or as close to the minimum point as possible, given a reasonable amount of time).

Let's illustrate this with an example: traffic minimization. The aim of this optimization task is to reduce congestion in a road system to reduce the amount of time users spend waiting in traffic.

Each configuration represents a different combination of routes assigned to the vehicles in the system. The cost is the overall traffic level (or congestion level), which is what we wish to minimize.



The graph above highlights some examples of different system configurations, each of which has a different cost value. We have visualized the cost here using color: the redder the road segment, the higher the traffic level and therefore the greater the cost. Conversely, greener road segments have fewer vehicles simultaneously occupying them and therefore lower traffic and cost values.

# What are quantum-inspired algorithms?

3/5/2021 • 2 minutes to read • [Edit Online](#)

There are many types of quantum-inspired algorithms. One commonly used quantum-inspired algorithm is based on a computational model called *adiabatic quantum computing*. This approach uses a concept from quantum physics known as the adiabatic theorem. When you apply that theorem to solve a problem, you:

- First prepare a system and initialize it to its lowest energy state. For a simple system, one which we completely understand, this is easy to do.
- Next, slowly transform that system into a more complex one that describes the problem you are trying to solve. The adiabatic theorem states that, as long as this transformation happens slowly enough, the system has time to adapt and will stay in that lowest energy configuration. When we're done with our transformations, we've solved our problem.

A good analogy of this is to imagine you have a glass of water. If you move that glass slowly across a table, the contents won't spill because the system has time to adapt to its new configuration. If you were to move the glass quickly however, the system has been forced to change too quickly, and we have water everywhere.

Adiabatic quantum computation is an area of active research that's already being used in the industry. A number of techniques have been developed to simulate this type of physics. These kinds of classical algorithms, which we can run on classical computers today, are also known as *quantum-inspired optimization*.

## What is quantum-inspired optimization (QIO)?

Optimization problems are found in every industry, such as manufacturing, finance, and transportation. In fact, industries such as logistics are dedicated entirely to solving optimization problems. To solve these problems, we search through feasible solutions. The best solution is the one with the lowest cost. Adiabatic quantum algorithms are well-suited to solving many optimization problems.

Today, we can emulate adiabatic quantum algorithms by using quantum-inspired techniques on classical hardware, an approach which is known as quantum-inspired optimization (QIO). These techniques often perform better than state-of-the-art classical optimization techniques.

Applying QIO to real-world problems may offer businesses new insights or help lower costs by making their processes more efficient. QIO gives us the opportunity to:

- Find a solution faster than other optimization techniques for a fixed use case and fixed quality of solution.
- Find a higher quality solution than other optimization techniques for a fixed problem and fixed amount of time.
- Use a more realistic model than other optimization techniques by extending the problem to consider more variables.

Since QIO methods are heuristics, they're not guaranteed to find the optimal solution. Also, they don't always outperform other optimization techniques. In reality, it depends on the problem, and discovering what makes QIO perform better than other methods in some situations and not others is still an active area of research.

# Key concepts for optimization

6/1/2021 • 8 minutes to read • [Edit Online](#)

To understand optimization problems, you first need to learn some basic terms and concepts.

## Cost function

A **cost function** is a mathematical description of a problem which, when evaluated, tells you the value of that solution. Typically in optimization problems, we are looking to find the lowest value solution to a problem. In other words, we are trying to minimize the cost.

## Search space

The **search space** contains all the feasible solutions to an optimization problem. Each point in this search space is a valid solution to the problem but it may not necessarily be the lowest point, which corresponds to the lowest cost solution.

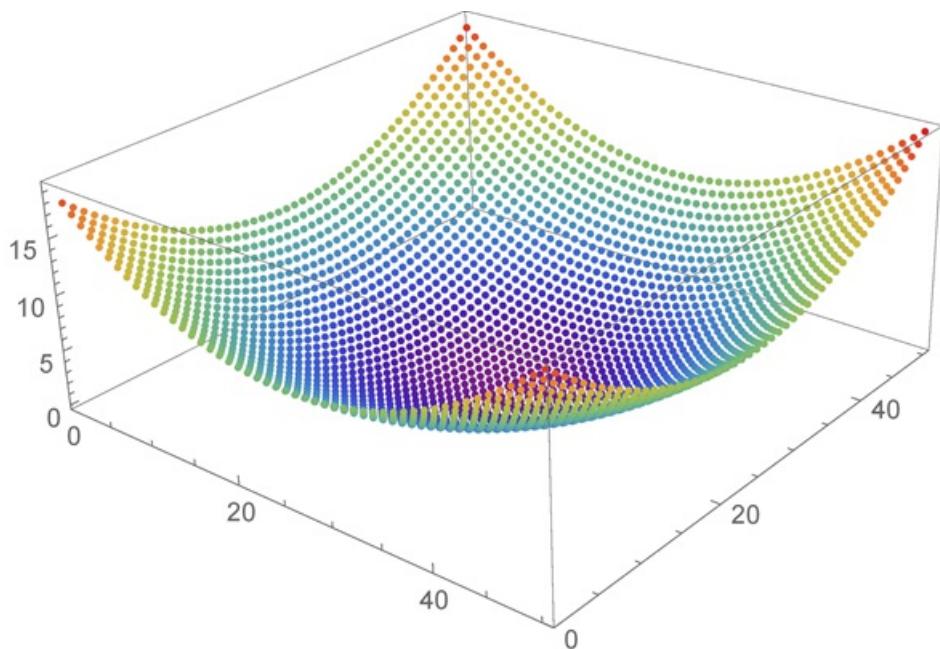
## Optimization landscape

Together, the search space and the cost function are often referred to as an **optimization landscape**. In the case of a problem that involves two continuous variables, the analogy to a landscape is quite direct.

Let's explore a few different optimization landscapes and see which are good candidates for Azure Quantum optimization.

### A smooth, convex landscape

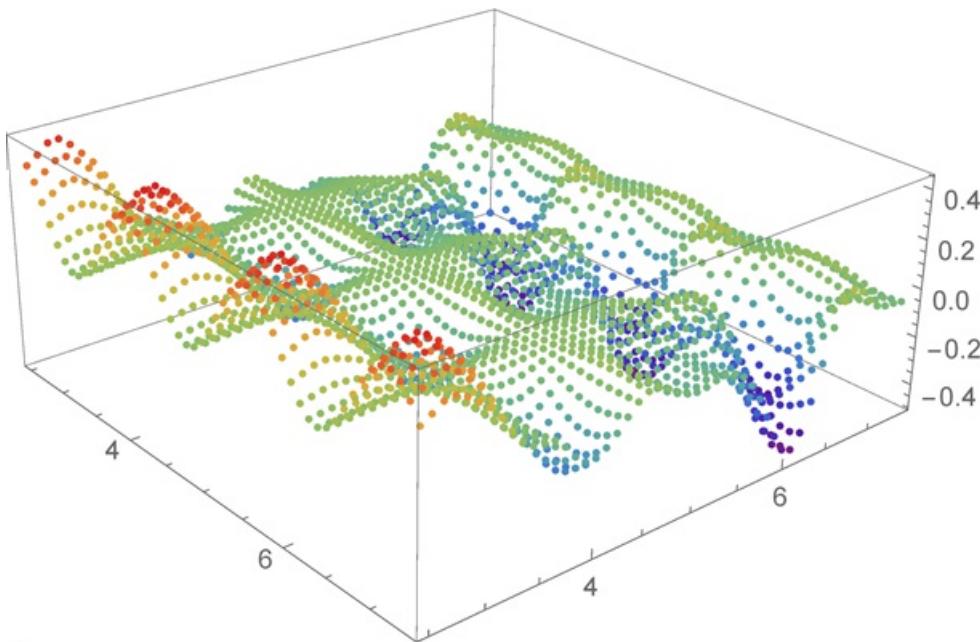
Consider the following plot of a cost function that looks like a single smooth valley:



This kind of problem is easily solved with techniques such as gradient descent, where you begin from an initial starting point and greedily move to any solution with a lower cost. After a few moves, the solution converges to the *global minimum*. The global minimum is the lowest point in the optimization landscape. Azure Quantum optimization solvers offer no advantages over other techniques with these straightforward problems.

## A structured, rugged landscape

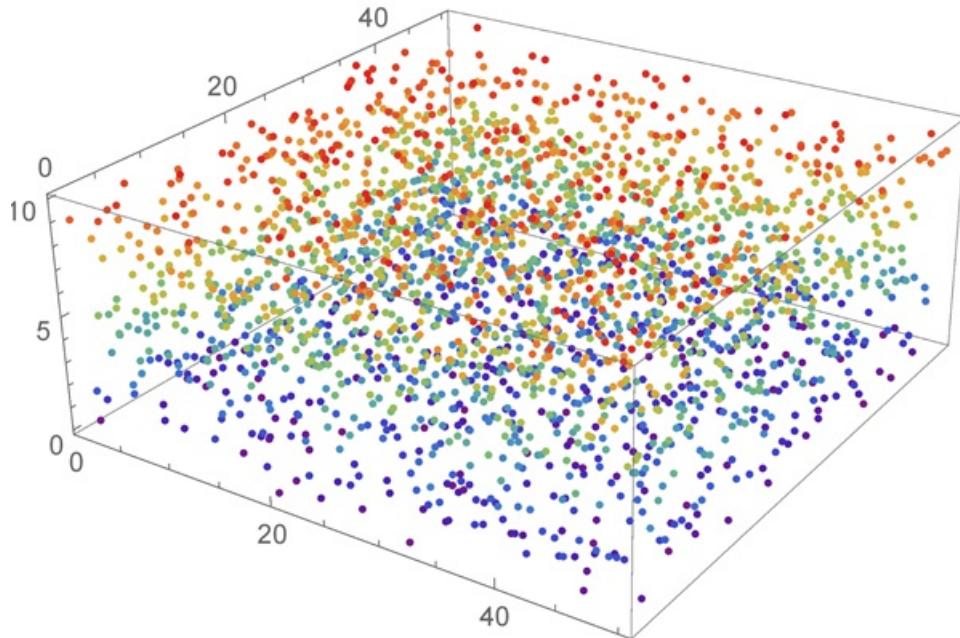
Azure Quantum works best with problems where the landscape is rugged, with many hills and valleys. Here's an example that considers two continuous variables.



In this scenario, one of the greatest challenges is to avoid getting stuck at any of the sub-optimal *local minima*. A rugged landscape can have multiple valleys. Each of these valleys will have a lowest point, which is the local minimum. One of these points will be the lowest overall, and that point is the global minimum. These rugged landscapes present situations where Azure Quantum optimization solvers can outperform other techniques.

## A scattered, random landscape

So far we have discussed smooth and rugged cost functions, but what if there is no structure at all? The following diagram shows such a landscape:



In these cases, where the solutions are completely random, then no algorithm can improve on a brute force search.

## Defining a cost function

As mentioned above, the cost function represents the quantity that you want to minimize. Its main purpose is to map each configuration of a problem to a single number. This allows the optimizer to easily compare potential solutions and determine which is better. The key to generating a cost function for your problem is in recognizing what parameters of your system affect the chosen cost.

In principle, the cost function could be any mathematical function  $f = f(x_0, x_1, \dots)$ , where the function variables  $x_1, x_2, \dots$  encode the different configurations. The smooth landscape shown above could for example be generated using a quadratic function of two continuous variables  $f(x, y) = x^2 + y^2$ . However, certain optimization methods may expect the cost function to be in a particular form. For instance, the Azure Quantum solvers expect a Binary Optimization Problem. For this problem type, configurations must be expressed via binary variables with  $x_i \in \{0, 1\}$ , and many problems are naturally suited to be expressed in this form.

## Variables

Let's take a look at how to define the cost function for a simple problem. Firstly, we have a number of variables. We can name these variables  $x$ , and if we have  $i$  variables, then we can index them individually as follows:

```
$$ x_{\{i\}} $$
```

For example, if we had 5 variables, we could index like so:

```
$$ x_{\{0\}}, x_{\{1\}}, x_{\{2\}}, x_{\{3\}}, x_{\{4\}} $$
```

These variables can take specific values, and in the case of a binary optimization problem they can only take two. In particular, if your problem is considering these variables as spins, as in the Ising model, then the values of the variables can be either +1 or -1.

For example:

```
$$ x_{\{0\}} = 1, x_{\{1\}} = 1, x_{\{2\}} = -1, x_{\{3\}} = -1, x_{\{4\}} = 1 $$
```

In other cases, the variables can simply be assigned 1 or 0, as in the Quadratic Unconstrained Binary Optimization (QUBO) or Polynomial Unconstrained Binary Optimization (PUBO) model.

For example:

```
$$ x_{\{0\}} = 1, x_{\{1\}} = 1, x_{\{2\}} = 0, x_{\{3\}} = 0, x_{\{4\}} = 1 $$
```

### NOTE

More information about cost functions and binary optimization models such as Ising and PUBO can be found in the article [Cost functions](#).

## Weights

Let us consider some variables. Each of these variables has an associated **weight**, which determines their influence on the overall cost function.

We can write these weights as  $w$ , and again, if we have  $i$  variables, then the associated weight for those individual variables can be indexed like this

```
$$ w_{\{i\}} $$
```

If we had 5 weights, we could index them like this:

```
$$ w_{\{0\}}, w_{\{1\}}, w_{\{2\}}, w_{\{3\}}, w_{\{4\}} $$
```

A **weight** can be any real-valued number. For example, we may give these weights the following values:

```
$$ w_{\{0\}} = 50, w_{\{1\}} = -2, w_{\{2\}} = 7, w_{\{3\}} = 24, w_{\{4\}} = -10 $$
```

## Terms

Terms are the combination of weights and variables, they look like this:

$\$ \$ w_{\{i\}}x_{\{i\}} \$ \$$

As an example, let's consider a term with index 0, a weight of 50, and a variable assignment of 1:

$\$ \$ w_{\{0\}}x_{\{0\}} = 50(1) = 50 \$ \$$

This was an example of evaluating the cost of a term.

## Cost function formulation

Returning to our definition of a cost function, it is a mathematical description of a problem which, when evaluated, tells you the value of that solution.

So to write a cost function, we write a sum of terms. That is, the sum of these weights and variables.

That looks like this:

$\$ \$ \backslash Large \sum_{\{i\}} w_{\{i\}}x_{\{i\}} \$ \$$

With 5 variables, this would be expanded to:

$\$ \$ w_{\{0\}}x_{\{0\}} + w_{\{1\}}x_{\{1\}} + w_{\{2\}}x_{\{2\}} + w_{\{3\}}x_{\{3\}} + w_{\{4\}}x_{\{4\}} \$ \$$

## Degree and "k-local"

The cost functions you will be working with will be **polynomial** functions of varying **degree**. The variables themselves will be binary ( $x_i \in \{\pm 1\}$  for Ising and  $x_i \in \{0, 1\}$  for QUBO/PUBO problems), which makes these binary optimization problems.

In some cases, the cost function will be linear. This means that the highest power any of the terms is raised to is 1.  $x + y + 1$  is an example of a linear function. Linear terms are said to have a **degree** of 1.

In other cases, you may have a **quadratic** cost function. In this case, the highest power any of the terms is raised to is 2.  $x^2 + y^2 + x + 1$  is an example of a quadratic function. These function therefore has a degree of 2 (you may see these referred to as **Quadratic Unconstrained Binary Optimization (QUBO)** problems).

When a cost function has terms raised to higher powers than 2, we refer to them as **Polynomial Unconstrained Binary Optimization (PUBO)** or **Higher Order Binary Optimization (HOBO)** problems. These cost functions have degrees higher than 2.  $x^3 + xy + x^2 + y^2 + x + 1$  is an example of a higher order polynomial function.

In general, we often talk about the maximum degree,  $k$ , and describe them as **k-local problems**. For instance, we might also refer to a QUBO as a 2-local problem. You can reduce a higher order polynomial function into a lower order one by introducing further variables, which will increase the problem size. This process is known as **degree reduction**.

In Azure Quantum, we use the term PUBO to describe problems with a maximum degree of  $k$ . This includes QUBO problems, as QUBOs are just PUBOs with degree 2.

## Heuristic

A **heuristic** is a technique for finding an approximate solution, when finding the exact solution may take too long. When we think about this in terms of our optimization landscape above, it may take a very long time to find the lowest cost solution, however we may be able to find a solution that is close to optimal in a reasonable amount of time. This often comes with experimentation: trying different techniques with different parameters and run times to find what gives good results.

## Walker

We can imagine a person or a particle in our **search space**, and each step taken creates a path, or walk, through the optimization landscape. Often, this will be referred to as a **walker**. Walkers can be used in a variety of ways, for example you may choose to have many walkers starting from the same starting point, or have them starting from different locations, and so on.

## Convert your problem to a Ising or QUBO model

Professor Andrew Lucas' paper [Ising formulations of many NP problems](#) is a good summary of how to convert an NP problem to QIO's QUBO or Ising model. You can download the paper from the link provided. After converting your field problem into the Ising or QUBO model, it is recommended to merge terms with the same variable list into a single term. For example:

```
$$ w_{\{0\}}x_{\{0\}}x_{\{1\}}, w_{\{1\}}x_{\{1\}}x_{\{0\}} $$
```

can be merged into

```
$$ (w_{\{0\}} + w_{\{1\}})x_{\{0\}}x_{\{1\}} $$
```

Expressed in terms of code:

```
term1 = Term(c=2, indices=[0,1])
term2 = Term(c=3, indices=[1,0])
```

can be merged into

```
merged_term = Term(c=5, indices=[0,1])
```

where the Python SDK is used to [express the terms of an optimization problem](#).

Merging terms may significantly improve the performance of QIO, if your problem has a lot of such terms. You can either use a hash map or sort algorithm to do the merging.

# Which optimization solver should I use?

6/4/2021 • 2 minutes to read • [Edit Online](#)

Azure Quantum offers a broad range of solvers for optimization problems. You can consult the full list in the [reference page](#). However, it is unfortunately not possible to determine *a priori* which solver will perform best for a new optimization problem. In the following, we describe our suggested strategy to find a suitable solver by benchmarking.

## Benchmarking objective

The benchmarking objective will have a large influence on the selection of a suitable solver. This objective for the solvers is dictated by the application. For example, one benchmarking objective is to find the closest solution to the global minimum. Another objective might be to find the closest solution to the global minimum but in a given time interval or for a specified runtime cost. If one is interested in solving many problems from a similar domain, a benchmarking objective might be to find a solver which produces good results for most instances, rather than returning very good results for some instances but failing to give a good enough solution for the remaining instances.

## Benchmarking strategy

Optimization problems from the same field of domain might share common features. We therefore suggest that you start with the solver which worked best for previous problems of the same domain and use this as a baseline. Nevertheless even in such a case it makes sense to benchmark the other solvers again from time to time.

We recommend that you to start with the parameter-free solvers as they don't require parameter tuning:

1. Parameter-free simulated annealing (SA): This provides a solid baseline for the runtime and possible minima.
2. Parameter-free parallel tempering (PT)

Automatically determining parameters for solvers is convenient but also creates a runtime overhead. If one has to solve many similar problems or wants to achieve better performance, parameterized solvers should be considered. We suggest to start with parameterized SA if the parameter-free SA solver provided better results than the parameter-free PT and otherwise start with parameterized PT.

Afterwards we recommend to benchmark the remaining solvers.

# Binary optimization

3/5/2021 • 3 minutes to read • [Edit Online](#)

Binary optimization is a subclass of more general combinatorial optimization problems in which the variables are restricted to a finite set of values, in this particular case just two. Therefore, a binary optimization problem can be stated as the effort of minimizing an objective function  $H(\vec{x})$  of  $N$  variables with values  $x_i \in \{0,1\}$  subject to some equality and/or inequality constraints. These constraints restrict the values that the variables can take, in order to ensure only feasible solutions are proposed. Some example constraints  $f(\vec{x})$  and  $g(\vec{x})$  are included below.

$$\min_{\vec{x}} H(\vec{x}) \quad \text{subject to} \quad f(\vec{x}) = 0, \quad g(\vec{x}) > 0.$$

Binary optimization constitutes a broad range of important problems of both scientific and industrial nature such as social network analysis, portfolio optimization in finance, traffic management and scheduling in transportation, lead optimization in pharmaceutical drug discovery, and many more.

## Polynomial Unconstrained Binary Optimization (PUBO)

PUBOs are a subset of binary optimization problems with no constraints where the objective function is a polynomial of the variables. The degree  $k$  of such a polynomial is often referred to as the *locality* of PUBO. For instance, the objective function of a  $k$ -local PUBO can be expressed in the following way:

$$H(\vec{x}) = \sum_{i_1}^N J_{i_1} x_{i_1} + \frac{1}{2!} \sum_{i_1, i_2}^N J_{i_1 i_2} x_{i_1} x_{i_2} + \dots + \frac{1}{k!} \sum_{i_1, \dots, i_k}^N J_{i_1 \dots i_k} x_{i_1} \dots x_{i_k},$$

in which  $J_{i_1 i_2 \dots i_n}$  represents the  $n$ -point interactions between variables  $x_{i_1}, x_{i_2}, \dots, x_{i_n}$ . We can interpret the variables to be (on) the nodes and their interactions as the edges of an underlying graph. Note that each of  $J_{i_1 i_2 \dots i_n}$  is a symmetric tensor since any permutation of the indices will leave the corresponding term in the above objective function unchanged, the reason being that such a permutation is equivalent to a reshuffling of the  $x_i$  values. In terms with an even number of variables, the diagonal terms can be ignored owing to the fact that  $x_i^2 = 1$ .

## Quadratic Unconstrained Binary Optimization (QUBO)

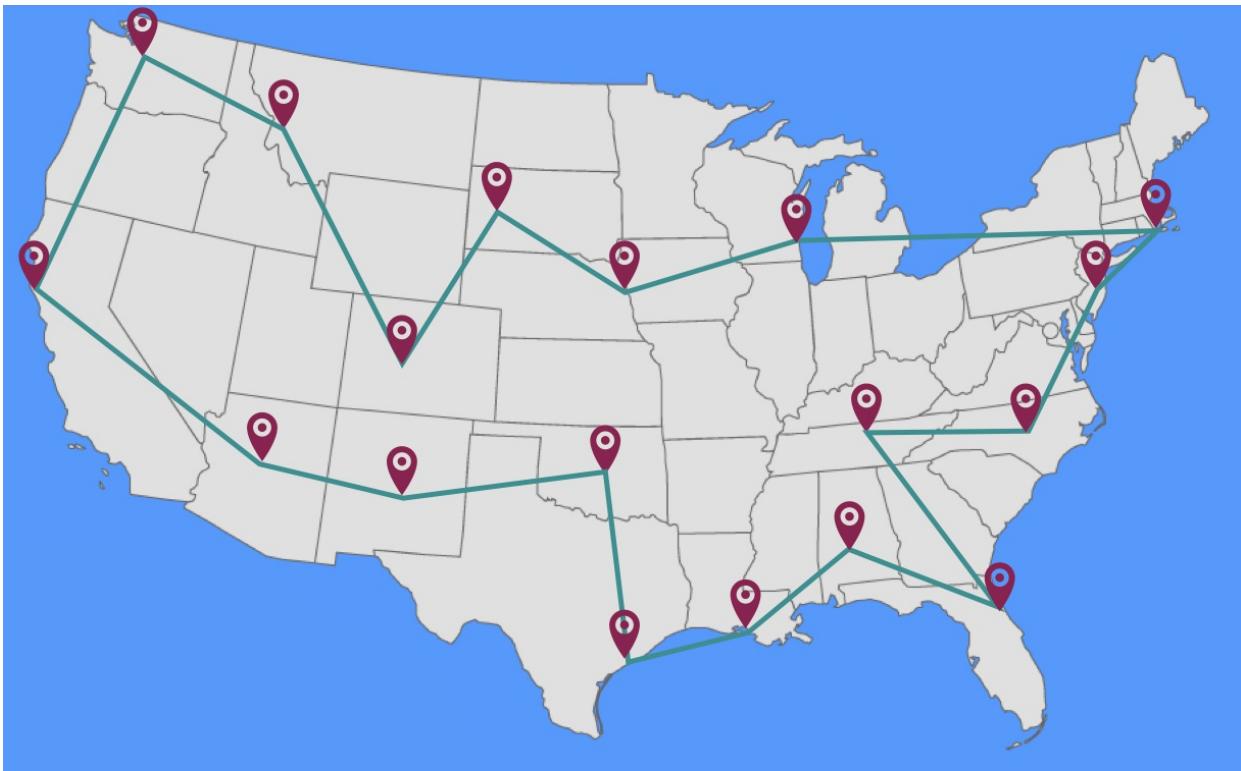
QUBOs are special cases of PUBOs where the order of the polynomial is 2, namely,

$$\min_{\vec{x}} H(\vec{x}) = \sum_i h_i x_i + \frac{1}{2} \sum_{i,j} J_{ij} x_i x_j, \quad \text{subject to} \quad x_i \in \{-1, 1\}.$$

If the elements of the adjacency matrix  $J_{ij}$  are drawn from a random distribution, the above objective function is equivalent to the *Hamiltonian* of so-called *spin glass*. Spin glasses are well-known in physics for their rugged energy landscape with exponentially many local minima which makes finding their ground state an NP-hard problem. Many non-trivial optimization problems such as max-cut, network flows, satisfiability, geometrical packing, graph coloring, and integer linear programming can be formulated as QUBOs although in some cases, doing so might increase the hardness of the problem.

### Example of a QUBO problem

Here we show the QUBO mapping for the famous Travelling Salesperson Problem (TSP).



The TSP is a simple yet important problem with applications in transportation. It consists of trying to find the shortest closed path between a set of  $N$  sites  $\mathcal{S}$  such that each site is visited exactly once except for the initial site. In other words, the path must not cross itself. Assuming that the distance between sites  $\alpha$  and  $\beta$  is  $w_{\alpha\beta}$ , the QUBO objective function can be expressed as follows:

$$\begin{aligned} H(\vec{x}) = & \sum_{\alpha, \beta \in \mathcal{S}} w_{\alpha\beta} x_{\alpha, i} x_{\beta, i+1} + \\ & \lambda \sum_{i=1}^N \left( 1 - \sum_{\alpha \in \mathcal{S}} x_{\alpha, i} \right)^2 + \\ & \lambda \sum_{\alpha \in \mathcal{S}} \left( 1 - \sum_{i=1}^N x_{\alpha, i} \right)^2, \end{aligned}$$

in which the binary variable  $x_{\alpha, i}$  is equal to 1 if site  $\alpha$  appears  $i$ th in the path, otherwise it is 0. The first term in the above expression is simply the length of the path, whereas the last two terms enforce the requirements that every site in  $\mathcal{S}$  appears in the path and no site is visited more than once, respectively. Note that the penalty coefficient  $\lambda$  must be a large positive number so that any deviation from the constraints is heavily suppressed.

# Ising model

3/5/2021 • 2 minutes to read • [Edit Online](#)

In statistical mechanics, the *Ising model* is a simplified representation of the interaction between individual magnetic moments in a ferromagnetic substance (for example, a fridge magnet). These moments, called *spins*, are presumed to point either "up" ( $+1$ ) or "down" ( $-1$ ) and interact with other spins depending on their relative position.



The behavior of such a magnetic material is temperature-dependent:

1. At high temperature, thermal excitations cause the spins to readily change orientation and there is little organization in the system.
2. As the temperature decreases, lower-energy states are favored. These are attained when neighboring spins agree, causing small aligned domains to form.
3. Once this alignment translates to a substantial part of the system, the individual moments add up to an overall magnetic field.

## Ising Hamiltonian

The energy of an Ising system is described by the *Hamiltonian*

$$\mathcal{H} = -\sum_{\langle i,j \rangle} J \sigma_i \sigma_j$$

where  $\sigma_i$  are the spin variables, the sum is over interacting ("neighboring") spin pairs and  $J > 0$  is an interaction constant (describing how strongly neighboring spins interact). For each pair of spins which are aligned, we get a contribution of  $-J$  to the overall energy.

### NOTE

The Ising Model has no disorder in the interaction; at low temperature all spins have a tendency to align with their respective neighbors and the lowest-energy state ("ground state") is attained when all spin variables have the same value (for example, they are either all  $+1$  or all  $-1$ ).

## Dynamics of the system

The probability of finding statistical-mechanical system in a specific state is given by the Boltzmann distribution. It depends on the energy of this state (compared to that of all other possible states) and the temperature:

$$p(\vec{\sigma}) = \frac{e^{-\mathcal{H}(\vec{\sigma})/k_B T}}{\sum_s e^{-\mathcal{H}(s)/k_B T}}$$

where the normalizing sum in the denominator is over all states ("partition function"),  $k_B$  is the

Boltzmann constant (typically simplified to  $k_B=1$  in theoretical models) and  $T$  is the current temperature (often expressed as the inverse temperature  $\beta = 1/T$ ).

#### NOTE

For the purpose of optimization, the key detail to note is that for  $T \rightarrow 0$  ( $\beta \rightarrow \infty$ ), the lowest-energy state dominates the sum and the chance of finding the system in this ground state tends to  $p(\vec{\sigma}_{\text{GS}}) \rightarrow 1$ .

## Disordered Ising systems

While the (ferromagnetic) Ising model offers some interesting dynamics, the system becomes more complex if the interaction constants  $J$  are allowed to be bond-dependent (for example, we use a different constant for each set of interacting spins).

$$\mathcal{H} = -\sum_{\langle ij \rangle} J_{ij} \sigma_i \sigma_j.$$

For these so-called *Ising Spin Glasses*,

- A term with  $J_{ij} < 0$  represents anti-ferromagnetic interaction: Spins which favor anti-alignment (alternating  $+1$  and  $-1$  spin values).
- A selection of  $J_{ij}$ 's with different signs (and possibly magnitudes), along with the interaction graph, can lead to *competing interactions*: A situation where no spin-variable assignment satisfies all the bonds (this is known as *frustration*).
- When finding the ground state, the choice of *how* to assign variables with competing interactions can have a ripple effect across the system (by changing how adjacent spins must be arranged, in turn impacting their respective neighbors).

As a result, finding the ground state is much more challenging in these complex systems.

# Cost functions

3/5/2021 • 4 minutes to read • [Edit Online](#)

An *optimization problem* is described by a set of *variables*, each having a set (or range) of possible values. They describe the decisions that the optimizer must make.

A *solution* assigns a value to each of these variables. These describe the choice for each of the aforementioned decisions.

The *cost function* associates a numerical value ("score") with each possible solution in order to compare them and select the most favorable one (typically identified by the lowest cost value).

## NOTE

In physics, the *Hamiltonian* takes the role of the cost function and its cost value is referred to as the *energy* of the system. Each choice of variable values is called a *state* and the lowest-energy state is the *ground state*.

## Implementation

In general, the cost function implementation could defer to a full reference table, a black box implementation or even external input. However, a frequent approach is to define it as a *mathematical expression* of the problem's variables and parameters.

**Example:** Find a fraction of integers  $x$ ,  $y$  which is close to  $\pi$ .

- Two variables:  $x$ ,  $y$  (integers  $\in [1..100]$ )
- Cost function:  $\text{cost} = |\frac{x}{y} - \pi|$  (we want the cost to be minimal when  $x/y \approx \pi$ ).
- Possible solutions:  $[1..100] \times [1..100]$  (independent choices of  $x$ ,  $y$ )
- Simple approximation:  $x=3, y=1 \Rightarrow \text{cost} = |\frac{3}{1} - \pi| = 0.14159$
- Best solution in this value range:  $x=22, y=7 \Rightarrow \text{cost} = |\frac{22}{7}| \approx 3.14286$

## NOTE

It is not required for the optimal solution of the cost function to have a  $\text{cost}=0$ .

## Constraints

A *constraint* is a relation between multiple variables which must hold for a solution to be deemed valid.

Solutions which violate constraints can either be assigned a very high cost ("penalty") by the cost function or be excluded from sampling explicitly by the optimizer.

**Example:** In the above problem of finding a fraction close to  $\pi$ , multiplying both  $x$  and  $y$  with the same number yields an equally optimal solution (for example,  $44/14$ ). We can avoid this by adding a penalty term for non-simplified fractions:

$$\text{cost} = |\frac{x}{y} - \pi| + \underbrace{100(\gcd(x,y)-1)}_{\text{penalty}}$$

where  $\text{gcd}(x,y)$  is the greatest common divisor of  $x$  and  $y$ , such that the term in parentheses vanishes for simplified fractions. With this addition, the optimal solution,  $22/7$ , is unique.

#### NOTE

Constraints on individual variables are typically incorporated into their respective set of allowed values rather than a constraint.

## Parameterized models

Typical optimization problems consist of many variables and several terms constituting the cost function. It is therefore pertinent to select a specific structure for the mathematical expression, while denoting merely the parameters and variable locations required to construct the cost function.

**Example:** Divide a set of  $N$  numbers into two groups of equal sum.

- Input Parameters:  $w_0..w_{N-1}$  the numbers in the set
- $N$  Variables:  $x_0..x_{N-1}$  denoting whether the  $i$ -th number is in the first ( $x_i=+1$ ) or second group ( $x_i=-1$ ).
- Model cost function:  $\text{cost} = \left| \sum_i w_i x_i \right|$

That is, we always construct a cost function of the form in the third bullet, but we adjust the parameters  $w_i$  according to the specific problem instance we are solving.

For instance, the numbers  $[18, 19, 36, 84, 163, 165, 243]$  would result in the cost function

$$\text{cost} = |18x_0 + 19x_1 + 36x_2 + 84x_3 + 163x_4 + 165x_5 + 243x_6|$$

#### NOTE

This instance has only two solutions with  $\text{cost}=0$  (one mirroring the other). Can you find them?

## Supported models

Models implemented in our optimizers include the [Ising Model](#), and Quadratic/Polynomial unconstrained [binary optimization](#) problems. These support versatile applications because several other optimization problems can be mapped to them.

**Example:** For the above number set division problem, one can substitute the absolute value with the square operator (which also has its lowest value at 0) to obtain:

$$\text{cost}' = (\sum_i w_i x_i)^2 = \sum_{ij} w_i w_j x_i x_j$$

When multiplied out, this cost function has more terms than the previous one, but it happens to be in the (polynomial) form supported by our optimizers (namely, an Ising cost function).

### Ising cost function

Ising variables take the values  $x_i \in \{\pm 1\}$  and the parameterized Ising cost function has the form:

$$\text{cost} = \sum_k \text{term}_k = \sum_k c_k \prod_i x_i$$

The parameters `c` and the `ids` of the variables  $x_i$  participating in each term  $k$  are listed as part of the input:

For instance, the input:

```

"cost_function": {
  "type": "ising",
  "version": "2.0",
  "terms": [
    { "c": 3, "ids": [0, 1, 2] },
    { "c": -2, "ids": [0, 3] },
    { "c": 1, "ids": [2, 3] }
  ]
}

```

describes the an Ising cost function with three terms:  $\text{cost} = 3x_0x_1x_2 - 2x_0x_3 + x_2x_3$

#### NOTE

An empty `ids` array (constant term) or with a single value ("local field") is allowed, but the same variable `id` cannot be repeated within a term.

#### Caution

The definition of the Ising cost function differs from the canonical Ising model Hamiltonian  $H = \sum_{ij} J_{ij} \sigma_i \sigma_j$  typically employed in statistical mechanics (by a global sign). As a result negative term constants  $c_k$  result in *ferromagnetic* interaction between two variables.

#### PUBO cost function

For binary optimization problems, variables take the values  $x_i \in \{0, 1\}$  and the cost function has the form:

$$\text{cost} = \sum_k \text{term}_k = \sum_k c_k \prod_i x_i$$

For instance, the input:

```

"cost_function" {
  "type": "pubo",
  "version": "2.0",
  "terms": [
    { "c": 3, "ids": [0, 1, 2] },
    { "c": -2, "ids": [0, 3] },
    { "c": 1, "ids": [2, 3] }
  ]
}

```

describes a PUBO cost function with 3 terms:  $\text{cost} = 3x_0x_1x_2 - 2x_0x_3 + x_2x_3$

Though the form of the cost function is identical to the Ising case, this describes a different optimization problem (the set of allowed variable values is different:  $\{0, 1\}$  vs  $\{\pm 1\}$ ).

#### NOTE

PUBO and QUBO are handled by the same cost function; there is no separate `"qubo"` identifier. Quadratic binary optimization problems are a special case of a PUBO where each `ids` array has length at most 2.

# List of targets on Azure Quantum

6/2/2021 • 3 minutes to read • [Edit Online](#)

Azure Quantum offers optimization targets to solve binary optimization problems on classical CPUs, or hardware accelerated on field-programmable gate arrays (FPGA), GPUs or hardware annealers.

## NOTE

Optimization targets can't run Q# applications or any other type of quantum computing program. Optimization solvers are optimization algorithms that run on specialized classical hardware.

## Provider: 1Qbit



### **1QBit Tabu Search Solver**

An iterative heuristic algorithm that uses local search techniques to solve a QUBO problem. It starts from a random solution and looks for an improved solution in the solution's neighborhood which includes all possible single flips. The algorithm stops when it reaches a stopping criterion, such as a specified number of consecutive iterations without improvement.

For more information, go to the [1QBit provider reference page](#).

### **1QBit PTICM Solver**

The parallel tempering with isoenergetic cluster moves (PTICM) solver is a Monte Carlo approach to solving QUBO problems. In this algorithm, multiple replicas of the original system, each with a different initial state, are simulated at different temperatures simultaneously. The replicas at neighboring temperatures are periodically swapped based on a Metropolis criterion. These swaps allow different replicas to do a random walk in the temperature space, thereby, efficiently overcoming energy barriers.

For more information, go to the [1QBit provider reference page](#).

### **1QBit Path-Relinking Solver**

The path-relinking algorithm is a heuristic algorithm that uses the tabu search as a subroutine to solve a QUBO problem. The algorithm starts from a set of elite solutions found by the tabu search. It then constructs a path between each pair of elite solutions, selects one of the solutions along the path, and repeats the tabu search. If the tabu search finds a distinct solution that is better than the current worst elite solution, the elite solutions set is updated with the new improved solution. This whole procedure is repeated until the algorithm meets a stopping condition.

For more information, go to the [1QBit provider reference page](#).

## Provider: Microsoft QIO



For an overview of the Microsoft QIO solvers available, please refer to the [Microsoft QIO overview page](#).

### **Simulated Annealing**

Rephrases the optimization problem as a thermodynamic system and considers the energy of a single system. Changes to the system are accepted if they decrease the energy or meet a criterion based on decreasing temperature. This target can be run on CPU or FPGA hardware. For more information, go to the [Microsoft QIO provider reference page](#).

### **Population Annealing**

Aims to alleviate the susceptibility of the Metropolis Algorithm to rough cost landscapes by simulating a population of metropolis walkers, which continuously consolidates search efforts around favorable states. This target is available on CPUs. For more information, go to the [Microsoft QIO provider reference page](#).

### **Parallel Tempering**

Rephrases the optimization problem as a thermodynamic system and runs multiple copies of a system, randomly initialized, at different temperatures. Then, based on a specific protocol, exchanges configurations at different temperatures to find the optimal configuration. This target is available on CPUs only. For more information, go to the [Microsoft QIO provider reference page](#).

### **Tabu search**

Tabu Search looks at neighboring configurations. It can accept worsening moves if no improving moves are available and prohibit moves to previously-visited solutions. For more information, go to the [Microsoft QIO provider reference page](#).

### **Quantum Monte Carlo**

Quantum Monte Carlo is a Metropolis annealing algorithm, similar in concept to simulated annealing that starts at a low temperature and improves the solution by searching across barriers with some probability as an external perturbation is applied to the system. As this external field is varied over every Monte Carlo step, the configuration may be able to tunnel through energy barriers and evolve towards a desired ground state (without possessing the thermal energy needed to climb the barriers, as would be required in simulated annealing). For more information, go to the [Microsoft QIO provider reference page](#).

### **Substochastic Monte Carlo**

Substochastic Monte Carlo is a diffusion Monte Carlo algorithm inspired by adiabatic quantum computation. It simulates the diffusion of a population of walkers in search space, while walkers are removed or duplicated based on how they perform according the cost function. This target is available on CPUs. For more information, go to the [Microsoft QIO provider reference page](#).

# 1QBit provider

6/16/2021 • 5 minutes to read • [Edit Online](#)

- Publisher: [1QBit](#)
- Provider ID: [1qbit](#)

## Targets

### Tabu Search Solver

An iterative heuristic algorithm that uses local search techniques to solve a QUBO problem. It starts from a random solution and looks for an improved solution in the solution's neighborhood which includes all possible single flips. The algorithm stops when it reaches a stopping criterion, such as a specified number of consecutive iterations without improvement.

Please note for the parameters values, they are required to be of JSON string type. The types listed below are the types the solvers expect within the string value. For example, 'improvement\_cutoff' is listed as type int, it is expected to be passed in in the format:

improvement\_cutoff: "5"

- Job type: [Quantum-Inspired Optimization Problem](#)
- Data Format: [microsoft.qio.v2](#)
- Target ID: [1qbit.tabu](#)
- Python Solver class name: [TabuSearch](#)

PARAMETER NAME	TYPES	REQUIRED	DESCRIPTION
<a href="#">improvement_cutoff</a>	int	Optional	The number of iterations that the solver attempts with no improvement before stopping. Default: 0
<a href="#">improvement_tolerance</a>	double	Optional	The tolerance value that determines if a solution is an improvement over the previous iteration. Default: 1e-9
<a href="#">tabu_tenure</a>	int	Optional	The tenure prevents a flipped variable from being flipped again during the iterations. Default: 0
<a href="#">tabu_tenure_rand_max</a>	int	Optional	The upper limit of the exclusive range of random integers. Valid value range: 1 to 200,000. Default: 0

PARAMETER NAME	TYPE	REQUIRED	DESCRIPTION
timeout	int	Optional	The duration in ms the solver runs before exiting. If the value is set to 0, it does not time out. Default: 0

## PTICM Solver

The parallel tempering with isoenergetic cluster moves (PTICM) solver is a Monte Carlo approach to solving QUBO problems. In this algorithm, multiple replicas of the original system, each with a different initial state, are simulated at different temperatures simultaneously. The replicas at neighboring temperatures are periodically swapped based on a Metropolis criterion. These swaps allow different replicas to do a random walk in the temperature space, thereby, efficiently overcoming energy barriers.

- Job type: `Quantum-Inspired Optimization Problem`
- Data Format: `microsoft.qio.v2`
- Target ID: `1qbit.pticm`
- Python Solver class name: `PticmSolver`

PARAMETER NAME	TYPE	REQUIRED	DESCRIPTION
auto_set_temperatures	boolean	Optional	This defines whether the temperature parameters are auto-calculated or not. Set it to True for auto-calculating and False for customizing the temperature parameters. Default: True
elite_threshold	double	Optional	The fraction of the best solutions used for the var_fixing_type parameter with value SPVAR. Default: 0.3
frac_icm_thermal_layers	double	Optional	The fraction of temperatures for the iso-energetic cluster moves. To change this value, set the perform_icm parameter to True. Default: 0.5
frac_sweeps_fixing	double	Optional	The fraction of sweeps used for fixing the QUBO variables. Default: 0.15
frac_sweeps_idle	double	Optional	The fraction of sweeps to wait before fixing the QUBO variables. Default: 1.0
frac_sweeps_stagnation	double	Optional	The fraction of sweeps without improvement that triggers a restart. Default: 1.0

PARAMETER NAME	TYPE	REQUIRED	DESCRIPTION
goal	string	Optional	This defines whether the solver is used for optimizing or sampling. Valid values: "OPTIMIZE" or "SAMPLE" Default: OPTIMIZE
high_temp	double	Optional	The highest temperature of a replica. Set the auto_set_temperatures parameter to False to use this feature. Default: 2
low_temp	double	Optional	The lowest temperature of a replica. Set the auto_set_temperatures parameter to False to use this feature. Default: 0.2
max_samples_per_layer	int	Optional	The maximum number of samples collected per replica. Default: 10
max_total_sweeps	int	Optional	The total number of sweeps before termination. Default: num_sweeps_per_run * 10
manual_temperatures	array[double]	Optional	An array of a custom temperature schedule which includes the high, intermediate, and low temperature values for the replicas. Set the auto_set_temperatures parameter to False to use this feature.
num_elite_temps	int	Optional	The number of elite temperatures used for fixing the variables with persistency. Default = 4
num_replicas	int	Optional	The number of replicas at each temperature. Default: 2
num_sweeps_per_run	int	Optional	The number of Monte Carlo sweeps. Default: 100
num_temps	int	Optional	The number of temperatures including the highest and the lowest temperatures. Set the auto_set_temperatures parameter to False to use this feature. Default: 30

PARAMETER NAME	TYPE	REQUIRED	DESCRIPTION
perform_icm	boolean	Optional	This defines whether or not to perform the isoenergetic cluster moves. Default: true
scaling_type	string	Optional	This defines whether the QUBO problem is automatically scaled or not. MEDIAN means it's automatically scaled, and NO_SCALING means it's not. Valid values: "MEDIAN" or "NO_SCALING"
var_fixing_type	string	Optional	This decides whether the values of QUBO variables are fixed or not. You can fix them with PERSISTENCY or SPVAR types. NO_FIXING means the variables are not fixed. If you choose PERSISTENCY or SPVAR, also set the solver.fraction_sweeps_fixing and solver.fraction_sweeps_idle parameters to a number less than one. Valid values: "PERSISTENCY", "SPVAR" or "NO_FIXING" Default: NO_FIXING

### Path-Relinking Solver

The path-relinking algorithm is a heuristic algorithm that uses the tabu search as a subroutine to solve a QUBO problem. The algorithm starts from a set of elite solutions found by the tabu search. It then constructs a path between each pair of elite solutions, selects one of the solutions along the path, and repeats the tabu search. If the tabu search finds a distinct solution that is better than the current worst elite solution, the elite solutions set is updated with the new improved solution. This whole procedure is repeated until the algorithm meets a stopping condition.

- Job type: `Quantum-Inspired Optimization Problem`
- Data Format: `microsoft.qio.v2`
- Target ID: `1qbit.pathrelinking`
- Python Solver class name: `PathRelinkingSolver`

PARAMETER NAME	TYPE	REQUIRED	DESCRIPTION
distance_scale	double	Optional	The minimum distance from the initiating and guiding solutions for constructing the candidate solution list. The highest quality solution in the candidate solution list is then selected for improvement. Valid values: 0.0 to 0.5 Default 0.33

PARAMETER NAME	TYPE	REQUIRED	DESCRIPTION
greedy_path_relinking	boolean	Optional	When you use the path-relinking solver there are two ways you can generate a path that leads to the solution: one is the greedy function and the other operates in a random matter. If you set the this parameter to true, the solver will use the greedy function. If you set it to false, it will use the random method. Default: false
ref_set_count	int	Optional	The number of initial elite solutions to be generated by the tabu search algorithm. Valid values: Greater than 1 Default: 10
timeout	int	Optional	The duration in ms the solver runs before exiting. If the value is set to 0, it does not time out. If a value is not specified, the solver uses the default value or estimates a new one based on the input problem. Estimated values are marked as "(COMPUTED)". Default: 0

# Support Policy for 1QBit in Azure Quantum

5/27/2021 • 2 minutes to read • [Edit Online](#)

This article describes the Microsoft support policy that applies when you use the 1QBit provider in Azure Quantum. The article applies to any of the targets under this provider.

If you are using the 1QBit provider and hit any unexpected issues that you cannot troubleshoot yourself, you can usually contact the Azure Support team for help by [creating an Azure support case](#).

There are however some situations, where the Azure Support team will need to redirect you to 1QBit's support team, or where you may receive a quicker response by reaching out to 1QBit directly. This article aims to provide more detail on this based on some frequently asked questions.

## Frequently asked questions

### **Q: What is the support policy for using 1QBit offerings through Azure Quantum?**

Microsoft will provide support for the Azure Platform and the Azure Quantum service only. Support for 1QBit hardware, simulators, and other products and targets will be provided directly by 1QBit. For more information about Azure support, [see Azure support plans](#). For information about 1QBit support, please [visit the 1QBit Support website](#).

### **Q: What happens if I raise a support issue with the Azure support team and it turns out that a third party provider (like 1QBit) needs to troubleshoot the issue further?**

The support engineer will create a redirection package for you. This is a PDF document that contains information about your case which you can provide to the 1QBit support team. The support engineer will also give you advice and guidance on how to reach out to 1QBit to continue troubleshooting.

### **Q: What happens if I raise a support issue with the 1QBit team and it turns out that there is an issue with the Azure Quantum service?**

The 1QBit support team will help you reach out to Microsoft and provide you with a redirection package. This is a PDF document that you can use when continuing your support enquiry with the Azure support team.

### **Third-party information disclaimer**

The third-party products that this article discusses are manufactured by companies that are independent of Microsoft. Microsoft makes no warranty, implied or otherwise, about the performance or reliability of these products.

### **Third-party contact disclaimer**

Microsoft provides third-party contact information to help you find additional information about this topic. This contact information may change without notice. Microsoft does not guarantee the accuracy of third-party contact information.

# Microsoft QIO provider

6/4/2021 • 5 minutes to read • [Edit Online](#)

The Microsoft QIO provider is enabled in every Quantum workspace.

- Publisher: [Microsoft](#)
- Provider ID: `microsoft`

## Targets

The Microsoft QIO provider makes the following targets available:

- [Solver: Simulated Annealing \(Parameter-free\)](#)
- [Solver: Simulated Annealing \(Parameter-free - FPGA\)](#)
- [Solver: Simulated Annealing \(Parameterized\)](#)
- [Solver: Simulated Annealing \(Parameterized - FPGA\)](#)
- [Solver: Population Annealing \(Parameterized\)](#)
- [Solver: Parallel Tempering \(Parameter-free\)](#)
- [Solver: Parallel Tempering \(Parameterized\)](#)
- [Solver: Tabu Search \(Parameter-free\)](#)
- [Solver: Tabu Search \(Parameterized\)](#)
- [Solver: Quantum Monte Carlo \(Parameterized\)](#)
- [Solver: Substochastic Monte Carlo \(Parameterized\)](#)

## Target Comparison

In the following table you can find a brief comparison between the available targets:

NAME	DESCRIPTION	BEST APPLICABLE SCENARIO
Parallel Tempering	Rephrases the optimization problem as a thermodynamic system and runs multiple copies of a system, randomly initialized, at different temperatures. Then, based on a specific protocol, exchanges configurations at different temperatures to find the optimal configuration.	<ul style="list-style-type: none"><li>• Generally outperforms Simulated Annealing on hard problems with rugged landscapes</li><li>• Very good at solving Ising problems</li></ul>
Simulated Annealing	Rephrases the optimization problem as a thermodynamic system and considers the energy of a single system. Changes to the system are accepted if they decrease the energy or meet a criterion based on decreasing temperature.	<ul style="list-style-type: none"><li>• Convex landscapes</li></ul>

Name	Description	Best Applicable Scenario
Population Annealing	Aims to alleviate the susceptibility of the Metropolis Algorithm to rough cost landscapes by simulating a population of metropolis walkers, which continuously consolidates search efforts around favorable states.	<ul style="list-style-type: none"> <li>We recommend Population Annealing for both sparse and dense graphs.</li> <li>The algorithm might not be suitable for constraint problems with large penalty terms.</li> </ul>
Quantum Monte Carlo	Similar to Simulated Annealing but the changes are by simulating quantum-tunneling through barriers rather than using thermal energy jumps.	<ul style="list-style-type: none"> <li>Optimization landscape has tall and thin barriers</li> <li>Due to its large overhead, is useful for small hard problems</li> </ul>
Substochastic Monte Carlo	Substochastic Monte Carlo is a diffusion Monte Carlo algorithm inspired by adiabatic quantum computation. It simulates the diffusion of a population of walkers in search space, while walkers are removed or duplicated based on how they perform according the cost function.	<ul style="list-style-type: none"> <li>The algorithm is suitable for rough optimization landscapes where Simulated Annealing or Tabu Search might return a diverse set of solutions.</li> </ul>
Tabu Search	Tabu Search looks at neighboring configurations. It can accept worsening moves if no improving moves are available and prohibit moves to previously-visited solutions	<ul style="list-style-type: none"> <li>Convex landscapes, high density problems, QUBO problems.</li> </ul>

## FPGA vs. CPU

For some solvers we offer two versions: an unlabeled version that runs on traditional CPUs and a labeled FPGA version. In the following table you can see the pros and cons of using FPGA solvers:

Pros/Cons	FPGA Solvers
Pros	<ul style="list-style-type: none"> <li>FPGA solvers run on highly optimized hardware that enables algorithms to parallelize very efficiently. This can achieve a significant performance gain when comparing CPU and FPGA solvers.</li> <li>FPGA solver use very condensed memory representation.</li> <li>This means that problems with a large number of terms may fail on a CPU solver due to a lack of memory, but run on an FPGA implementation of that solver.</li> </ul>
Cons	<ul style="list-style-type: none"> <li>Our FPGA solvers support up to 65535 variables. This is a hard limitation.</li> <li>To achieve the best performance, FPGA solvers use 32-bit floating point operations.</li> <li>As a result, the accuracy of FPGA solvers is a little lower than for CPU solvers.</li> </ul>

## FPGA Regional Availability

FPGA-based solvers are only available in a limited set of Azure regions. When creating your Azure Quantum

workspace you can see if FPGA targets are available in the region that you have selected by accessing the Microsoft QIO provider blade on the Create screen. Regions that offer access to FPGA solvers will show "FPGA simulated annealing" in their list of available targets.

For existing workspaces you can check the "Providers" blade. Select "Modify" to view your Microsoft QIO SKU. If your workspace is in a region where FPGA solvers are available "FPGA simulated annealing" will show up in the list of targets.

#### Recommendations for FPGA solvers

FPGA solvers use the same parameters as their corresponding CPU solvers, but for the best performance, please tune the parameters of FPGA solvers, instead of just directly using CPU solvers' parameters. For example, in FPGA solvers, we build about 200 parallel pipelines, and each pipeline can handle one restart, so the restarts of FPGA shall be no less than 200.

FPGA solvers have an initialization time that may take a large percentage of the total runtime for small problems. If your problem can be solved on a CPU solver within a number of seconds, then you will likely not see a performance gain by switching to FPGA. We recommend using FPGA solvers when the execution timing on CPU is at least a couple minutes.

## Pricing

For the most up-to-date pricing information on Microsoft's QIO offering, please refer to the Providers tab of your workspace on the [Azure portal](#) or visit the [Azure Quantum pricing page](#).

## General advice for Microsoft QIO solvers

Here are some things to keep in mind when using our QIO solvers, and steps you can take to improve performance in certain cases. Note that other providers might have different requirements and recommendations specific to their solutions. The advice below applies to the terms that [represent your problem](#) and [cost function](#). Remember that a term is composed of a coefficient  $c$  and a set of indices  $\{i\}$ .

### 1. Remove coefficients that exceed the computational precision:

If the ratio of largest to smallest coefficient is greater than  $2^{64}$ , the term with the small coefficient will likely not be taken into account and should be removed. In other words, you should remove any terms with coefficients  $|c_i| < \frac{\max(|c_j|)}{2^{64}}$ .

### 2. Merge duplicate terms:

If you automatically generate your terms, you may encounter some that are duplicates of each other, that is, they contain the same set of decision variables/indices. Avoiding duplicate terms will increase the performance of the solver as it has to handle fewer terms.

Multiple terms with the same set of variables should be merged into a single term by adding up the coefficients. For example,  $3 \cdot x_2 x_4 x_1$  and  $2 \cdot x_1 x_4 x_2$  can be merged into the single term  $5 \cdot x_1 x_2 x_4$ .

### 3. Use integer coefficients:

Whenever possible, you should use integers for your coefficients over floating point numbers, as these will provide greater precision.

# Simulated annealing

6/8/2021 • 5 minutes to read • [Edit Online](#)

[Simulated annealing](#) is a Monte Carlo search method named from the heating-cooling methodology of metal annealing. The algorithm simulates a state of varying temperatures where the temperature of a state (in our implementation, represented by parameter beta - the inverse of temperature with the Boltzmann constant set to 1 ( $\beta = 1 / T$ )) influences the decision making probability at each step.

In the context of optimization problems, the algorithm starts at an initial high temperature state (low beta, or `beta_start`) where "bad" moves in the system are accepted with a higher probability, and slowly "cools" on each sweep until the state reaches the lowest specified temperature (high beta, or `beta_stop`). At lower temperatures, moves that do not improve the objective value are less likely to be accepted.

When sweeping for a binary problem, each decision variable is "flipped" based on the objective value impact of that flip. Flips that improve the objective value will be accepted automatically, while moves that do not improve the objective value are accepted on a probabilistic basis, calculated via the [Metropolis Criterion](#).

## Features of simulated annealing on Azure Quantum

Simulated annealing in Azure Quantum supports:

- Parameter-free mode and parameterized mode (with parameters)
- Ising and PUBO input formats
- CPU & FPGA hardware (see below for instructions on how to use both)

## When to use simulated annealing

Simulated annealing is a standard jack-of-all-trades algorithm that performs well on many kinds of problems. It is recommended to start with this algorithm as it will reliably produce good quality and fast solution to most problems.

It is also a good algorithm for larger problems (thousands of variables).

### NOTE

For further information on determining which solver to use, refer to [Which optimization solver should I use?](#).

## Parameter-free simulated annealing (CPU)

The parameter-free version of simulated annealing is recommended for new users, those who don't want to manually tune parameters (especially betas), and even as a starting point for further manual tuning. The main parameters to be tuned for this solver are the number of `sweeps`, `beta_start` and `beta_stop` (described in the next section).

The parameter-free solver will halt either on `timeout` (specified in seconds) or when there is sufficient convergence on a solution. A seed can be supplied to reproduce results.

PARAMETER NAME	DESCRIPTION
----------------	-------------

PARAMETER NAME	DESCRIPTION
<code>timeout</code>	Max execution time for the solver (in seconds). This is a best effort mechanism, so the solver may not stop immediately when the timeout is reached.
<code>seed</code> (optional)	Seed value - used for reproducing results.

To create a parameter-free simulated annealing solver for the CPU platform using the SDK:

```
from azure.quantum.optimization import SimulatedAnnealing
# Requires a workspace already created.
solver = SimulatedAnnealing(workspace, timeout=100, seed=22)
```

The parameter-free solver will return the parameters used in the result JSON. You can then use these parameters to solve similar problems (similar number of variables, terms, locality and similar coefficient scale) using the parameterized simulated annealing solver.

Running the solver without any parameters also triggers the parameter-free version:

```
from azure.quantum.optimization import SimulatedAnnealing
# Requires a workspace already created.
# Not specifying any parameters runs the parameter-free version of the solver.
solver = SimulatedAnnealing(workspace)
```

## Parameterized simulated annealing (CPU)

Simulated annealing with specified parameters is best used if you are already familiar with simulated annealing terminology (sweeps, betas) and/or have an idea of which parameter values you intend to use. **If this is your first time using simulated annealing for a problem, the parameter-free version is recommended.** Some of the parameters like `beta_start` and `beta_stop` are hard to estimate without a good starting point.

Simulated annealing supports the following parameters:

PARAMETER NAME	DESCRIPTION
<code>sweeps</code>	Number of sets of iterations to run over the variables of a problem. More sweeps will usually improve the solution (unless it is already at the global min).
<code>beta_start/beta_stop</code>	Represents the starting and stopping betas of the annealing schedule. A suitable value for these parameters will depend entirely on the problem and the magnitude of its changing moves. In general a non-zero and declining acceptance probability is sufficient.
<code>restarts</code>	The number of repeats of the annealing schedule to run. Each restart will start with a random configuration <b>unless an initial configuration is supplied in the problem file</b> . The restarts will be executed in parallel and split amongst the threads of the VM. Recommended to set this value to at least 72.
<code>seed</code> (optional)	Seed value - used for reproducing results

To create a parameterized simulated annealing solver for the CPU platform using the SDK:

```
from azure.quantum.optimization import SimulatedAnnealing
# Requires a workspace already created.
solver = SimulatedAnnealing(workspace, sweeps=2, beta_start=0.1, beta_stop=1, restarts=72, seed=22)
```

## Simulated annealing (FPGA)

The simulated annealing solver is also available on FPGA hardware for both parameter and parameter-free modes. The parameters are the same as in the CPU versions.

This section will describe the advantages and disadvantages of using the FPGA solver. Users are encouraged to make an informed decision based on their own problem features.

### When to use FPGA simulated annealing

FPGA solvers have some built-in costs like PCIe transfers, FPGA device initialization etc. If an optimization problem is too small (for example, the CPU execution takes seconds), then the built-in cost of FPGA hardware will be the bulk of the cost, with minimal solution benefits. Thus, it is recommended to use FPGA solvers when the execution time on the CPU is minutes or higher.

### Advantages of FPGA simulated annealing

- Highly optimized for parallelization. Compared to the equivalent CPU simulated annealing solver with the same parameters, the FPGA simulated annealing solver is on average 10 times faster.
- Highly condensed memory representation. A problem with a large number of terms may fail on a CPU solver due to memory limits, but may fit on FPGA hardware.

The performance gain may not be obvious for the parameter-free mode because both algorithms are gated by a `timeout` setting (which halts most problems). However on FPGA hardware, the solver most likely will run many more sweeps in the same amount of time than on the CPU.

### Limitations of FPGA simulated annealing

- the FPGA solver supports up to **65535 variables**, and this is a hard limitation. This number is limited by the available DRAM, and this is generally not an issue for FPGA (since most problems are smaller than 65535).
- For best performance, FPGA solvers on Azure Quantum use 32 bit floating-point operations. Because of this, the computation accuracy of FPGA solvers is a little lower than that of the CPU solvers.

### Parameter guide for FPGA simulated annealing

FPGA simulated annealing uses the same parameters as the corresponding CPU solver, but it is still recommended to tune the parameters of FPGA solvers separately instead of using pre-tuned parameters from CPU solvers.

It is also recommended to set the number of restarts to at least 216 if using the FPGA solver, as it can support a higher degree of parallelization.

To create a simulated annealing solver for the FPGA platform using the SDK, simply specify the platform option as follows:

```
from azure.quantum.optimization import SimulatedAnnealing, HardwarePlatform
# Requires a workspace already created.
solver = SimulatedAnnealing(workspace, timeout=100, seed=22, platform=HardwarePlatform.FPGA)
```

The `timeout` and `seed` parameters are optional.

For the parameterized version:

```
from azure.quantum.optimization import SimulatedAnnealing, HardwarePlatform
# Requires a workspace already created.
solver = SimulatedAnnealing(workspace, sweeps=2, beta_start=0.1, beta_stop=1, restarts=72, seed=22,
platform=HardwarePlatform.FPGA)
```

# Population Annealing

6/17/2021 • 3 minutes to read • [Edit Online](#)

[Population Annealing](#) is a sequential Monte Carlo method which aims to alleviate the susceptibility of the Metropolis Algorithm to rough cost landscapes (i.e., with many local minima) by simulating a population of metropolis walkers. Akin to [Simulated Annealing](#), the algorithm proceeds over a set of decreasing temperatures  $\$T\$$  (or increasing  $\beta = 1/T$ ), but the population is resampled at each temperature step. During resampling, some walkers are removed and some are duplicated (with a bias towards retaining lower cost walkers). This has the effect of continuously consolidating search efforts around favorable states. In this sense, Population Annealing shares many features with evolutionary-type algorithms.

Intuitively, one can picture Population Annealing as a set of walkers spread across the configuration space of the problem. Each walker is free to explore its own neighborhood. Once a walker finds a better state, the rest of the population is gravitated toward that state. Therefore, the algorithm is designed to take advantage of the "collective knowledge" of the walkers to find its way toward the solution.

In the context of optimization problems, the algorithm starts with an initial population of random states at high temperature. In this regime, "bad" transitions are still accepted with a high probability and the bias towards lower-cost walkers in the resampling is weak. At lower temperatures, the [Metropolis Criterion](#) strongly favors transitions which decrease the cost and resampling progressively weeds out non-competitive states. The removal rate during resampling depends on the size of the individual temperature steps. Therefore, the temperature schedule must be slow enough such that a reasonable number of walkers (about 10%-20%) are dropped while roughly the same number of them are copied to keep the population size stable.

Eventually, all walkers are resampled into the same lowest-cost state discovered ("population collapse"). At this stage it is more efficient to restart the process with a new set of random walkers ([restarts](#)).

## Features of Population Annealing on Azure Quantum

Population Annealing in Azure Quantum supports:

- Parameterized mode
- Ising and PUBO input formats

## When to use Population Annealing

Generally speaking, given enough resources, Population Annealing can solve any problem that [Simulated Annealing](#) is used for more efficiently. However, due to the memory footprint of multiple walkers, Population Annealing is most suitable for very hard moderately-sized problems. We recommend Population Annealing for both sparse and dense graphs. The algorithm might struggle for constraint problems with large penalty terms.

### NOTE

For further information on choosing which solver to use, please refer to [this document](#).

## Parameterized Population Annealing

Suitable values for the `population` size and the annealing schedule `beta`, will depend entirely on the problem and the magnitude (cost difference) of its variable changes.

A good starting point for Population Annealing is to use the beta range from [Simulated Annealing](#) (`beta_start` and `beta_stop`) for the annealing schedule while renaming the `restarts` parameter to `population`.

Population Annealing supports the following parameters:

PARAMETER NAME	DEFAULT VALUE	DESCRIPTION
<code>sweeps</code>	10000	Number of sweeps. More sweeps will usually improve the solution (unless it is already at the global minimum).
<code>beta</code>	linear <code>0 .. 5</code>	Annealing schedule (beta must be increasing)
<code>population</code>	<i>number of threads</i>	The number of walkers in the population (must be positive).
<code>seed</code> (optional)	<i>time based</i>	Seed value - used for reproducing results.

To create a parameterized Population Annealing solver for the CPU using the SDK:

```
from azure.quantum.optimization import PopulationAnnealing, RangeSchedule  
# Requires a workspace already created.  
solver = PopulationAnnealing(workspace, sweeps=200, beta=RangeSchedule("linear", 0, 5), population=128,  
seed=42)
```

Running the solver without parameters will apply the default parameters shown in the table above. These default values are subject to change and we strongly recommend setting the values based on your problem rather than using the defaults.

# Parallel tempering

6/8/2021 • 3 minutes to read • [Edit Online](#)

Parallel tempering can be regarded as a variant of the [simulated annealing](#) algorithm, or more generally Monte Carlo Markov Chain methods. Azure Quantum's Parallel Tempering solvers are designed to solve binary optimization problems through random sampling.

As with simulated annealing, the cost function is explored through thermal jumps. Unlike simulated annealing, a cooling temperature is not used.

Instead of running a single copy of the system, Parallel Tempering creates multiple copies of a system, called replicas, that are randomly initialized and run at different temperatures. Then the same process is followed as in simulated annealing, but based on a specific protocol two replicas can be exchanged between different temperatures. This change can enable walkers that were previously stuck in local optima to be bumped out of them, and thus encourages a wider exploration of the problem space.

## NOTE

You can find further information on the parallel tempering algorithm in [Marinari and Parisi 1992 - Simulated Tempering: A New Monte Carlo Scheme](#)

## Features of parallel tempering on Azure Quantum

Parallel tempering in Azure Quantum supports:

- Parameter-free mode and parameterized mode (with parameters)
- Ising and PUBO input formats
- CPU only

## When to use parallel tempering

Parallel tempering generally outperforms [simulated annealing](#) on hard problems with rugged landscapes.

It is also very good at solving Ising problems, or problems that are equivalent (such as min-cut problems).

## NOTE

For further information on determining which solver to use, refer to [Which optimization solver should I use?](#).

## Parameter-free parallel tempering

The parameter-free version of parallel tempering is recommended for new users, those who don't want to manually tune parameters, and even as a starting point for further manual tuning. The main parameters to be tuned for this solver are the number of `sweeps`, `replicas` and `all_betas` (described in the next section).

The parameter-free solver will halt either on `timeout` (specified in seconds) or when there is sufficient convergence on a solution.

PARAMETER NAME	DESCRIPTION
<code>timeout</code>	Max execution time for the solver (in seconds). This is a best effort mechanism, so the solver may not stop immediately when the timeout is reached.
<code>seed</code> (optional)	Seed value - used for reproducing results.

To create a parameter-free parallel tempering solver using the SDK:

```
from azure.quantum.optimization import ParallelTempering
# Requires a workspace already created.
solver = ParallelTempering(workspace, timeout=100, seed=22)
```

The parameter-free solver will return the parameters used in the result JSON. You can then use these parameters to solve similar problems (similar number of variables, terms, locality and similar coefficient scale) using the parameterized parallel tempering solver.

Running the solver without any parameters also triggers the parameter-free version:

```
from azure.quantum.optimization import ParallelTempering
# Requires a workspace already created.
# Not specifying any parameters runs the parameter-free version of the solver.
solver = ParallelTempering(workspace)
```

## Parameterized parallel tempering

Parallel tempering with specified parameters is best used if you are already familiar with parallel tempering terminology (sweeps, betas) and/or have an idea of which parameter values you intend to use. **If this is your first time using parallel tempering for a problem, the parameter-free version is recommended.** Some of the parameters like `beta_start` and `beta_stop` are hard to estimate without a good starting point.

Parallel tempering supports the following parameters:

PARAMETER NAME	DESCRIPTION
<code>sweeps</code>	Number of sets of iterations to run over the variables of a problem. More sweeps will usually improve the solution (unless it is already at the global min).
<code>replicas</code>	The number of concurrent running copies for sampling. Each instance will start with a random configuration. We recommend this value to be no less than the number of cores available on the machine, in order to fully utilize the available compute power. Currently on Azure Quantum this should be no less than 72.
<code>all_betas</code>	The list of beta values used in each replica for sampling. The number of beta values must equal the number of replicas, as each replica will be assigned one beta value from the list. These beta values control how the solver escapes optimization saddle points - the larger the beta values, the less likely the sampling process will be to jump out of a local optimum.

PARAMETER NAME	DESCRIPTION
seed (optional)	Seed value - used for reproducing results

The larger the number of sweeps and replicas, the more likely the parallel tempering solver will be to find an optimal or near-optimal solution, however the solver will take longer to run.

To create a parameterized parallel tempering solver using the SDK:

```
from azure.quantum.optimization import ParallelTempering
# Requires a workspace already created.
solver = ParallelTempering(workspace, sweeps=2, all_betas=[1.15, 3.14], replicas=2, seed=22)
```

# Tabu search optimization solver

6/8/2021 • 3 minutes to read • [Edit Online](#)

Tabu Search is a neighborhood search algorithm that employs a tabu list. A tabu list represents a set of potential solutions that the search is forbidden to visit for a number of steps (tabu tenure). The decision making process per step is similar to that of a greedy algorithm, but with a list of forbidden moves (usually moves that were recently visited). On every sweep, the move that results in the best objective function improvement and is not on the tabu list, is performed.

In Azure Quantum, the core algorithmic approach to our tabu search implementation is described in [Beasley 1999 - Heuristic Algorithms for the Unconstrained Binary Quadratic Programming Problem](#), and we have extended this approach with various improvements in computational efficiency. For multi-threading, we borrow the multistart approach described in [Palubeckis et al. 2004 - Multi-start Tabu Search Strategies](#).

## Features of tabu search on Azure Quantum

Tabu search in Azure Quantum supports:

- Parameter-free mode and parameterized mode (with parameters)
- Ising and PUBO input formats
- CPU only

## When to use tabu search

In the context of binary programming problems, tabu search is most commonly used for QUBO problems (decision variables with value 0 and 1, with quadratic locality).

Tabu is expected to perform very well on problem landscapes with the following features:

- High term density (anywhere from densities of 5%-100%)
- Highly convex "simple" problems that will suit a greedy algorithm
- 0,1 decision variables and quadratic locality (QUBO problems)
- Smaller problems (although there is no limit on the number of variables accepted). Because only one move is made per sweep, this algorithm will naturally take longer for larger problems.

### NOTE

For further information on determining which solver to use, refer to [Which optimization solver should I use?](#).

## Parameter-free tabu search

The parameter-free version of tabu search is recommended for new users, those who don't want to manually tune parameters, and even as a starting point for further manual tuning. The main parameters to be tuned for this solver are the number of `sweeps` and the `tabu_tenure` (described in the next section).

The parameter-free solver will halt either on `timeout` (specified in seconds) or when there is sufficient convergence on a solution.

PARAMETER NAME	DESCRIPTION
timeout	Max execution time for the solver (in seconds). This is a best effort mechanism, so the solver may not stop immediately when the timeout is reached.
seed (optional)	Seed value - used for reproducing results.

To create a parameter-free Tabu solver using the SDK:

```
from azure.quantum.optimization import Tabu
# Requires a workspace already created.
solver = Tabu(workspace, timeout=100, seed=22)
```

The parameter-free solver will return the parameters used in the result JSON. You can then use these parameters to solve similar problems (similar number of variables, terms, locality and similar coefficient scale) using the parameterized tabu search solver.

Running the solver without any parameters also triggers the parameter-free version:

```
from azure.quantum.optimization import Tabu
# Requires a workspace already created.
# Not specifying any parameters runs the parameter-free version of the solver.
solver = Tabu(workspace)
```

## Parameterized tabu search

Tabu search with specified parameters is best used if you are already familiar with tabu search terminology (iterations, tenure etc.) and/or have an idea of which parameters you intend to use. If this is your first time using tabu search for a problem, the parameter-free version (described below) is recommended.

Tabu search supports the following parameters:

PARAMETER NAME	DESCRIPTION
sweeps	Number of sets of iterations to run over the variables of a problem. The more sweeps will usually always improve the solution (unless it is already at the global min).
tabu_tenure	Tenure of the tabu list in moves. Describes how many moves a variable stays on the tabu list once it has been made. This value is recommended to be between 1 and the number of variables to have any impact. This will be the main tuneable parameter that will determine the quality of the solution. A good starting point is 20.
restarts	The number of repeated instances to run the solver. Each instance will start with a random configuration <b>unless an initial configuration is supplied in the problem file</b> . This is recommended to be at least 72 to fully utilize machine capabilities.
replicas (deprecated)	The number of concurrent solvers to initialize. This parameter will now default to the number of available processors on the machine, and is no longer accepting input.

PARAMETER NAME	DESCRIPTION
seed (optional)	Seed value - used for reproducing results

To create a parameterized Tabu solver using the SDK:

```
from azure.quantum.optimization import Tabu
# Requires a workspace already created.
solver = Tabu(workspace, sweeps=2, tabu_tenure=5, restarts=72, seed=22)
```

# Quantum Monte Carlo

6/8/2021 • 2 minutes to read • [Edit Online](#)

**Quantum Monte Carlo** is a Metropolis annealing algorithm, similar in concept to [simulated annealing](#) that starts at a low temperature and improves the solution by searching across barriers with some probability as an external perturbation is applied to the system. As this external field is varied over every Monte Carlo step, the configuration may be able to tunnel through energy barriers and evolve towards a desired ground state (without possessing the thermal energy needed to climb the barriers, as would be required in simulated annealing).

In Azure Quantum the core of algorithmic approach to our QMC implementation is based on the [Wolff algorithm](#) for annealing and we extended this approach with various improvement in our computational efficiency.

## Features of Quantum Monte Carlo on Azure Quantum

- Parameterized mode (with parameters)
- Ising and PUBO input formats
- CPU only

## When to use Quantum Monte Carlo

This algorithm should perform best in the following two scenarios:

- When there are tall, narrow barriers in the energy landscape (cost function)
- If the solution is already at a feasible configuration at a low temperature and the user wishes to improve the solution

### NOTE

For further information on determining which solver to use, refer to [Which optimization solver should I use?](#).

## Parameterized Quantum Monte Carlo (CPU)

Quantum Monte Carlo supports the following parameters:

PARAMETER NAME	DESCRIPTION
<code>sweeps</code>	Number of sets of iterations to run over the variables of a problem. More sweeps will usually always improve the solution (unless it is already at the global min).
<code>trotter_number</code>	The number of copies of every variable to generate for running the simulation.

PARAMETER NAME	DESCRIPTION
<code>restarts</code>	The number of repeats of the annealing schedule to run. Each restart will start with a random configuration unless an initial configuration is supplied in the problem file. The restarts will be executed in parallel and split amongst the threads of the virtual machine. Recommended to set this value to at least 72.
<code>beta_start</code>	Represents the temperature at which the annealing schedule is executed. This should be a value low enough to produce a feasible configuration.
<code>transverse_field_start</code> & <code>transverse_field_stop</code>	Represents the starting and stopping values of the external field applied to the annealing schedule. A suitable value for these parameters will depend entirely on the problem and the magnitude of its changing moves. In general a non-zero and declining acceptance probability is sufficient.
<code>seed</code> (optional)	Seed value - used for reproducing results.

To create a parameterized Quantum Monte Carlo solver using the SDK:

```
from azure.quantum.optimization import QuantumMonteCarlo
# Requires a workspace to be already created
solver = QuantumMonteCarlo(workspace, sweeps = 2, trotter_number = 10, restarts = 72, beta_start = 0.1,
transverse_field_start = 10, transverse_field_stop = 0.1, seed = 22)
```

# Substochastic Monte Carlo

7/9/2021 • 3 minutes to read • [Edit Online](#)

**Substochastic Monte Carlo** is a diffusion Monte Carlo algorithm inspired by adiabatic quantum computation. It simulates the diffusion of a population of walkers in search space, while walkers are removed or duplicated based on how they perform according the cost function.

The initial set of walkers consists of random starting points (`target_population` = number of walkers), which are subjected to *random* transitions and a resampling process parameterized by `alpha` and `beta`:

- `alpha`: The probability of making a random transition (single variable change in the context of binary models)
- `beta`: The factor to apply when resampling the population (higher values for `beta` favor low-cost states more heavily)

Like [Simulated Annealing](#) or [Population Annealing](#), the parameters `alpha` and `beta` are typically chosen in a time-dependent form, with `alpha` decreasing over time and `beta` increasing. This has the effect that the simulation focuses on exploration initially, and optimization later. Note that `beta` is not the same as inverse temperature in other annealing-based optimization methods, although it does govern roughly the same effect: when `beta` is small walkers are free to explore the space, but when `beta` gets large walkers in poor configurations are likely to be removed while those in good regimes will duplicate.

## Features of Substochastic Monte Carlo on Azure Quantum

Substochastic Monte Carlo in Azure Quantum supports:

- Parameterized mode
- Ising and PUBO input formats

## When to use Substochastic Monte Carlo

Substochastic Monte Carlo exhibits a tunneling-like property, akin to quantum adiabatic evolution, through the combination of diffusion and resampling. Hence it is likely to find best use in problems obtained from constrained optimization where the constraints cause severe nonconvexity that traps sequential methods such as [Simulated Annealing](#) or [Tabu Search](#). If long runs of Simulated Annealing or Tabu Search are returning diverse values, then it is likely they are being trapped by a rough optimization landscape. In this case Substochastic Monte Carlo with a modest population size would be a good alternative to try.

### NOTE

For further information on choosing which solver to use, please refer to [this document](#).

## Parameterized Substochastic Monte Carlo

Suitable values for the `target_population`, `step_limit` and the resampling schedule depend on the problem and the magnitude (cost difference) of its variable changes.

- Modest `target_population` sizes (10-100) tend to work best. While some problems can benefit from larger populations, often one sees diminishing returns on effort to success.

- The `beta_start`, `beta_stop` range from [Simulated Annealing](#) can be a helpful guidance for the resampling parameter schedule.
- Substochastic Monte Carlo performs individual variable updates between resampling (rather than full sweeps). As a result the `step_limit` should be higher as compared to, e.g., Simulated Annealing.

Substochastic Monte Carlo supports the following parameters:

PARAMETER NAME	DEFAULT VALUE	DESCRIPTION
<code>step_limit</code>	10000	Number of monte carlo steps. More steps will usually improve the solution (unless it is already at the global minimum).
<code>target_population</code>	<i>number of threads</i>	The number of walkers in the population (should be greater-equal 8).
<code>alpha</code>	linear <code>1 .. 0</code>	Schedule for the stepping chance (must be decreasing, i.e. <code>alpha.initial &gt; alpha.final</code> ).
<code>beta</code>	linear <code>0 .. 5</code>	Schedule for the resampling factor (must be increasing, i.e. <code>beta.initial &lt; beta.final</code> ).
<code>seed</code> (optional)	<i>time based</i>	Seed value - used for reproducing results.

To create a parameterized Substochastic Monte Carlo solver for the CPU using the SDK:

```
from azure.quantum.optimization import SubstochasticMonteCarlo, RangeSchedule
# Requires a workspace already created.
solver = SubstochasticMonteCarlo(workspace, step_limit=10000, target_population=64,
beta=RangeSchedule("linear", 0, 5), seed=42)
```

## Parameter-Free Substochastic Monte Carlo

### NOTE

Parameter-Free Substochastic Monte Carlo is currently available to users in the Azure Quantum Early Access program. It will be available to all users in the near future.

Parameter-free Substochastic Monte Carlo searches for "optimal" parameters of the Substochastic Monte Carlo solver at runtime. This means that you do not need to set up parameters like `alpha`, `beta`, and so on. The only parameter required to run the parameter-free Substochastic Monte Carlo solver is `timeout` which represents the physical time in seconds that the solver is allowed to run.

PARAMETER NAME	DEFAULT VALUE	DESCRIPTION
<code>timeout</code> (required)	5	Number of seconds to allow the solver to run.

PARAMETER NAME	DEFAULT VALUE	DESCRIPTION
seed (optional)	<i>time based</i>	Seed value - used for reproducing results.

Note that the `timeout` parameter is required to trigger the parameter-free Substochastic Monte Carlo solver. If you do not use the `timeout` parameter the default parameters for parameterized Substochastic Monte Carlo will be used instead.

While the parameter-free version of Substochastic Monte Carlo is in 'early access' you can trigger the parameter-free solver by using code similar to the sample shown below:

```
from azure.quantum.optimization import SubstochasticMonteCarlo
# Requires a workspace already created.
solver = SubstochasticMonteCarlo(workspace, seed=48)
solver.target = 'microsoft.substochasticmontecarlo-parameterfree.cpu'
solver.set_one_param("timeout", 10)
```

# Quantum Workspace

5/24/2021 • 2 minutes to read • [Edit Online](#)

```
from azure.quantum import Workspace
```

## Constructor

To create a `Workspace` object, you must supply the following arguments in order to connect. If you have not already created a workspace, follow the steps in [Creating an Azure Quantum workspace guide](#) using the following values:

- `subscription_id` : The subscription ID where the workspace is deployed.
- `resource_group` : The name of the resource group where the workspace is deployed.
- `name` : The name of the workspace.
- `location` : The location where the workspace is deployed, for example `West US`, with either the `create` or `show` commands.
- `credential` : (Optional) The credential to use to connect to the Azure Quantum and Storage services. Normally one of the [credential types from Azure.Identity](#). Defaults to `DefaultAzureCredential`, which will attempt [multiple forms of authentication](#).

You can find these values by viewing your Azure Quantum Workspace details through the Azure portal.

In case you have not specified any credentials, the first time you run a method which interacts with the Azure service, a window might prompt in your default browser asking for your credentials.

```
workspace = Workspace(  
    subscription_id = "", # Add your subscription_id  
    resource_group = "", # Add your resource_group  
    name = ""           # Add your workspace name  
    location= ""        # Add the workspace location, for example, westus  
)
```

## Workspace.get\_job

Retrieves information about a job.

```
from azure.quantum import Workspace  
  
workspace = Workspace(...)  
job = workspace.get_job("285cfccb4-6822-11ea-a05f-2a16a847b8a3")  
print(job.details.status)  
  
> Succeeded
```

## Workspace.list\_jobs

Returns the list of existing jobs in the workspace.

```
from azure.quantum import Workspace

workspace = Workspace(...)
jobs = workspace.list_jobs()
for job in jobs:
    print(job.id, job.details.status)

> 08ea8792-68f2-11ea-acc5-2a16a847b8a3 Succeeded
> 0ab1863a-68f2-11ea-82b3-2a16a847b8a3 Succeeded
> 0c5c507e-68f2-11ea-ba75-2a16a847b8a3 Cancelled
> f0c8de58-68f1-11ea-a565-2a16a847b8a3 Executing
```

## Workspace.cancel\_job

Cancels a job that was previously submitted.

```
from azure.quantum import Workspace

workspace = Workspace(...)
job = workspace.get_job("285cfccb4-6822-11ea-a05f-2a16a847b8a3")

workspace.cancel_job(job)
print(job.details.status)

> Succeeded
```

# Quantum optimization Job

6/8/2021 • 2 minutes to read • [Edit Online](#)

```
from azure.quantum.optimization import Job
```

## Job.get\_results

Retrieves the job result (that is, the computed solution and cost). If the job has not yet finished, blocks until it has.

```
results = job.get_results()
print(results)

> {'version': '1.0', 'solutions': [{'configuration': {'1': 1, '0': 1, '2': -1, '3': 1}, 'cost': -23.0}]}  
[{"id": "5d2f9cd70f55f149e3ed3aef"}]
```

## Job.refresh

Refreshes the job's details by querying the workspace.

```
job = workspace.get_job(jobId)
job.refresh()
print(job.id)

> 5d2f9cd70f55f149e3ed3aef
```

## Job.has\_completed

Returns a boolean value indicating whether the job has finished (for example, the job is in a [final state](#)).

```
job = workspace.get_job(jobId)
job.refresh()
print(job.has_completed())

> False
```

## Job.wait\_until\_completed

Keeps refreshing the job's details until it reaches a final state. For more information on job states, see [Azure Quantum Overview](#).

```
job = workspace.get_job(jobId)
job.wait_until_completed()
print(job.has_completed())

> True
```

# Quantum optimization problem

6/8/2021 • 2 minutes to read • [Edit Online](#)

## Problem

```
from azure.quantum.optimization import Problem
```

### Constructor

To create a `Problem` object, you specify the following information:

- `name` : A friendly name for your problem. No uniqueness constraints.
- [optional] `terms` : A list of `Term` objects to add to the problem.
- [optional] `problem_type` : The type of problem. Must be either `ProblemType.ising` or `ProblemType.pobo`. Default is `ProblemType.ising`.
- [optional] `init_config` : A dictionary of variable ids to value if user wants to specify an initial configuration for the problem.

```
terms = [
    Term(c=-9, indices=[0]),
    Term(c=-3, indices=[1,0]),
    Term(c=5, indices=[2,0])
]

problem = Problem(name="My Difficult Problem", terms=terms)

# with initial configuration set
config = {'0': 1, '1': 1, '2': 0}
problem2 = Problem(name="Problem with Initial Configuration", terms=terms, init_config=config)
```

### Problem.add\_term

Adds a single term to the problem. It takes a coefficient for the term and the indices of variables that appear in the term.

```
coefficient = 0.13
problem.add_term(c=coefficient, indices=[2,0])
```

### Problem.add\_terms

Adds multiple terms to the problem using a list of `Terms`.

```
problem.add_terms([
    Term(c=-9, indices=[0]),
    Term(c=-3, indices=[1,0]),
    Term(c=5, indices=[2,0]),
    Term(c=9, indices=[2,1]),
    Term(c=2, indices=[3,0]),
    Term(c=-4, indices=[3,1]),
    Term(c=4, indices=[3,2])
])
```

### Problem.serialize

Serializes a problem to a json string.

```
problem = Problem("My Problem", [Term(c=1, indices=[0,1])])
problem.serialize()

> {"cost_function": {"version": "1.0", "type": "ising", "terms": [{"c": 1, "ids": [0, 1]}]}}
```

## Problem.upload

Problem data can be explicitly uploaded to an Azure storage account using its `upload` method that receives as a parameter the `Workspace` instance:

```
problem.upload(workspace=workspace)
```

Once a problem is explicitly uploaded, it will not be automatically uploaded during submission unless its terms change.

## Problem.evaluate

Once a problem has been defined, the user can evaluate the problem on any configuration they supply. The configuration should be supplied as a dictionary of variable ids to values.

```
problem = Problem("My Problem", [Term(c=1, indices=[0,1])])
problem.evaluate({0:1, 1:1})
> 1

problem.evaluate({0:1, 1:0})
> 0
```

## Problem.set\_fixed\_variables

During experimentation, the user may want to set a variable (or a group of variables) to a particular value. Calling `set_fixed_variables` will return a new Problem object representing the modified problem after such variables have been fixed.

```
fixed_var = {'1': 1, '2': 1}
problem = Problem("My Problem", [Term(c=1, indices=[0,1]), Term(c=11, indices=[1,2]), Term(c=5, indices=[])])
new_problem = problem.set_fixed_variables(fixed_var)
new_problem.terms

> [{'c': 1, 'ids': [0]}, {'c': 16, 'ids': []}]
```

To piece back the fixed variables with the solution on the reduced problem:

```
result = solver.optimize(new_problem)
result_config = json.loads(result)['solutions'][0]['configuration']
result_config.update(fixed_var) # join the fixed variables with the result
```

## Problem.get\_terms

The user can get the terms in which a variable exists using this function.

```
problem = Problem("My problem" , [Term(c=1, indices=[0,1]), Term(c=11, indices=[1,2]), Term(c=5, indices=[1])])
terms = problem.get_terms(id = 1)
terms

> [{"c": 11, "ids": [1,2]}, {"c": 5, "ids": [1]}]
```

## StreamingProblem

The StreamingProblem class can handle large problems that exceeds local memory limits. Unlike with the Problem class, terms in the StreamingProblem are uploaded directly to blob and are not kept in memory.

The StreamingProblem class uses the same interface as the Problem class. See [StreamingProblem](#) usage documentation.

There are some features that are not yet supported on the StreamingProblem class due to its streaming nature:

- Problem.set\_fixed\_variables()
- Problem.evaluate()

## OnlineProblem

The OnlineProblem class creates a problem from the url of the blob storage where an optimization problem has been uploaded. It is essentially used to reuse already submitted problems. It does not support client side analysis, e.g. the `evaluate` and `set_fixed_variables` functions. It allows you to download the problem from the blob storage as an instance of the Problem class to do any of the client side operations. For an example of how to use the OnlineProblem class, have a look at [reusing problem definitions](#).

# Quantum optimization ProblemType

3/5/2021 • 2 minutes to read • [Edit Online](#)

## Quantum optimization problem type

```
from azure.quantum.optimization import ProblemType
```

The `ProblemType` enum allows you to specify the type of optimization problem you would like to solve.

### **ProblemType.pubo**

A polynomial unconstrained binary optimization problem (PUBO) is a problem of the form:

$$\$ H = \sum_i c_{\{i\}} x_{\{i\}} + \sum_{i,j} c_{\{i,j\}} x_{\{i\}} x_{\{j\}} + \sum_{i,j,k} c_{\{i,j,k\}} x_{\{i\}} x_{\{j\}} x_{\{k\}} \text{ where } c_{\{i,j,k\}} \in \mathbb{R} \text{ and } x_{\{i,j,k\}} \in [0, 1] \$$$

Where  $H$  is the cost function, also known as the Hamiltonian. It is called  $k$ -local if the maximum degree of the polynomial is  $k$ .

### **ProblemType.ising**

An Ising Model is a cost function of the form:

$$\$ H = \sum_i c_{\{i\}} s_{\{i\}} + \sum_{i,j} c_{\{i,j\}} s_{\{i\}} s_{\{j\}} + \sum_{i,j,k} c_{\{i,j,k\}} s_{\{i\}} s_{\{j\}} s_{\{k\}} \text{ where } c_{\{i,j,k\}} \in \mathbb{R} \text{ and } s_{\{i,j,k\}} \in [-1, 1] \$$$

It is called  $k$ -local if the maximum degree of the polynomial is  $k$ .

# Streaming upload of large optimization problems

5/27/2021 • 5 minutes to read • [Edit Online](#)

When formulating very large problems with the Python SDK you may find that you do not have enough memory to keep the entire problem definition loaded, which is the behavior of the `Problem` class. If you do not need to keep your whole problem definition in memory for later access or modification you should consider using the `StreamingProblem` class instead, which is a drop-in replacement for the `Problem` class, but that streams the problem definition to Azure as you formulate the problem to reduce memory requirements and increase performance.

## StreamingProblem

```
from azure.quantum.optimization import StreamingProblem
```

The `StreamingProblem` class supports the same interface for adding terms to a problem definition as the `Problem` class. However, once terms are added to the problem they are queued to be uploaded by a background thread and are not kept in memory for future reference.

### Lifecycle of a StreamingProblem

The first time you call one of the `add_term()` or `add_terms()` functions on a streaming problem, the following things happen:

1. A background thread is started
2. A [Block Blob](#) is initialized in Azure Storage, which is where the problem will be uploaded
3. The terms that were added are queued for upload

The background thread will stay alive until the problem definition is finalized, and will continuously attempt to chunk terms into blocks that are individually uploaded in Azure. A chunk upload is triggered when the queued terms surpass a threshold on the number of terms, or the size of the data. This means that not every call to `add_terms()` triggers an upload.

After the first call to `add_terms()` you may continue to call this method as many times as needed to add all of your terms. Note that this method is thread-safe so you may have multiple threads simultaneously adding terms to the problem definition although there is only ever one uploader thread.

Once you have finished adding all of your terms, you must call the `upload()` method to finalize the problem definition (note that if you call `solver.submit(problem)`, `upload()` is called for you). The first time the `upload()` method is called the streaming problem will block on the background upload thread to complete. Once this is done, the block blob is finalized in Azure Storage and is ready to be used to solve the problem with Azure Quantum. Note that you may not add terms to a problem after it has been finalized.

Note that if `upload()` is not called on a streaming problem, the block blob in Azure Storage is never finalized. When this happens, Azure Storage will automatically delete the uploaded data within a period of time. For more information, see the [Block Blob reference](#).

### Tuning the upload

Depending on the characteristics of your problem (especially density) or of your CPU/network connection you may want to tune the `StreamingProblem` class. There are two options for tuning the upload:

- `StreamingProblem.upload_size_threshold` - the size, in bytes, of the compressed payload to upload. As terms are added they are compressed on the fly. When the size of the staged compressed payload surpasses this

threshold the chunk is uploaded.

- `StreamingProblem.upload_terms_threshold` - the threshold for the number of terms that trigger an upload. When the number of queued terms exceeds this threshold an upload is triggered. If your problem has high connectivity (terms have many variables) you may choose to lower this threshold.

To tune these parameters, set one or both as in the example below **before adding terms to the problem definition**. Changing these parameters after the background uploader begins will have no effect.

```
problem = StreamingProblem(workspace = workspace, name="My Streamed Problem",
problem_type=ProblemType.ising)
problem.upload_size_threshold = 10e6
problem.upload_terms_threshold = 1000
```

In general:

- If you have plenty of free memory, consider *increasing* the thresholds
- If you have a slow network connection, consider *increasing* the thresholds
- If you are highly memory constrained, consider *decreasing* the size threshold

#### Statistics of the uploaded problem

The `StreamingProblem` class calculates some statistics about the generated problem as terms are added. To see information about the terms already added to the problem, access the `stats` member of the class which is a dictionary of calculated statistics and contains the following information:

```
problem = StreamingProblem(workspace = workspace, name="My Streamed Problem",
problem_type=ProblemType.ising)
# Terms are added...
print(problem.stats)

> {'type': 'ising', 'max_coupling': 4, 'avg_coupling': 2.4, 'min_coupling': 2, 'num_terms': 6125}
```

- `type` : The type of problem - `pubo` or `ising`
- `max_coupling` : The largest coupling (number of variable indices) in any term
- `avg_coupling` : The average coupling (number of variable indices) of all terms
- `min_coupling` : The smallest coupling (number of variable indices) in any term
- `num_terms` : The total number of terms

These statistics are also set as metadata on the uploaded blob in Azure Storage. See the [Azure Storage documentation](#) for more information on accessing these properties.

#### Constructor

To create a `StreamingProblem` object, specify the following information:

- `workspace` : The Quantum Workspace this problem is being uploaded to be solved in. The problem will be stored in the linked storage account from the workspace.
- `name` : A friendly name for your problem. No uniqueness constraints.
- [optional] `terms` : A list of `Term` objects to add to the problem. If provided this will start the background uploader.
- [optional] `problem_type` : The type of problem. Must be either `ProblemType.ising` or `ProblemType.pubo`. Default is `ProblemType.ising`.

```
problem = StreamingProblem(workspace = workspace, name="My Streamed Problem",
problem_type=ProblemType.ising)
```

## **StreamingProblem.add\_term**

Adds a single term to the problem. It takes a coefficient for the term and the indices of variables that appear in the term.

Thread-safe: may be called from multiple calling threads.

```
coefficient = 0.13
problem.add_term(c=coefficient, indices=[2,0])
```

## **StreamingProblem.add\_terms**

Adds multiple terms to the problem using a list of `Terms`.

Thread-safe: may be called from multiple calling threads.

```
problem.add_terms([
    Term(c=-9, indices=[0]),
    Term(c=-3, indices=[1,0]),
    Term(c=5, indices=[2,0]),
    Term(c=9, indices=[2,1]),
    Term(c=2, indices=[3,0]),
    Term(c=-4, indices=[3,1]),
    Term(c=4, indices=[3,2])
])
```

## **StreamingProblem.download**

Downloads the uploaded problem definition as an instance of `Problem`. May only be called on a streaming problem after it has been finalized by calling `upload()`.

## **StreamingProblem.upload**

Finalizes the upload of the streaming problem in Azure Storage by blocking on the background uploader thread until it completes. Once this method returns, the problem is ready to be submitted to an Azure Quantum Solver.

This method may be called multiple times. If the the uploading has not yet completed all calls will block. If the uploading has completed, it will immediately return the URL of the uploaded problem (without re-uploading).

```
problem.upload()
```

# Input format for optimization problems

6/8/2021 • 3 minutes to read • [Edit Online](#)

This document explains how the parameters to optimization problems may be specified for all the different solvers. All solvers set default values for their parameters but we strongly recommend setting them to values appropriate for your problem. Where there is a parameter-free solver available not setting any parameters will call the parameter-free version of that solver which will complete when there is sufficient convergence on a solution.

## Parallel Tempering

PROPERTY NAME (CASE SENSITIVE)	TYPE	DESCRIPTION
all_betas	a list of floats	Specifies the list of inverse temperatures. This list should be equal in length to the number of replicas.
replicas	integer	Specifies the number of iterations of the solver to run.
sweeps	integer	Specifies the number of Monte Carlo steps to perform in each iteration of a solver.
seed	A random integer.	Specifies a random value to start the simulation.
timeout	integer	Specifies the maximum number of seconds to run the core solver loop. Initialization time does not respect this value, so the solver may run longer than the value specified.

## Simulated Annealing

PROPERTY NAME (CASE SENSITIVE)	TYPE	DESCRIPTION
beta_start	float	Specifies the list of inverse temperatures. This list should be equal in length to the number of replicas.
beta_stop	float	Specifies the number of iterations of solver to run.
sweeps	integer	Specifies the number of Monte Carlo steps to perform in each iteration of a solver.
seed	A random integer.	Specifies a random value to start the simulation.

PROPERTY NAME (CASE SENSITIVE)	TYPE	DESCRIPTION
timeout	integer	Specifies the maximum number of seconds to run the core solver loop. Initialization time does not respect this value, so the solver may run longer than the value specified.
restarts	integers	Specifies the number of iterations of the simulation to run.

## Population Annealing

PROPERTY NAME (CASE SENSITIVE)	TYPE	DESCRIPTION
sweeps	integer	Number of sweeps. More sweeps will usually improve the solution if it has not yet found a global minimum.
beta	RangeSchedule	Specifies a range from the initial temperature value to the final value. This schedule must increase over time.
population	integer	The number of walkers in the population that the algorithm should use.
seed	A random integer	Used to initialize the algorithm. Use the same seed to reproduce results.

## Tabu

PROPERTY NAME (CASE SENSITIVE)	TYPE	DESCRIPTION
tabu_tenure	integer	Specifies the Tabu tenure.
timeout	integer	Specifies the maximum number of seconds to run the core solver loop. Initialization time does not respect this value, so the solver may run longer than the value specified.
seed	A random integer between 0 and 101	Specifies a random value to start the simulation.
sweep	integer	Specifies the number of Monte Carlo steps to perform in each iteration of the simulation.

## Quantum Monte Carlo

PROPERTY NAME (CASE SENSITIVE)	TYPE	DESCRIPTION

PROPERTY NAME (CASE SENSITIVE)	TYPE	DESCRIPTION
beta_start	float	Specifies the inverse of the starting temperature for the algorithm.
transverse_field_start	float	Specifies the starting value of the external field supplied to the simulation.
transverse_field_end	float	Specifies the ending value of the external field supplied to the simulation.
sweep	integer	Specifies the number of Monte Carlo steps to perform in each iteration of the simulation.
trotter_number	integer	Specifies the number of copies of each variable to create in a simulation.
seed	A random integer	Specifies a random value to start the simulation.

## Substochastic Monte Carlo

PROPERTY NAME (CASE SENSITIVE)	TYPE	DESCRIPTION
step_limit	integer	Number of Monte Carlo steps. More steps will usually improve the solution if it has not yet found a global minimum.
target_population	integer	Specifies the number of walkers in the population. Should be greater than or equal to 8.
alpha	RangeSchedule	Specifies a range from the initial value to the final value. This is the schedule for the stepping chance and should decrease over time.
beta	RangeSchedule	Specifies a range from the initial value to the final value. This is the schedule for the resampling factor that will increase over time.
seed	A random integer	Used to initialize the algorithm. Use the same seed to reproduce results.

# Quantum optimization term

5/27/2021 • 2 minutes to read • [Edit Online](#)

## Term

```
from azure.quantum.optimization import Term
```

### Constructor

An optimization problem consists of a sum of terms, each of which we can represent with the `Term` object. To create a `Term` object, you specify the following parameters:

- `c`: This corresponds to the coefficient.
- `indices`: This corresponds to the product. More specifically, it is populated with the indices of all variables appearing in the term.

For instance, the term  $2 \cdot (x_1 \cdot x_2)$  translates to the following object:

```
Term(c=2, indices=[1,2])
```

Or, the term  $3 \cdot (x_0 \cdot x_1 \cdot x_2)$  translates to the following object:

```
Term(c=3, indices=[0,1,2])
```

For more information on cost functions and how terms relate to a problem definition, see [Cost functions](#). Terms can be supplied to a `Problem` object, see [Problem](#).

# Quantum optimization range schedule

6/8/2021 • 2 minutes to read • [Edit Online](#)

## RangeSchedule

```
from azure.quantum.optimization import RangeSchedule
```

Some solvers allow you to specify an instance of the RangeSchedule class as an input. A RangeSchedule can be created with two different types of schedules (either linear or geometric). We recommend reviewing our [Solver Inputs](#) documentation for more details.

### Constructor

- `schedule_type` : This is a string indicating the type of schedule. The accepted values are `linear` and `geometric`.
- `initial` : A floating point value indicating the initial value of the schedule.
- `final` : A floating point value indicating the final value of the schedule.

For instance, if you want to create a linear range schedule that increases from 0 to 5 over time:

```
RangeSchedule("linear", 0, 5)
```

Or, if you want to create a geometric range schedule that decreases from 1 to 0:

```
RangeSchedule("geometric", 1, 0)
```

# Express an optimization problem

3/5/2021 • 2 minutes to read • [Edit Online](#)

To express a simple problem to be solved, create an instance of a `Problem` and set the `problem_type` to either `ProblemType.ising` or `ProblemType.pobo`. For more information, see [ProblemType](#).

```
from azure.quantum.optimization import Problem, ProblemType, Term, ParallelTempering

problem = Problem(name="My First Problem", problem_type=ProblemType.ising)
```

Next, create an array of terms and add them to the `problem`:

```
terms = [
    Term(c=-9, indices=[0]),
    Term(c=-3, indices=[1,0]),
    Term(c=5, indices=[2,0]),
    Term(c=9, indices=[2,1]),
    Term(c=2, indices=[3,0]),
    Term(c=-4, indices=[3,1]),
    Term(c=4, indices=[3,2])
]

problem.add_terms(terms=terms)
```

## NOTE

As described below, there are multiple ways to supply terms to the problem.

## Ways to supply problem terms

There are three ways to supply terms for a `Problem`: in the constructor, individually, and as a list of `Term` objects.

### In the constructor

You can supply an array of `Term` objects in the constructor of a `Problem`.

```
terms = [
    Term(c=-9, indices=[0]),
    Term(c=-3, indices=[1,0]),
    Term(c=5, indices=[2,0])
]

problem = Problem(name="My Difficult Problem", terms=terms)
```

### Individually

You can supply each term individually by calling the `add_term` method on the `Problem`.

```
problem = Problem(name="My Difficult Problem", problem_type=ProblemType.ising)
problem.add_term(c=-9, indices=[0])
problem.add_term(c=-3, indices=[1,0])
problem.add_term(c=5, indices=[2,0])
```

## Add a list of terms

You can also supply a list of `Term` objects using the `add-terms` method on the `Problem`.

```
problem = Problem(name="My Difficult Problem")
terms = [
    Term(c=-9, indices=[0]),
    Term(c=-3, indices=[1,0]),
    Term(c=5, indices=[2,0]),
    Term(c=9, indices=[2,1]),
    Term(c=2, indices=[3,0]),
    Term(c=-4, indices=[3,1]),
    Term(c=4, indices=[3,2])
]

problem.add_terms(terms=terms)
```

# Apply solvers to solve optimization problems

6/25/2021 • 2 minutes to read • [Edit Online](#)

Once we have a `Problem`, we're ready to solve it by applying a **solver**. In this example we'll use a parameter-free version of parallel tempering. You can find documentation on this solver and the other available solvers in the [Solver overview](#).

```
solver = ParallelTempering(workspace, timeout=100)
```

For arguments, the solver takes the `Workspace` created previously, plus a single parameter which is the maximum amount of time (in seconds) to run the solver. Detailed documentation on parameters is available in the reference for each solver.

## NOTE

See [Use the Python SDK](#) for details on connecting to a workspace and getting a `Workspace` object for it.

Solvers provide an `optimize` method that expects a `Problem`. The `optimize` method uploads the problem definition, submits a job to solve the problem, and polls the status until the job has completed running. Once the job has completed, it returns a `JobOutput` object which contains the results. See [Understand Solver results](#) for interpreting the results.

```
result = solver.optimize(problem)
print(result)
```

This method will submit the problem to Azure Quantum for optimization and synchronously wait for it to be solved. You'll see output like the following in your terminal:

```
> {'solutions':[{'configuration': {'0': 1, '1': 1, '2': -1, '3': 1}, 'cost': -32.0}]}
```

See [Solve long running problems](#) for solving problems asynchronously.

## Using CPU vs FPGA solvers

By default, the CPU version of a solver is used. In order to specify a different version like FPGA, specify a `platform` parameter as shown below.

```
solver = SimulatedAnnealing(workspace, timeout=100, platform=HardwarePlatform.FPGA)
```

## Returning multiple solutions

CPU-based solvers in the Microsoft QIO provider support the option to return more than 1 solution on a single run. The below example shows how to set this option.

```
# return a maximum of 5 solutions
solver.set_number_of_solutions(5)
```

The option will have the following behavior and requirements:

- Value must be between 1 and 10 (default to 1 if not set).
- Solvers are not guaranteed to return the exact number of solutions specified by the parameter as solvers may not find that many solutions.
- Solvers are guaranteed to return the best solution they find in index 0. They will return the rest of the solutions (if any) in the same list and in sorted order.
- A build of the Python SDK for Optimization with version 0.17.2106 or higher is required to take advantage of the feature

**Not supported:** this option is not available on FPGA solvers.

# Understand solver results

6/25/2021 • 2 minutes to read • [Edit Online](#)

The result of a solver job is a `JobOutput` object which can be examined to obtain useful information.

Some of the useful properties in the result are:

1. `configuration`: The dictionary describes the assignment of variables, where for each key-value pair in the dictionary the key is the index of a variable and value is the value assigned to that variable by the solver.
2. `cost`: The optimized solution to the problem when the `configuration` is applied to the variables
3. `parameters`: This field will be present if a parameter-free solver was used. It will contain the optimal parameters found by the solver for the specific problem submitted. Different solvers take different parameters to solve the problem. In this example, we get:
  - a. `all_betas`: An array of the starting temperatures for the parallel tempering solver
  - b. `replicas`: The number of runs of the solver before evaluating the best configuration from all these runs
  - c. `sweeps`: The number of Monte Carlo steps performed in each iteration of the solver.
4. `solutions`: Contains all solutions that the solver has returned. Including the best solution found which is also shown in `configuration`.

The example below shows how to print the result if a solver job was submitted synchronously.

```
result = solver.optimize(problem)
print(result)
```

The example below shows the asynchronous version.

```
job = solver.submit(problem)
result = job.get_results()
print(result)
```

In both cases, the result should look like the following:

```
> {'version': '1.0', 'configuration': {'0': 1, '1': 1, '2': -1, '3': 1}, 'cost': -32.0, 'parameters': {'all_betas': [0.1, 0.5, 1, 2, 4], 'replicas': 70, 'sweeps': 600}, 'solutions': [{"configuration": {"0": 1, "1": 1, "2": -1, "3": 1}, "cost": -32.0}]}</pre>
```

It should be noted that for now, the solution appears in both the root of the payload ('configurations' and 'cost') as well as in the 'solutions' field. **It is recommended to consume solutions from the 'solutions' field (as the root 'configuration' and 'cost' fields will be deprecated in the future).**

The best solution found by the solver will always appear in index 0 of the 'solutions' list (if multiple solutions are specified). You can specify the amount of solutions that you would like to receive when [configuring your solver](#).

Here is an example of a small problem where two solutions were returned.

```
> {'version': '1.0', 'configuration': {'0': 1, '1': -1, '2': 1, '3': 1}, 'cost': -32.0, 'parameters': {'all_betas': [0.1, 0.5, 1, 2, 4], 'replicas': 70, 'sweeps': 600}, 'solutions': [{"configuration": {"0": 1, "1": -1, "2": 1, "3": 1}, "cost": -32.0}, {"configuration": {"0": -1, "1": 1, "2": -1, "3": -1}, "cost": -32.0}]}</pre>
```

# Job management

6/14/2021 • 2 minutes to read • [Edit Online](#)

When a problem is submitted to a solver, a `Job` is created in Azure Quantum. The `Workspace` provides the following methods for managing jobs:

- `get_job`: Returns the `Job` metadata and results for a specific job (based on job `id`).
- `list_jobs`: Returns a list of all jobs in the workspace.
- `cancel_job`: Cancels a specific job.

See [Job Cancellation](#) for more information on how cancellation requests are processed.

You can use the `list_jobs` method to get a list of all jobs in the workspace:

```
jobs = [job.id for job in workspace.list_jobs()]
print(jobs)
```

```
['5d2f9cd70f55f149e3ed3aef', '23as12fs5d2f9cd70f55f', '1644428ea8507edb7361']
```

The next piece of code shows how to submit a job asynchronously and obtain its job id:

```
from azure.quantum.optimization import Problem, ProblemType, Term, ParallelTempering, SimulatedAnnealing

problem = Problem(name="MyOptimizationJob", problem_type=ProblemType.ising)
problem.add_term(c=-9, indices=[0])
problem.add_term(c=-3, indices=[1,0])
problem.add_term(c=5, indices=[2,0])

solver = SimulatedAnnealing(workspace)
job = solver.submit(problem)
print(job.id)
```

```
5d2f9cd70f55f149e3ed3aef
```

The function `get_job` can be called to get the metadata (including results) for a previously submitted job, using the job `id`:

```
job = workspace.get_job('5d2f9cd70f55f149e3ed3aef')
results = job.get_results()
print(results)
```

```
{'solutions': [{ 'configuration': { '0': 1, '1': 1, '2': -1}, 'cost': -17.0}]}  
}
```

In order to cancel a job, use the function `cancel_job` as shown in this next piece of code:

```
job = workspace.get_job('5d2f9cd70f55f149e3ed3aef')
workspace.cancel_job(job)

print(job.details.status)
```

Cancelled

# Reusing problem definitions

5/24/2021 • 2 minutes to read • [Edit Online](#)

Sometimes it is more efficient to upload a problem definition once and find its solution using different algorithms (solvers) or with different parameters. You can upload a problem definition using the `upload` method, which returns a URL, and then provide this URL to a solver's `submit` or `optimize` methods:

```
url = problem.upload(workspace)
job = solver.submit(url)
print(job.id)
```

```
> 9228ea88-6832-11ea-8271-c49dede60d7c
```

You can also create an online problem from the submitted url and assign it a name. This `OnlineProblem` object also allows you to download the problem and get the terms of the problem to perform any client side analysis (`set_fixed_variables`, evaluate the cost function for a configuration or get specific terms from variable ids):

```
online_problem = OnlineProblem(name = "o_prob", blob_uri = url)
job = solver.submit(online_problem)
problem = online_problem.download(workspace)
```

# Solve long running problems

6/8/2021 • 2 minutes to read • [Edit Online](#)

In the example in [Apply solvers to solve optimization problems](#), a problem was submitted to Azure Quantum and solved synchronously. This is convenient for certain environments, but unsuitable for others where there is a need to either submit a problem and check on it later, or submit many problems and compare the results.

## Submit the problem

To submit a problem asynchronously, use the `submit` method on the `solver`. This submits a `Job` which is returned by the method:

```
solver = ParallelTempering(workspace, timeout=100, seed=11)
job = solver.submit(problem)
print(job.id)

> ea81bb40-682f-11ea-8271-c49dede60d7c
```

## Refresh job status

After submitting the job, you can check on the status of the job by calling the `refresh` method. Each time `refresh` is called, the job metadata gets refreshed.

```
job.refresh()
print(job.details.status)

> Succeeded
```

## Get the job output

Once the job is in a final state, such as `Succeeded`, you may download the job output using `get_results`:

```
jobId = "ea81bb40-682f-11ea-8271-c49dede60d7c"
job = workspace.get_job(jobId)
result = job.get_results()
print(result)

> {'solutions': [{}{'configuration': {'0': 1, '1': 1, '2': -1, '3': 1}, 'cost': -32.0}]}{}
```

# Glossary for Azure Quantum

5/27/2021 • 2 minutes to read • [Edit Online](#)

**Azure Active Directory (AAD) / Microsoft Identity Platform** - Azure's identity service, used to implement access control and authentication to resources. The names Azure Active Directory and Microsoft Identity Platform are interchangeable. For more information, see [Azure Active Directory](#).

**Azure Managed Application (AMA) / Managed Application** – A type of application offered to end customers in Azure through the Azure Marketplace. For more information, see [Azure managed applications](#).

**Azure Marketplace** – A storefront for cloud-based software in Azure. For more information, see [Azure Marketplace](#).

**Azure Quantum Job (Job)** – A program, problem, or application, submitted to Azure Quantum for processing by an Azure Quantum provider.

**Azure Quantum Job Target (Target)** – Providers expose one or more targets that can be used to run jobs. Customers select which target they would like to use to run a particular job. For example, a three-qubit machine and a six-qubit machine may each be targets.

**Azure Quantum Provider (provider)** – A provider is a component in Azure Quantum that provides the ability to run jobs. Providers may be made available by Microsoft or by third-party partners.

**Azure Quantum** – Microsoft's quantum service for Azure, enabling customers access to quantum solutions from both Microsoft providers and partner providers.

**Azure Resource Manager (ARM)** – Azure's deployment and management service. For more information, see [Azure Resource Manager](#).

**Quantum Development Kit (QDK)** - Microsoft's software development kit for developing quantum applications in the Azure Quantum service. The QDK contains Q#, Microsoft's programming language for quantum computing, along with Q# libraries, samples and tutorials. It also contains developer APIs for running jobs on the Azure Quantum service. For more information, see the [Microsoft QDK Documentation](#)

**Quantum-Inspired Optimization Problem (QIO Problem)** - A problem expressed using our Python optimization library to be solved using Azure Quantum. Problems may be expressed as PUBOs (Polynomial Unconstrained Binary Optimization) or Ising forms.

**Quantum Program** - In the scope of Azure Quantum, a program written in Q# that targets a provider in Azure Quantum.

## Next steps

For more information, see:

- The [Microsoft Azure glossary](#)
- The [Quantum Development Kit glossary](#)

# Azure Quantum common issues

5/27/2021 • 2 minutes to read • [Edit Online](#)

When you first start working with Azure Quantum, you may run into these common issues.

## Submitting jobs

### **Issue: Operation returns an invalid status code 'Unauthorized'**

Steps to resolve this issue:

1. Open your Azure portal (<https://portal.azure.com>) and authenticate your account.
2. Under **Navigate**, click **Subscriptions** and select your subscription.
3. Click **Access control (IAM)**.
4. Under **Check access**, search for your email address and select the account.
5. You should not see an **Owner** or a **Contributor** role listed.
6. Click the **Role assignments** tab.

#### **NOTE**

If you don't see the **Role assignments** tab, you may need to expand the portal to full screen or close the <your name> assignments pane.

7. Click the **Role** dropdown, select either **Owner** or **Contributor**, then enter your email address and select your account.
8. Click **Save**.
9. You should now see your account set configured with either the **Owner** or **Contributor** role.
10. Create your Quantum Workspace again and then submit a job against this new Quantum Workspace.

### **Issue: "Failed to compile program" when attempting to submit a Q# program through the CLI**

When attempting to submit a job at the command prompt using the `az quantum submit` command, you may encounter the following error message:

```
> az quantum job submit ...
Failed to compile program.
Command ran in 21.181 seconds (init: 0.457, invoke: 20.724)
```

This error occurs when there is a problem with the Q# program that causes the compilation to fail. To see the specific error that is causing the failure, run `dotnet build` in the same folder.

### **Issue: Operation returned an invalid status code 'Forbidden'**

When you submit your first job you may get a 'forbidden' error code.

This issue may originate during the workspace creation: Azure Quantum fails to complete the role assignment linking the new workspace to the storage account that was specified. A typical scenario for this situation happens if the tab or web browser window is closed before the workspace creation is completed.

You can verify that you are running into this role assignment issue by following these steps:

- Navigate to your new quantum workspace in Azure Portal
- Under **Overview > Essentials > Storage account**, click on the storage account link
- In the left navigation bar, select **Access Control (IAM)**
- Select **Role Assignments**
- Verify that your workspace appears as a **Contributor**
- If the workspace does not appear as a **Contributor** you can either:
  - Create a new workspace and make sure to wait for the workspace creation to be completed before closing the web browser tab or window.
  - Add the proper role assignment under the storage account
    - Access Control (IAM) > Add role assignments
    - Role > Contributor
    - Assign access to > User, group, or service principal
    - Select > [Workspace name]
    - Save

# Using a service principal to authenticate

6/1/2021 • 2 minutes to read • [Edit Online](#)

Sometimes it is unsuitable to use interactive authentication or to authenticate as a user account. These cases may arise when you want to submit jobs from a web service, another worker role, or an automated system. In this case you typically want to authenticate using a [Service Principal](#).

## Prerequisite: Create a service principal and application secret

To authenticate as a service principal, you must first [create a service principal](#).

To create a service principal, assign access, and generate a credential:

1. [Create an Azure AAD application](#):

### NOTE

You do not need to set a redirect URI

- a. Once created, write down the *Application (client) ID* and the *Directory (tenant) ID*.

2. [Create a credential](#) to login as the application:

- a. In the settings for your application, select **Certificates & secrets**.
- b. Under **Client Secrets**, select **Create New Secret**.
- c. Provide a description and duration, then select **Add**.
- d. Copy the value of the secret to a safe place immediately - you won't be able to see it again!

3. Give your service principal permissions to access your workspace:

- a. Open the Azure Portal.
- b. In the search bar, enter the name of the resource group you created your workspace in. Select the resource group when it comes up in the results.
- c. On the resource group overview, select **Access control (IAM)**.
- d. Click **Add Role Assignment**.
- e. Search for and select the service principal.
- f. Assign either the **Contributor** or **Owner** role.

## Authenticate as the service principal

Step 1: Install the `azure-common` python package:

```
pip3 install azure-common
```

Step 2: Before you call `workspace.login()`, instantiate your service principal and provide it to the workspace:

```
from azure.common.credentials import ServicePrincipalCredentials
workspace.credentials = ServicePrincipalCredentials(
    tenant    = "", # From service principal creation, your Directory (tenant) ID
    client_id = "", # From service principal creation, your Application (client) ID
    secret    = "", # From service principal creation, your secret
    resource  = "https://quantum.microsoft.com" # Do not change! This is the resource you want to
authenticate against - the Azure Quantum service
)
```

#### NOTE

The `workspace.login()` method has been deprecated and is no longer necessary. The first time there is a call to the service, an authentication will be attempted using the credentials passed in the `Workspace` constructor or its `credentials` property. If no credentials were passed, several authentication methods will be attempted by the `DefaultAzureCredential`.

#### NOTE

In order to create a role assignment on the resource group or workspace, you need to be an *owner* or *user access administrator* at the scope of the role assignment. If you do not have permissions to create the Service Principal in your subscription, you will need to request permission from the *owner* or *administrator* of the Azure subscription.

If you have permissions only at Resource Group or Workspace level, you can try to create the service principal without subscription level assignment using:

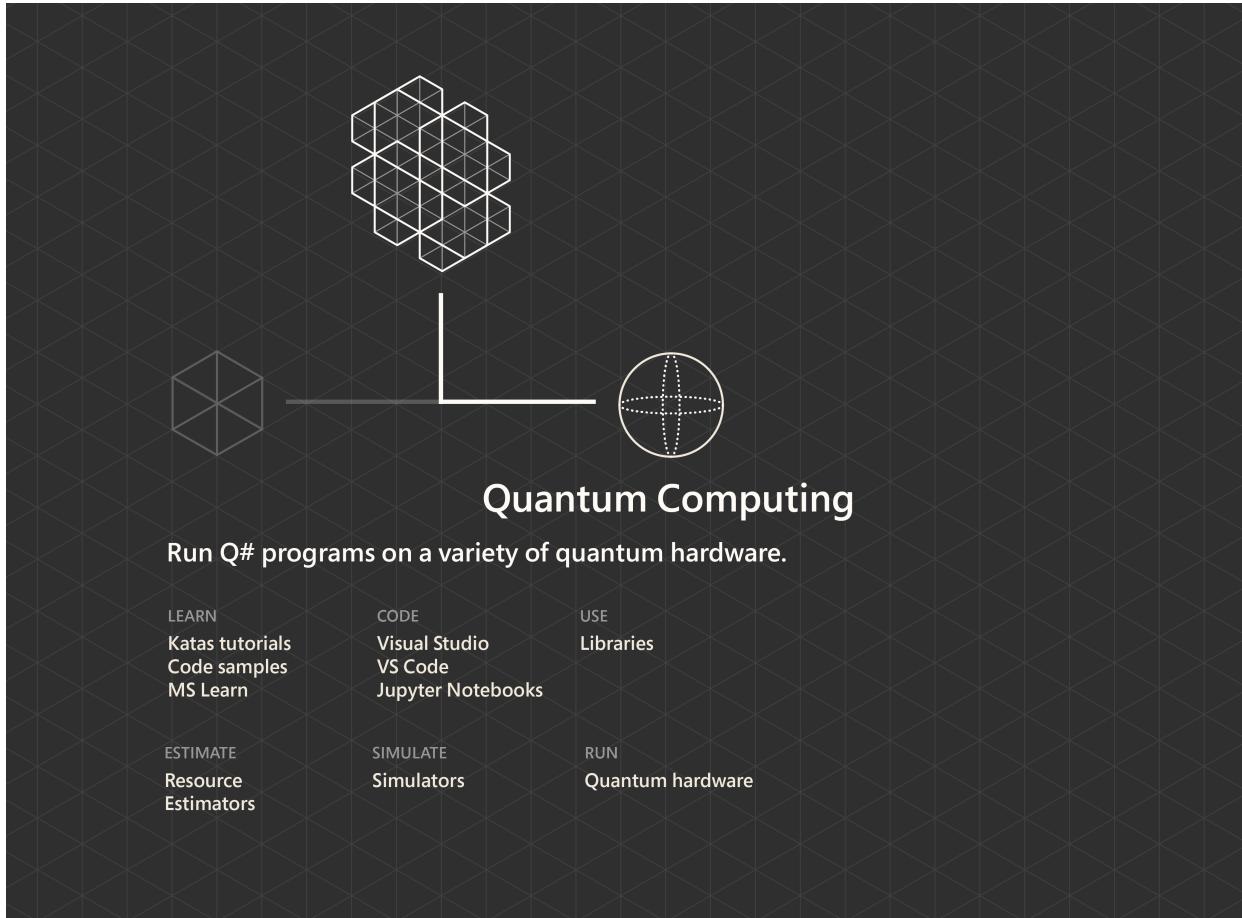
```
az ad sp create-for-rbac --skip-assignment true
```

Then, based on the output for the Service Principal name, you can go directly to the Resource Group or Azure Quantum Workspace in the portal to create a role assignment for that Service Principal, and assign the *contributor* role.

# What is quantum computing?

4/29/2021 • 3 minutes to read • [Edit Online](#)

Harnessing the unique behavior of quantum physics and applying it to computing, quantum computers introduce new concepts to traditional programming methods, making use of quantum physics behaviors such as superposition, entanglement, and quantum interference.



The graphic features a dark background with a light gray wireframe grid. In the center, there's a large 3D cube icon composed of lines, with a smaller version of the same icon to its left. To the right of the central cube is a circle containing a four-pointed star or crosshair symbol. Below these icons, the text "Quantum Computing" is centered in a bold, sans-serif font. Underneath that, the tagline "Run Q# programs on a variety of quantum hardware." is displayed in a smaller, regular font. The page is organized into a grid of sections with headings and links:

LEARN	CODE	USE
<a href="#">Katas tutorials</a>	<a href="#">Visual Studio</a>	<a href="#">Libraries</a>
<a href="#">Code samples</a>	<a href="#">VS Code</a>	
<a href="#">MS Learn</a>	<a href="#">Jupyter Notebooks</a>	
ESTIMATE	SIMULATE	RUN
<a href="#">Resource Estimators</a>	<a href="#">Simulators</a>	<a href="#">Quantum hardware</a>

Quantum computing holds the promise to solve some of our planet's biggest challenges - in the areas of environment, agriculture, health, energy, climate, materials science, and others we haven't encountered yet. For some of these problems, even our most powerful computers run into problems. While quantum technology is just beginning to impact the computing world, it could be far-reaching and change the way we think about computing.

In modern usage, the word quantum means the smallest possible discrete unit of any physical property, usually referring to properties of atomic or subatomic particles. Quantum computers use actual quantum particles, artificial atoms, or collective properties of quantum particles as processing units, and are large, complex, and expensive devices.

## What can a quantum computer do?

A quantum computer isn't a supercomputer that can do everything faster, but there are a few areas where quantum computers do exceptionally well.

- [Quantum simulation](#)
- [Cryptography](#)
- [Search](#)

- Optimization
- Machine learning

## Quantum simulation

Since quantum computers use quantum phenomena in computation, they are well suited for modeling other quantum systems. Photosynthesis, superconductivity, and complex molecular formations are examples of quantum mechanisms that quantum programs can simulate.

## Cryptography and Shor's algorithm

In 1994, Peter Shor showed that a scalable quantum computer could break widely used encryption techniques such as the RSA algorithm. Classical cryptography relies on the intractability of problems such as integer factorization or discrete logarithms, many of which can be solved more efficiently using quantum computers.

## Search and Grover's algorithm

In 1996, Lov Grover developed a quantum algorithm that dramatically sped up the solution to unstructured data searches, running the search in fewer steps than any classical algorithm could.

## Quantum-inspired computing and optimization

Quantum-inspired algorithms use quantum principles for increased speed and accuracy but implement on classical computer systems. This approach allows developers to leverage the power of new quantum techniques today without waiting for quantum hardware, which is still an emerging industry.

Optimization is the process of finding the best solution to a problem, given its desired outcome and constraints. Factors such as cost, quality, or production time all play into critical decisions made by industry and science. Quantum-inspired optimization algorithms running on today's classical computers can find solutions that up to now have not been possible. In addition to optimizing traffic flow to reduce congestion, there is airplane gate assignment, package delivery, job scheduling and more. With breakthroughs in materials science, there will be new forms of energy, batteries with larger capacity, and lighter and more durable materials.

### NOTE

Read more about how Microsoft quantum-inspired computing is being used in [materials science](#), [risk management](#), and [medicine](#).

## Quantum machine learning

Machine learning on classical computers is revolutionizing the world of science and business. However, the high computational cost of training the models hinders the development and scope of the field. The area of quantum machine learning explores how to devise and implement quantum software that enables machine learning that runs faster than classical computers.

The Quantum Development Kit comes with the [quantum machine learning library](#) that gives you the ability to run hybrid quantum/classical machine learning experiments. The library includes samples and tutorials, and provides the necessary tools to implement a new hybrid quantum–classical algorithm, the circuit-centric quantum classifier, to solve supervised classification problems.

## Q# and the Microsoft Quantum Development Kit (QDK)

Q# is Microsoft's open-source programming language for developing and running quantum algorithms. It is

part of the [QDK](#), a full-featured development kit for Q# that you can use with standard tools and languages to develop quantum applications that you can run in various environments, including the built-in full-state quantum simulator.

There are extensions for Visual Studio and VS Code, and packages for use with Python and Jupyter Notebook.

The QDK includes a standard library along with specialized chemistry, machine learning, and numerics libraries.

The documentation includes a Q# language guide, tutorials, and sample code to get you started quickly, and rich articles to help you dive deeper into quantum computing concepts.

## Microsoft quantum hardware partners

Microsoft is partnering with quantum hardware companies to provide developers with cloud access to quantum hardware. Leveraging the [Azure Quantum](#) platform and the Q# language, developers will be able to explore quantum algorithms and run their quantum programs on different types of quantum hardware.

[IonQ](#) and [Honeywell](#) both use **trapped ion-based** processors, utilizing individual ions trapped in an electronic field, whereas [QCI](#) uses superconducting circuits.

## Next steps

[Key concepts for quantum computing Quickstarts](#)

# Understanding quantum computing

3/5/2021 • 5 minutes to read • [Edit Online](#)

Quantum computing uses the principles of quantum mechanics to process information. Because of this, quantum computing requires a different approach than classical computing. One example of this difference is the processor used in quantum computers. Where classical computers use familiar silicon-based chips, quantum computers use quantum systems such as atoms, ions, photons, or electrons. They use their quantum properties to represent bits that can be prepared in different quantum superpositions of 1 and 0.

The quantum material behaves according to the laws of quantum mechanics, leveraging concepts such as probabilistic computation, superposition, and entanglement. These concepts provide the basis for quantum algorithms that harness the power of quantum computing to solve complex problems. This article describes some of the essential concepts of quantum mechanics on which quantum computing is based.

## A bird's-eye view of quantum mechanics

Quantum mechanics, also called quantum theory, is a branch of physics that deals with particles at the atomic and subatomic levels. At the quantum level, however, many of the laws of mechanics you take for granted don't apply. Superposition, quantum measurement, and entanglement are three phenomena that are central to quantum computing.

## Superposition vs. binary computing

Imagine that you are exercising in your living room. You turn all the way to your left and then all the way to your right. Now turn to your left and your right at the same time. You can't do it (not without splitting yourself in two, at least). Obviously, you can't be in both of those states at once – you can't be facing left and facing right at the same time.

However, if you are a quantum particle, then you can have a certain probability of *facing left* AND a certain probability of *facing right* due to a phenomenon known as **superposition** (also known as **coherence**).

A quantum particle such as an electron has its own "facing left or facing right" properties, for example **spin**, referred to as either up or down, or to make it more relatable to classical binary computing, let's just say 1 or 0. When a quantum particle is in a superposition state, it's a linear combination of an infinite number of states between 1 and 0, but you don't know which one it will be until you actually look at it, which brings up our next phenomenon, **quantum measurement**.

## Quantum measurement

Now, let's say your friend comes over and wants to take a picture of you exercising. Most likely, they'll get a blurry image of you turning somewhere between all the way left and all the way right.

But if you're a quantum particle, an interesting thing happens. No matter where you are when they take the picture, it will always show you either all the way left or all the way right – nothing in-between.

This is because the act of observing or measuring a quantum particle **collapses** the superposition state (also known as **decoherence**) and the particle takes on a classical binary state of either 1 or 0.

This binary state is helpful to us, because in computing you can do lots of things with 1's and 0's. However, once a quantum particle has been measured and collapsed, it stays in that state forever (just like your picture) and will always be a 1 or 0. As you'll see later, though, in quantum computing there are operations that can "reset" a particle back to a superposition state so it can be used for quantum calculations again.

# Entanglement

Possibly the most interesting phenomenon of quantum mechanics is the ability of two or more quantum particles to become **entangled** with each other. When particles become entangled, they form a single system such that the quantum state of any one particle cannot be described independently of the quantum state of the other particles. This means that whatever operation or process you apply to one particle correlates to the other particles as well.

In addition to this interdependency, particles can maintain this connection even when separated over incredibly large distances, even light-years. The effects of quantum measurement also apply to entangled particles, such that when one particle is measured and collapses, the other particle collapses as well. Because there is a correlation between the entangled qubits, measuring the state of one qubit provides information about the state of the other qubit – this particular property is very helpful in quantum computing.

## Qubits and probability

Classical computers store and process information in bits, which can have a state of either 1 or 0, but never both. The equivalent in quantum computing is the **qubit**, which represents the state of a quantum particle. Because of superposition, qubits can either be 1 or 0 or anything in between. Depending on its configuration, a qubit has a certain *probability* of collapsing to 1 or 0. The qubit's probability of collapsing one way or the other is determined by **quantum interference**.

Remember your friend that was taking your picture? Suppose they have special filters on their camera called *Interference* filters. If they select the *70/30* filter and start taking pictures, in 70% of them you will be facing left, and in 30% you will be facing right. The filter has interfered with the regular state of the camera to influence the probability of its behavior.

Similarly, quantum interference affects the state of a qubit in order to influence the probability of a certain outcome during measurement, and this probabilistic state is where the power of quantum computing excels.

For example, with two bits in a classical computer, each bit can store 1 or 0, so together you can store four possible values – **00, 01, 10, and 11** – but only one of those at a time. With two qubits in superposition, however, each qubit can be 1 or 0 or *both*, so you can represent the same four values simultaneously. With three qubits, you can represent eight values, with four qubits, you can represent 16 values, and so on.

## Summary

These concepts just scratch the surface of quantum mechanics, but are fundamentally important concepts to know for quantum computing.

- **Superposition** - The ability of quantum particles to be a combination of all possible states.
- **Quantum measurement** - The act of observing a quantum particle in superposition and resulting in one of the possible states.
- **Entanglement** - The ability of quantum particles to correlate their measurement results with each other.
- **Qubit** - The basic unit of information in quantum computing. A qubit represents a quantum particle in superposition of all possible states.
- **Interference** - Intrinsic behavior of a qubit due to superposition to influence the probability of it collapsing one way or another.

## Next Steps

[Quantum computers and quantum simulators](#)

# Quantum computers and quantum simulators

3/5/2021 • 3 minutes to read • [Edit Online](#)

Quantum computers are still in the infancy of their development. The hardware and maintenance are expensive, and most systems are located in universities and research labs. The technology is advancing, though, and limited public access to some systems is available.

Quantum simulators are software programs that run on classical computers and make it possible to run and test quantum programs in an environment that predicts how qubits react to different operations.

## Quantum hardware

A quantum computer has three primary parts: an area that houses the qubits, a method for transferring signals to the qubits, and a classical computer to run a program and send instructions.

- The quantum material used for qubits is fragile and highly sensitive to environmental interferences. For some methods of qubit storage, the unit that houses the qubits is kept at a temperature just above absolute zero to maximize their coherence. Other types of qubit housing use a vacuum chamber to help minimize vibrations and stabilize the qubits.
- Signals can be sent to the qubits using a variety of methods including microwaves, laser, and voltage.

Quantum computers face a multitude of challenges to operate correctly. Error correction in quantum computers is a significant issue, and scaling up (adding more qubits) increases the error rate. Because of these limitations, a quantum PC for your desktop is far in the future, but a commercially-viable lab-based quantum computer is closer.

## Quantum simulators

Quantum simulators that run on classical computers allow you to simulate the running of quantum algorithms on a quantum system. Microsoft's Quantum Development Kit (QDK) includes a full-state vector simulator along with other specialized quantum simulators.

## Topological qubit

Microsoft is developing a quantum computer based on topological qubits. A topological qubit is less impacted by changes in its environment, therefore reducing the degree of external error correction required.

Topological qubits feature increased stability and resistance to environmental noise, which means they can more readily scale and remain reliable longer.

## Microsoft and quantum hardware partnerships

Microsoft is partnering with quantum hardware manufacturers IonQ, Honeywell, and QCI to make quantum computers accessible to developers in the future. Leveraging the Azure Quantum platform, developers can use Microsoft's Quantum Development Kit (QDK) and Q# to write quantum programs and run them remotely.

## Quantum computations

Performing computations on a quantum computer or quantum simulator follows a basic process:

- Access the qubits

- Initialize the qubits to the desired state
- Perform operations to transform the states of the qubits
- Measure the new states of the qubits

Initializing and transforming qubits is done using **quantum operations** (sometimes called quantum gates). Quantum operations are similar to logic operations in classical computing, such as AND, OR, NOT, and XOR. An operation can be as basic as flipping a qubit's state from 1 to 0 or entangling a pair of qubits, to using multiple operations in series to affect the probability of a superposed qubit collapsing one way or the other.

#### NOTE

The [Q# libraries](#) provide built-in operations that define complex combinations of lower-level quantum operations. You can use the library operations to transform qubits and to create more complex user-defined operations.

Measuring the result of the computation tells us an answer, but for some quantum algorithms, not necessarily the correct answer. Because the result of some quantum algorithms is based on the probability that was configured by the quantum operations, these computations are run multiple times to get a probability distribution and refine the accuracy of the results. Assurance that an operation returned a correct answer is known as quantum verification and is a significant challenge in quantum computing.

## Summary

Quantum computing shares some of the same concepts as classical computing but adds a few new twists. Here are some key takeaways:

- Quantum hardware is expensive and fragile to work with, so quantum simulators are used to write and test programs.
- Both classical and quantum computers use logic operations (or gates) to prepare computations.
- Quantum computations return probabilities.

Advancements in quantum hardware and techniques is rapidly changing the field. Here are just a few of the [current developments](#).

## Next steps

[What are the Q# programming language and QDK?](#)

# What are the Q# programming language and Quantum Development Kit (QDK)?

3/5/2021 • 3 minutes to read • [Edit Online](#)

Q# is Microsoft's open-source programming language for developing and running quantum algorithms. It's part of the Quantum Development Kit (QDK), which includes [Q# libraries](#), [quantum simulators](#), [extensions for other programming environments](#), and [API documentation](#). In addition to the Standard Q# library, the QDK includes Chemistry, Machine Learning, and Numeric libraries.

As a programming language, Q# draws familiar elements from Python, C#, and F# and supports a basic procedural model for writing programs with loops, if/then statements, and common data types. It also introduces new quantum-specific data structures and operations.

## What can I do with the QDK?

The QDK is a full-featured development kit for Q# that you can use with common tools and languages to develop quantum applications that you can run in various environments. Q# programs can run as a console application, through Jupyter Notebooks, or use a Python or .NET host program.

### Develop in common tools and environments

Integrate your quantum development with [Visual Studio](#), [Visual Studio Code](#), and [Jupyter Notebooks](#). Use the built-in APIs for pairing your programs with [Python](#) and [.NET](#) host languages.

### Try quantum operations and domain-specific libraries

Write and test quantum algorithms to explore superposition, entanglement, and other quantum operations. The Q# libraries enable you to run complex quantum operations without having to design low-level operation sequences.

### Run programs in simulators

Run your quantum programs on a full-state quantum simulator, a limited-scope Toffoli simulator, or test your Q# code in different resource estimators.

## Where can I learn more?

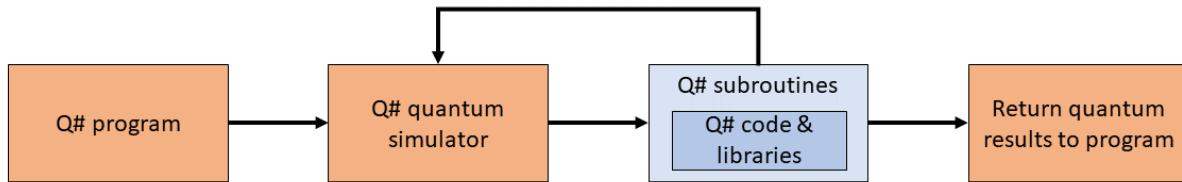
I'm new to quantum computing	Review some basics of quantum physics and quantum computing in <a href="#">Key Concepts</a> .
I want to dive deeper into the Q# language	Explore types, expressions, variables, and quantum program structure in the <a href="#">Q# User Guide</a> .
I just want to start writing quantum programs	Set up your Q# environment and start writing quantum programs in <a href="#">QuickStarts</a> .

## How does Q# work?

A Q# program can compile into a standalone application or be called by a host program that is written either in Python or a .NET language.

When you compile and run the program, it creates an instance of the quantum simulator and passes the Q# code to it. The simulator uses the Q# code to create qubits (simulations of quantum particles) and apply transformations to modify their state. The results of the quantum operations in the simulator are then returned to the program.

Isolating the Q# code in the simulator ensures that the algorithms follow the laws of quantum physics and can run correctly on quantum computers.



## How do I use the QDK?

Everything you need to write and run Q# programs, including the Q# compiler, the Q# libraries, and the quantum simulators, can be installed and run from your local computer. Eventually you will be able to run your Q# programs remotely on an actual quantum computer, but until then the quantum simulators provided with the QDK provide accurate and reliable results.

- Developing [Q# applications](#) is the quickest way to get started.
- Run standalone [Jupyter Notebooks with IQ#](#), a Jupyter extension for compiling, simulating, and visualizing Q# programs.
- If you are familiar with [Python](#), you can use it as a host programming platform to get started. Python enjoys widespread use not only among developers, but also scientists, researchers and teachers.
- If you already have experience with [C#, F#, or VB.NET](#) and are familiar with the Visual Studio development environment, there are just a few extensions you need to add to Visual Studio to prepare it for Q#.

## Summary

Q# is an open-source programming language for developing quantum programs. It has libraries that let you create complex quantum operations, and quantum simulators to accurately run and test your programs. Q# programs can run as standalone apps or be called from a Python or .NET host program, and can be written, run, and tested from your local computer.

## Next Steps

[Linear algebra for quantum computing](#)

# Linear algebra for quantum computing

3/5/2021 • 4 minutes to read • [Edit Online](#)

Linear algebra is the language of quantum computing. Although you don't need to know it to implement or write quantum programs, it is widely used to describe qubit states, quantum operations, and to predict what a quantum computer does in response to a sequence of instructions.

Just like being familiar with the [basic concepts of quantum physics](#) can help you understand quantum computing, knowing some basic linear algebra can help you understand how quantum algorithms work. At the least, you'll want to be familiar with **vectors** and **matrix multiplication**. If you need to refresh your knowledge of these algebra concepts, here are some tutorials that cover the basics:

- [Jupyter notebook tutorial on linear algebra](#)
- [Jupyter notebook tutorial on complex arithmetic](#)
- [Linear Algebra for Quantum Computation](#)
- [Fundamentals of Linear Algebra](#)
- [Quantum Computation Primer](#)

## Vectors and matrices in quantum computing

In the topic [Understanding quantum computing](#), you saw that a qubit can be in a state of 1 or 0 or a superposition or both. Using linear algebra, the state of a qubit is described as a vector and is represented by a single column **matrix**  $\begin{bmatrix} a \\ b \end{bmatrix}$ . It is also known as a **quantum state vector** and must meet the requirement that  $|a|^2 + |b|^2 = 1$ .

The elements of the matrix represent the probability of the qubit collapsing one way or the other, with  $|a|^2$  being the probability of collapsing to zero, and  $|b|^2$  being the probability of collapsing to one. The following matrices all represent valid quantum state vectors:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$$

In [Quantum computers and quantum simulators](#) you also saw that quantum operations are used to modify the state of a qubit. Quantum operations can also be represented by a matrix. When a quantum operation is applied to a qubit, the two matrices that represent them are multiplied and the resulting answer represents the new state of the qubit after the operation.

Here are two common quantum operations represented with matrix multiplication.

The **X** operation is represented by the Pauli matrix  $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ ,

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

and is used to flip the state of a qubit from 0 to 1 (or vice-versa), for example

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The **H** operation is represented by the Hadamard transformation  $\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$ ,

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

and puts a qubit into a superposition state where it has an even probability of collapsing either way, as shown

here

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

Notice that  $|a|^2 = |b|^2 = \frac{1}{2}$ , meaning that the probability of collapsing to zero and one state is the same.

A matrix that represents a quantum operation has one requirement – it must be a **unitary** matrix. A matrix is unitary if the **inverse** of the matrix is equal to the **conjugate transpose** of the matrix.

## Representing two-qubit states

In the examples above, the state of one qubit was described using a single column matrix  $\begin{bmatrix} a \\ b \end{bmatrix}$ , and applying an operation to it was described by multiplying the two matrices. However, quantum computers use more than one qubit, so how do you describe the combined state of two qubits?

Remember that each qubit is a vector space, so they can't just be multiplied. Instead, you use a **tensor product**, which is a related operation that creates a new vector space from individual vector spaces, and is represented by the  $\otimes$  symbol. For example, the tensor product of two qubit states  $\begin{bmatrix} a \\ b \end{bmatrix}$  and  $\begin{bmatrix} c \\ d \end{bmatrix}$  is calculated

$$\begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix}$$

The result is a four-dimensional matrix, with each element representing a probability. For example,  $|ac|^2$  is the probability of the two qubits collapsing to 0 and 0,  $|ad|^2$  is the probability of 0 and 1, and so on.

Just as a single qubit state  $\begin{bmatrix} a \\ b \end{bmatrix}$  must meet the requirement that  $|a|^2 + |b|^2 = 1$  in order to represent a quantum state, a two-qubit state  $\begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix}$  must meet the requirement that  $|ac|^2 + |ad|^2 + |bc|^2 + |bd|^2 = 1$ .

## Summary

Linear algebra is the standard language for describing quantum computing and quantum physics. Even though the [libraries](#) included with the Microsoft Quantum Development Kit helps you run advanced quantum algorithms without diving into the underlying math, understanding the basics helps you get started quickly and provide a solid foundation to build on.

## Next steps

[Install the QDK](#)

# Set up the Quantum Development Kit (QDK)

6/1/2021 • 2 minutes to read • [Edit Online](#)

Learn how to set up the Quantum Development Kit (QDK) to develop quantum computing and optimization applications on your environment.

The QDK consists of:

- The Q# programming language
- A set of libraries that abstract complex functionality in Q#
- APIs for Python and .NET languages (C#, F#, and VB.NET) for running quantum programs written in Q#
- A Python SDK to use optimization solvers on Azure Quantum
- Tools to facilitate your development

## Set up the Quantum Development Kit to develop quantum computing applications in Q#

Q# programs can run as standalone applications using Visual Studio Code or Visual Studio, through Jupyter Notebooks with the IQ# Jupyter kernel, or paired with a host program written in Python or a .NET language (C#, F#). You can also run Q# programs online using [MyBinder.org](#), or [Docker](#).

### Options for setting up the QDK for quantum computing

You can use the QDK in three ways:

- [Install the QDK for quantum computing locally](#)
- [Use the QDK for quantum computing online](#)
- [Use a QDK for quantum computing Docker image](#)

### Install the QDK for quantum computing locally

You can develop Q# code in most of your favorites IDEs, as well as integrate Q# with other languages such as Python and .NET (C#, F#).

		 VS CODE (2019 OR LATER)	 VISUAL STUDIO (2019 OR LATER)	 JUPYTER NOTEBOOKS	COMMAND LINE
	OS support:	Windows, macOS, Linux	Windows only	Windows, macOS, Linux	Windows, macOS, Linux
	Q# standalone	<a href="#">Install</a>	<a href="#">Install</a>	<a href="#">Install</a>	<a href="#">Install</a>
	Q# and Python	<a href="#">Install</a>	<a href="#">Install</a>	<a href="#">Install</a>	<a href="#">Install</a>
	Q# and .NET (C#, F#)	<a href="#">Install</a>	<a href="#">Install</a>	✗	<a href="#">Install</a>

## Use the QDK for quantum computing Online

You can also develop Q# code without installing anything locally with these options:

RESOURCE	ADVANTAGES	LIMITATIONS
<a href="#">Binder</a>	Free online notebook experience	No persistence

## Use the QDK for quantum computing with Docker

You can use our QDK Docker image in your local Docker installation or in the cloud via any service that supports Docker images, such as ACI.

You can download the IQ# Docker image from <https://github.com/microsoft/iqsharp/#using-iq-as-a-container>.

You can also use Docker with a Visual Studio Code Remote Development Container to quickly define development environments. For more information about VS Code Development Containers, see <https://github.com/microsoft/Quantum/tree/master/.devcontainer>.

The workflows for each of these setups are described and compared in [Ways to run a Q# program](#).

## Set up the Quantum Development Kit to manage optimization solvers in Azure Quantum

You can use the Python SDK of the Quantum Development Kit to solve optimization problems using Azure Quantum solvers.

To set up the Python SDK, follow the steps in [Install and use the Python SDK for Azure Quantum](#).

# Get started with the Quantum Development Kit (QDK) for quantum computing

5/27/2021 • 4 minutes to read • [Edit Online](#)

The Quantum Development Kit (QDK) contains all the tools you'll need to build your own quantum programs and experiments with Q#, a programming language designed specifically for quantum application development.

To jump right in, start with the [QDK setup guide](#). You'll be guided through setting up the Quantum Development Kit on Windows, Linux, or MacOS machines so that you can write your own quantum programs.

If you're new to quantum computing, review the [Overview](#) section to learn what quantum computers can do and the fundamentals of quantum computing.

## Get started programming with Q#

The Quantum Development Kit provides many ways to learn how to develop a quantum program with Q#. To get up and running with the power of quantum, you can try out our tutorials:

- [Quantum random number generator](#) - Start with a "Q# Hello World" style application, providing a brief introduction to quantum concepts while letting you build and run a quantum application in minutes.
- [Explore entanglement with Q#](#) - This tutorial guides you on writing a Q# program that demonstrates some of the foundational concepts of quantum programming. If you are not ready to start coding, you can still follow along without installing the QDK and get an overview of the Q# programming language and the first concepts of quantum computing.
- [Grover's search algorithm](#) - Explore this example of a Q# program to get an idea of the power of Q# for expressing the quantum algorithm in a way that abstracts the low-level quantum operations. This tutorial guides you through developing the program as a Q# application, using Visual Studio or Visual Studio Code.

## Learning further

- Microsoft Learn offers free online training for quantum computing. The [Quantum computing foundations](#) Learning Path introduces the fundamental concepts of quantum computing and quantum algorithms, and gets you started building quantum programs using Q#.
- If you want to dive deeper into Q# programming, check out the [Quantum Katas](#) - a collection of self-paced programming exercises that introduce you to quantum computing via programming exercises in Q#. Many of these katas are also available as Q# Notebooks.
- Our [samples repository](#) showcases multiple examples on how to write quantum programs using Q#. Most of these samples are written using our open-source [quantum libraries](#), including our [standard](#) and [chemistry](#) libraries (more info on these below).

## Key concepts for quantum computing

If you are a newcomer to quantum development, we know that this can all seem a bit daunting. These key concepts are designed to help you step into the quantum world and understand how quantum computing differs from classical computing.

- [Understanding quantum computing](#)
- [Quantum computers and quantum simulators](#)
- [What are the Q# programming language and the QDK?](#)

- [Linear algebra for quantum computing](#)

## Quantum Development Kit Documentation

The current documentation includes the following additional topics.

### **Q# developer guides**

- [Q# User Guide](#) details the core concepts used to create quantum programs in Q#.
- [Quantum simulators and host applications](#) describes how quantum algorithms are run, what quantum machines are available, and how to write a non-Q# driver for the quantum program.

### **Q# libraries**

- [Q# standard libraries](#) describes the operations and functions that support both the classical language control requirement and the Q# quantum algorithms. Topics include control flow, data structures, error correction, testing, and debugging.
- [Q# chemistry library](#) describes the operations and functions that support quantum chemistry simulation---a critical application of quantum computing. Topics include simulating Hamiltonian dynamics and quantum phase estimation, among others.
- [Q# numerics library](#) describes the operations and functions that support expressing complicated arithmetic functions in terms of the native operations of target machines.
- [Q# library reference](#) contains reference information about library entities by namespace.

### **General quantum computing**

- [Quantum computing concepts](#) includes topics such as the relevance of linear algebra to quantum computing, the nature and use of a qubit, how to read a quantum circuit, and more.
- [Quantum computing glossary](#) is a glossary of some crucial terms specific to quantum computing and program development. If you are new to the field, this could be a handy reference as you read through our documentation.
- [Further reading](#) contains specially selected references for in-depth coverage of quantum computing topics.

### **Additional info**

- [Microsoft Quantum Development kit release notes](#).

## Be a part of the Q# Open-Source Community

The Quantum Development Kit is an open-source development kit that empowers developers to make quantum computing more accessible to all so that we can solve some of the world's most pressing challenges. Academic institutions who require open-source software will be able to deploy Q# for their quantum learning and development. Open-sourcing the development kit also empowers developers and domain experts an opportunity to contribute improvements and ideas via their code.

Your feedback, participation and contributions to the Quantum Development Kit is important. To learn more about the Quantum Development Kit sources, provide feedback, and find out how you can participate in the decisions and contribute to this growing quantum development platform, see [Contributing to the Quantum Development Kit](#).

If you'd like more general information about Microsoft's quantum computing initiative, see [Microsoft Quantum](#).

# Develop with Q# applications in an IDE

6/2/2021 • 3 minutes to read • [Edit Online](#)

Learn how to develop Q# applications in Visual Studio Code (VS Code), Visual Studio, or with any editor/IDE and run applications from the .NET console. Q# programs can run on their own, without a driver in a host language like C#, F#, or Python.

## Prerequisites for all environments

- [.NET Core SDK 3.1](#)

## Installation

While you can build Q# applications in any IDE, we recommend using Visual Studio Code (VS Code) or Visual Studio IDE for developing your Q# applications locally. Developing in these environments leverages the rich functionality of the Quantum Development Kit (QDK) extension, which includes warnings, syntax highlighting, project templates, and more.

### IMPORTANT

If you are working on Linux, you may encounter a missing dependency depending on your particular distribution and installation method (e.g. certain Docker images). Please make sure that the `libgomp` library is installed on your system, as the GNU OpenMP support library is required by the quantum simulator of the QDK. On Ubuntu, you can do so by running `sudo apt install libgomp1`, or `yum install libgomp` on CentOS. For other distributions, please refer to your particular package manager.

Configure the QDK for your preferred environment from one of the following options:

- [VS Code](#)
- [Visual Studio \(Windows only\)](#)
- [Other editors with the command prompt](#)

1. Download and install [VS Code](#) 1.52.0 or greater (Windows, Linux and Mac).
2. Install the [QDK for VS Code](#).

## Develop with Q#

Follow the instructions on the tab corresponding to your development environment.

- [VS Code](#)
- [Visual Studio \(Windows only\)](#)
- [Other editors with the command prompt](#)

If you are receiving an error "'npm' is not recognized as an internal or external command", in the below steps, install [node.js including npm](#). Alternatively, use our the command line templates to create a Q# project , or use Visual Studio.

To create a new project:

1. Click **View -> Command Palette** and select **Q#: Create New Project**.

2. Click **Standalone console application**.
3. Navigate to the location to save the project. Enter the project name and click **Create Project**.
4. When the project is successfully created, click **Open new project...** in the lower right.

Inspect the project. You should see a source file named `Program.qs`, which is a Q# program that defines a simple operation to print a message to the console.

To run the application:

1. Click **Terminal -> New Terminal**.
2. At the terminal prompt, enter `dotnet run`.
3. You should see the following text in the output window `Hello quantum world!`

**NOTE**

Workspaces with multiple root folders are not currently supported by the VS Code Q# extension. If you have multiple projects within one VS Code workspace, all projects need to be contained within the same root folder.

## Next steps

Now that you have installed the Quantum Development Kit in your preferred environment, you can write and run [your first quantum program](#).

# Develop with Q# Jupyter Notebooks

7/13/2021 • 5 minutes to read • [Edit Online](#)

This article shows you how to install the Quantum Developer Kit (QDK) for developing Q# operations on Q# Jupyter Notebooks. Then, learn how to start developing your own Q# notebooks.

Jupyter Notebooks allow running code in-place alongside instructions, notes, and other content. This environment is ideal for writing Q# code with embedded explanations or quantum computing interactive tutorials.

## Install the IQ# Jupyter kernel

IQ# (pronounced i-q-sharp) is an extension primarily used by Jupyter and Python to the .NET Core SDK that provides the core functionality for compiling and simulating Q# operations.

- [Install using conda \(recommended\)](#)
- [Install using .NET CLI \(advanced\)](#)

1. Install [Miniconda](#) or [Anaconda](#). Consult their [installation guide](#) if you are unsure about any steps. **Note:** 64-bit installation required.

2. Initialize conda for your preferred shell with the `conda init` command. The steps below are tailored to your operating system:

**(Windows)** Open an Anaconda Prompt by searching for it in the start menu. Then run the initialization command for your shell, e.g. `conda init powershell cmd.exe` will set up both the Windows PowerShell and Command Prompt for you. You can then close this prompt.

### IMPORTANT

To work with PowerShell, conda will configure a startup script to run whenever you launch a PowerShell instance. By default, the script's execution will be blocked on Windows, and requires modifying the PowerShell execution policy with the following command (executed from within PowerShell):

```
Set-ExecutionPolicy -Scope CurrentUser RemoteSigned
```

**(Linux)** If haven't done so during installation, you can still initialize conda now. Open a terminal and navigate to the `bin` directory inside your selected install location (e.g. `/home/ubuntu/miniconda3/bin`). Then run the appropriate command for your shell, e.g. `./conda init bash`. Close your terminal for the changes to take effect.

3. From a new terminal, create and activate a new conda environment named `qsharp-env` with the required packages (including Jupyter Notebook and IQ#) by running the following commands:

```
conda create -n qsharp-env -c quantum-engineering qsharp notebook  
conda activate qsharp-env
```

4. Finally, run `python -c "import qsharp"` to verify your installation and populate your local package cache with all required QDK components.

That's it! You now have the IQ# kernel for Jupyter, allowing you to compile and run Q# operations from Q# Jupyter Notebooks.

## Create your first Q# notebook

Now you are ready to verify your Q# Jupyter Notebook installation by writing and running a simple Q# operation.

1. From the environment you created during installation (that is, either the conda environment you created, or the Python environment where you installed Jupyter), run the following command to start the Jupyter Notebook server:

```
jupyter notebook
```

- If the Jupyter Notebook doesn't open automatically in your browser, copy and paste the URL provided by the command line into your browser.

2. Choose **New → Q#** to create a Jupyter Notebook with a Q# kernel, and add the following code to the first notebook cell:

```
operation SampleQuantumRandomNumberGenerator() : Result {
    use q = Qubit(); // Allocate a qubit in the |0⟩ state.
    H(q);           // Put the qubit to superposition. It now has a 50% chance of being 0 or 1.
    let r = M(q);   // Measure the qubit value.
    Reset(q);
    return r;
}
```

### NOTE

Callable from [Microsoft.Quantum.Intrinsic](#) and [Microsoft.Quantum.Canon](#) (for example, the `H` operation belongs to `Microsoft.Quantum.Intrinsic` namespace) are automatically available to operations defined within cells in Q# Jupyter Notebooks. You don't need to open those namespaces in your Q# program. However, this is not true for code brought in from external Q# source files (a process shown at [Intro to Q# and Jupyter Notebooks](#)).

1. Run this cell of the notebook. You should see `SampleQuantumRandomNumberGenerator` in the output of the cell. When running in Jupyter Notebook, the Q# code is compiled, and the cell outputs the name of any operations that it finds.

2. In a new cell, run the operation you just created in a simulator by using the `%simulate` magic command:

```
In [2]: ┌ %simulate SampleQuantumRandomNumberGenerator
          Out[2]: One
```

You should see the result of the operation you invoked. In this case, because your operation generates a random result, you will see either `Zero` or `One` printed on the screen. If you run the cell repeatedly, you should see each result approximately half the time.

### NOTE

In a Jupyter Notebook, Q# code is automatically wrapped in a namespace for you. Since Q# does not feature nested namespaces, you should make sure not to declare any additional namespaces inside of code cells.

## Next steps

Now that you have installed the QDK for Q# Jupyter Notebooks, you can write and run [your first quantum program](#) by writing Q# code directly within the Jupyter Notebook environment.

For more examples of what you can do with Q# Jupyter Notebooks, please take a look at:

- [Intro to Q# and Jupyter Notebook](#). There you will find a Q# Jupyter Notebook that provides more details on how to use Q# in the Jupyter environment.
- [Quantum Katas](#), an open-source collection of self-paced tutorials and sets of programming exercises in the form of Q# Jupyter Notebooks. The [Quantum Katas tutorial notebooks](#) are a good starting point. The Quantum Katas are aimed at teaching you elements of quantum computing and Q# programming at the same time. They're an excellent example of what kind of content you can create with Q# Jupyter Notebooks.

# Develop with Q# and Python

6/14/2021 • 5 minutes to read • [Edit Online](#)

Learn how to install the Quantum Development Kit (QDK) to develop Python host programs that call Q# operations.

## Install the `qsharp` Python package

The `qsharp` Python package, which includes the IQ# kernel, contains the necessary functionality for compiling and simulating Q# operations from a regular Python program.

- [Install using conda \(recommended\)](#)
- [Install using .NET CLI and pip \(advanced\)](#)

1. Install [Miniconda](#) or [Anaconda](#). Consult their [installation guide](#) if you are unsure about any steps. **Note:** 64-bit installation required.

2. Initialize conda for your preferred shell with the `conda init` initialization command. The steps below are tailored to your operating system:

**(Windows)** Open an Anaconda Prompt by searching for it in the start menu. Then run the initialization command for your shell, e.g. `conda init powershell cmd.exe` will set up both the Windows PowerShell and Command Prompt for you. You can then close this prompt.

### IMPORTANT

To work with PowerShell, conda will configure a startup script to run whenever you launch a PowerShell instance. By default, the script's execution will be blocked on Windows, and requires modifying the PowerShell execution policy with the following command (executed from within PowerShell):

```
Set-ExecutionPolicy -Scope CurrentUser RemoteSigned
```

**(Linux)** If haven't done so during installation, you can still initialize conda now. Open a terminal and navigate to the `bin` directory inside your selected install location (e.g. `/home/ubuntu/miniconda3/bin`). Then run the appropriate command for your shell, e.g. `./conda init bash`. Close your terminal for the changes to take effect.

3. From a new terminal, create and activate a new conda environment named `qsharp-env` with the required packages (including Jupyter Notebook and IQ#) by running the following commands:

```
conda create -n qsharp-env -c quantum-engineering qsharp notebook  
conda activate qsharp-env
```

4. Finally, run `python -c "import qsharp"` to verify your installation and populate your local package cache with all required QDK components.

That's it! You now have both the `qsharp` Python package and the IQ# kernel for Jupyter, allowing you to compile and run Q# operations from Python and Q# Jupyter Notebooks.

## Choose your IDE

While you can use Q# with Python in any IDE, we highly recommend using Visual Studio Code (VS Code) for your Q# + Python applications. With the QDK extension for VS Code you gain access to richer functionality such as warnings, syntax highlighting, project templates, and more.

If you would like to use VS Code:

- Install [VS Code](#) (Windows, Linux and Mac).
- Install the [QDK extension for VS Code](#).

VS Code also offers its own terminal from which you can run code. If you are using conda, make sure you follow the procedure detailed in the installation section to initialize conda for the shell used by VS Code. On Windows, VS Code will use PowerShell unless configured differently. Doing so will allow you to run *Q# with Python* programs directly from VS Code's integrated terminal, however you can use any terminal of your choice with access to Python. Remember to activate your Q# environment there before running any programs, using `conda activate qsharp-env`.

If you would like to use a different editor, the instructions so far have you all set.

## Write your first Q# program

Now you are ready to verify your `qsharp` Python package installation by writing a simple Q# program and running it on a quantum [simulator](#).

1. Create a minimal Q# operation by creating a file called `operation.qs` and adding the following code to it:

```
namespace Qrng {
    open Microsoft.Quantum.Intrinsic;

    operation SampleQuantumRandomNumberGenerator() : Result {
        use q = Qubit(); // Allocate a qubit.
        H(q);           // Put the qubit to superposition. It now has a 50% chance of being 0 or 1.
        let r = M(q);   // Measure the qubit value.
        Reset(q);
        return r;
    }
}
```

2. In the same folder as `operation.qs`, create the following Python program called `host.py`. This program imports the Q# operation `SampleQuantumRandomNumberGenerator()` defined in step 1 and runs it on the default simulator with a `.simulate()` call:

```
import qsharp
from Qrng import SampleQuantumRandomNumberGenerator

print(SampleQuantumRandomNumberGenerator.simulate())
```

3. From a terminal with access to your Python/Q# environment created during installation, navigate to your project folder and run the Python host program:

```
python host.py
```

4. You should see the result of the operation you invoked. In this case, because your operation generates a random result, you will see either `0` or `1` printed on the screen. If you run the program repeatedly, you should see each result approximately half the time.

#### **NOTE**

The Python code is just a normal Python program. You can use any Python environment, including Python-based Jupyter Notebooks, to write the Python program and call Q# operations. The Python program can import Q# operations from any .qs files located in the same folder as the Python code itself.

## Next steps

Now that you have tested the Quantum Development Kit in your preferred environment, you can follow this tutorial to write and run [your first quantum program](#).

For more information on how to run Q# programs with Python, see the following articles:

- how [Q# interacts with a Python host program](#)
- how to [run Q# on a local simulator](#)
- how to [run Q# on quantum hardware](#) through Azure Quantum
- how to first [estimate quantum resources](#) required by your program

# Develop with Q# and .NET

3/5/2021 • 3 minutes to read • [Edit Online](#)

The Q# programming language is built to work well with .NET languages such as C# and F#. In this guide, we demonstrate how to use Q# with a host program written in a .NET language.

First we create the Q# application and .NET host, and then demonstrate how to call to Q# from the host.

## Prerequisites

- Install the Quantum Development Kit (QDK) [for use with Q# projects](#).

## Creating a Q# library and a .NET host

The first step is to create projects for your Q# library, and for the .NET host that will call into the operations and functions defined in your Q# library.

Follow the instructions in the tab corresponding to your development environment. If you are using an editor other than Visual Studio or VS Code, simply follow the command prompt steps.

- [Visual Studio Code or command prompt](#)
- [Visual Studio 2019](#)
- Create a new Q# library

```
dotnet new classlib -lang Q# -o quantum
```

- Create a new C# or F# console project

```
dotnet new console -lang C# -o host
```

- Add your Q# library as a reference from your host program

```
cd host  
dotnet add reference ../quantum/quantum.csproj
```

- [Optional] Create a solution for both projects

```
dotnet new sln -n quantum-dotnet  
dotnet sln quantum-dotnet.sln add ./quantum/quantum.csproj  
dotnet sln quantum-dotnet.sln add ./host/host.csproj
```

## Calling into Q# from .NET

Once you have your projects set up following the above instructions, you can call into Q# from your .NET console application. The Q# compiler will create .NET classes for each Q# operation and function that allow you to run your quantum programs on a simulator.

For example, the [.NET interoperability sample](#) includes the following example of a Q# operation:

```

/// Instantiates the oracle and runs the parameter restoration algorithm.
operation RunAlgorithm(bits : Bool[]) : Bool[] {
    Message("Hello, quantum world!");
    // construct an oracle using the input array
    let oracle = ApplyProductWithNegationFunction(bits, _, _);
    // run the algorithm on this oracle and return the result
    return ReconstructOracleParameters(Length(bits), oracle);
}

```

To call this operation from .NET on a quantum simulator, you can use the `Run` method of the `RunAlgorithm` .NET class generated by the Q# compiler:

- [C#](#)
- [F#](#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using static System.Diagnostics.Debug;

using Microsoft.Quantum.Simulation.Core;
using Microsoft.Quantum.Simulation.Simulators;

namespace Microsoft.Quantum.Samples
{
    static class Program
    {
        static async Task Main(string[] args)
        {
            var bits = new[] { false, true, false };
            using var sim = new QuantumSimulator();

            Console.WriteLine($"Input: {bits.ToDelimitedString()}");

            var restored = await RunAlgorithm.Run(sim, new QArray<bool>(bits));
            Console.WriteLine($"Output: {restored.ToDelimitedString()}");

            Assert(bits.Parity() == restored.Parity());
        }

        static bool Parity(this IEnumerable<bool> bitVector) =>
            bitVector.Aggregate(
                (acc, next) => acc ^ next
            );

        static string ToDelimitedString(this IEnumerable<bool> values) =>
            String.Join(", ", values.Select(x => x.ToString()));
    }
}

```

## Next steps

Now that you have the Quantum Development Kit set up for both Q# applications and interoperability with .NET, you can write and run [your first quantum program](#).

# Develop with Q# and Binder

5/19/2021 • 3 minutes to read • [Edit Online](#)

Learn how to create a Q# application using Binder. You can use Binder to run and share Jupyter Notebooks online, and even run Q# console applications online, which allows you to try Q# without installing the QDK.

## Use Binder with the QDK samples

To configure Binder automatically to use the Quantum Development Kit (QDK) samples:

1. Open a browser and navigate to <https://aka.ms/try-qsharp>.
2. On the **Quantum Development Kit Samples** landing page, click the **Q# notebook** link on the **Intro to IQ#** sample to learn how to use IQ# and write your own quantum application notebooks.

Quantum Development Kit Samples			
	Sample	Run in browser...	Run at command line...
Getting started:	<a href="#">Intro to IQ#</a>	<a href="#">Q# notebook</a>	
	<a href="#">Measurement</a>	Q# standalone	
	<a href="#">Quantum random number generator</a>	Q# standalone	
	<a href="#">Simple quantum algorithms</a>	Q# standalone	
	<a href="#">Teleportation</a>	Q# standalone	
Algorithms:	<a href="#">CHSH Game</a>	<a href="#">Q# + Python</a>	<a href="#">Q# + .NET</a>
	<a href="#">Database Search</a>	<a href="#">Q# notebook</a> <a href="#">Q# + Python</a>	<a href="#">Q# + .NET</a>
	<a href="#">Integer factorization</a>	<a href="#">Q# + Python</a>	<a href="#">Q# + .NET</a>
	<a href="#">Oracle synthesis</a>	Q# standalone	

Note that you are not restricted to the existing samples, as you can create new notebook or text files by first selecting **File -> Open...** from the Jupyter interface to open the directory view, and then hitting the **New ▾** button in the top right of the page.

### WARNING

Files created in a Binder environment will not persist across sessions. Should you wish to preserve any changes or new files you created during your session, make sure to save them locally by downloading them via the Jupyter interface.

## Run Jupyter Notebook samples

Binder supports both types of Q# development styles for Jupyter Notebook:

- the *Q# notebook*, which uses the IQ# kernel to directly run code cells written in Q# (see [developing with Q# Jupyter Notebooks](#)).
- the *Q# + Python* notebook, which contains regular Python code that calls into Q# operations from a `.qs` file (see [developing with Q# and Python](#)).

You will find that the different Jupyter samples might use either of the two styles, and are sometimes available in both, so feel free to explore what best suites your preferences. You can create a notebook of your own by clicking on **New ▾ → Python 3 (Q# + Python style)** or **New ▾ → Q# (Q# notebook style)** from the directory

view.

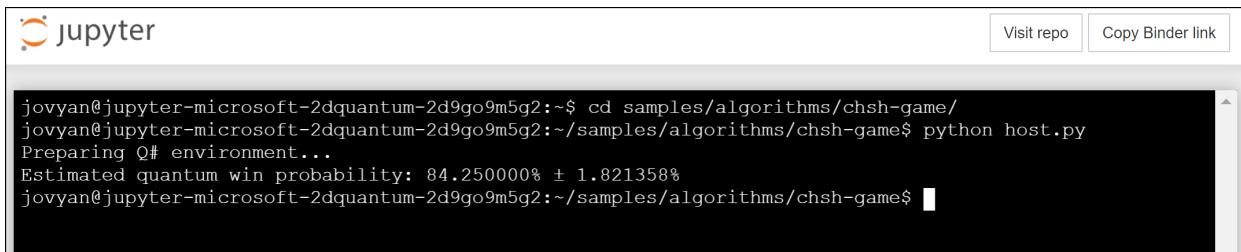


## Run console application samples

In addition to notebooks, you can also run Q# console applications via Binder. You'll notice that there are different types of console applications present in the samples:

- the *Q# standalone* application, which uses an `EntryPoint` function/operation in a `.qs` file to run the Q# program from the command line (see [developing with Q# applications](#)).
- the *Q# + .NET* application, which uses a .NET language (C# or F#) host program to call into operations from a `.qs` file (see [developing with Q# and .NET](#)).
- while not directly shown in the samples overview, a lot of samples are also available as Python console applications. These contain the same Python code as a *Q# + Python* Jupyter Notebook would, just without the notebook part (see [developing with Q# and Python](#)).

To run the samples, you can open a terminal in the Jupyter interface by selecting **New** → **Terminal** from the directory view. You may then run any bash commands from within the Binder environment, for example to run the CHSH sample via `python host.py` as shown below.



As an alternative to the Jupyter terminal, you can also run Q# console applications from a .NET PowerShell notebook in which the cells act like a PowerShell terminal. You can then run some of the samples via their README file by clicking on the sample name (e.g. the CHSH README shown below), or from your own notebook via **New** → **.NET (PowerShell)**.



# Use Binder with your own notebooks and repository

If you already have notebooks in a (public!) GitHub repository, you can configure Binder to work with your repo:

1. Ensure that there is a file named *Dockerfile* in the root of your repository. The file must contain at least the following lines:

```
FROM mcr.microsoft.com/quantum/iqsharp-base:0.12.20082513

USER root
COPY . ${HOME}
RUN chown -R ${USER} ${HOME}

USER ${USER}
```

## NOTE

You can verify the most current version of IQ# in the [Release Notes](#).

For more information about creating a Dockerfile, see the [Dockerfile reference](#).

2. Open a browser to [mybinder.org](#).
3. Enter your repository name as the **GitHub URL** (for example *MyName/MyRepo*), and click **launch**.

The screenshot shows the 'Build and launch a repository' interface. At the top, there's a header 'Build and launch a repository'. Below it, a form with several fields:

- A large input field labeled 'GitHub repository name or URL' containing 'MyName/MyRepo'. This field is highlighted with a red box.
- Below it are two smaller input fields: 'Git branch, tag, or commit' and 'Path to a notebook file (optional)'. The 'Git branch, tag, or commit' field contains 'MyName/MyRepo'.
- To the right of these is a 'File' dropdown and a large orange 'launch' button.
- Below these fields is a section titled 'Copy the URL below and share your Binder with others:' containing a text input field with placeholder text 'Fill in the fields to see a URL for sharing your Binder.' and a 'copy' icon.
- At the bottom is another section titled 'Copy the text below, then paste into your README to show a binder badge:' containing a text input field with placeholder text 'Copy the text below, then paste into your README to show a binder badge: [ ] launch [ ] binder' and a 'copy' icon.

# Use Binder with the Quantum Katas

To configure Binder automatically to use the Quantum Katas:

1. Open a browser and navigate to <https://aka.ms/try-quantum-katas>.
2. On the **Quantum Katas and Tutorials** landing page, select any of the Katas listed in the learning path to open them as Jupyter Notebooks, allowing you to run and interact with a Kata without requiring any installation.

## Quantum Katas and Tutorials as Jupyter Notebooks

To run the katas and tutorials online, make sure you're viewing this file on Binder (if not, use [this link](#)).

To run the katas and tutorials locally, follow [these installation instructions](#).

While running the Katas online is the easiest option to get started, if you want to save your progress and enjoy better performance, we recommend you to choose the local option.

### Learning path

Here is the learning path we suggest you to follow if you are starting to learn quantum computing and quantum programming. Once you're comfortable with the basics, you're welcome to jump ahead to the topics that pique your interest!

#### Quantum Computing Concepts: Qubits and Gates

- [Complex arithmetic \(tutorial\)](#). Learn about complex numbers and the mathematics required to work with quantum computing.
- [Linear algebra \(tutorial\)](#). Learn about vectors and matrices used to represent quantum states and quantum operations.
- [The qubit \(tutorial\)](#). Learn what a qubit is.
- [Single-qubit gates \(tutorial\)](#). Learn what a quantum gate is and about the most common single-qubit gates.
- [Basic quantum computing gates](#). Learn to apply the most common gates used in quantum computing.
- [Multi-qubit systems \(tutorial\)](#). Learn to represent multi-qubit systems.
- [Multi-qubit gates \(tutorial\)](#). Learn about the most common multi-qubit gates.
- [Superposition](#). Learn to prepare superposition states.

## Next steps

Now that you have set up your Binder environment, you can write and run [your first quantum program](#).

# Update the Quantum Development Kit (QDK) to the latest version

6/14/2021 • 6 minutes to read • [Edit Online](#)

Learn how to update the Quantum Development Kit (QDK) to the latest version.

This article assumes that you already have the QDK installed. If you are installing for the first time, then please refer to the [installation guide](#).

We recommend keeping up to date with the latest QDK release. Follow this update guide to upgrade to the most recent QDK version. The process consists of two parts:

1. Updating your existing Q# files and projects to align your code with any updated syntax.
2. Updating the QDK itself for your chosen development environment.

## Update Q# projects

Regardless of whether you are using C# or Python to host Q# operations, follow these instructions to update your Q# projects.

1. First, check that you have the latest version of the [.NET Core SDK 3.1](#). Run the following command in the command prompt:

```
dotnet --version
```

Verify the output is `3.1.100` or higher. If not, install the [latest version](#) and check again. Then follow the instructions below depending on your setup (Visual Studio, Visual Studio Code, or directly from the command prompt).

### Update Q# projects in Visual Studio

1. Update to the latest version of Visual Studio 2019, see [here](#) for instructions.
2. Open your solution in Visual Studio.
3. From the menu, select **Build -> Clean Solution**.
4. In each of your .csproj files, update the target framework to `netcoreapp3.1` (or `netstandard2.1` if it is a library project). That is, edit lines of the form:

```
<TargetFramework>netcoreapp3.1</TargetFramework>
```

You can find more details on specifying target frameworks [here](#).

5. In each of the .csproj files, set the SDK to `Microsoft.Quantum.Sdk`, as indicated in the line below. Please notice that the version number should be the latest available, and you can determine it by reviewing the [release notes](#).

```
<Project Sdk="Microsoft.Quantum.Sdk/0.17.2105143879">
```

6. Save and close all files in your solution.

7. Select **Tools** -> **Command Line** -> **Developer Command Prompt**. Alternatively, you can use the package management console in Visual Studio.

8. For each project in the solution, run the following command to **remove** this package:

```
dotnet remove [project_name].csproj package Microsoft.Quantum.Development.Kit
```

If your projects use any other Microsoft.Quantum or Microsoft.Azure.Quantum packages (for example, Microsoft.Quantum.Numerics), run the **add** command for these to update the version used.

```
dotnet add [project_name].csproj package [package_name]
```

9. Close the command prompt and select **Build** -> **Build Solution** (do *not* select Rebuild Solution).

You can now skip ahead to [update your Visual Studio QDK extension](#).

### Update Q# projects in Visual Studio Code

1. In Visual Studio Code, open the folder containing the project to update.
2. Select **Terminal** -> **New Terminal**.
3. Follow the instructions for updating using the command prompt (directly below).

### Update Q# projects using the command prompt

1. Navigate to the folder containing your main project file.

2. Run the following command:

```
dotnet clean [project_name].csproj
```

3. Determine the current version of the QDK. To find it, you can review the [release notes](#). The version will be in a format similar to `0.12.20072031`.

4. In each of your `.csproj` files, go through the following steps:

- Update the target framework to `netcoreapp3.1` (or `netstandard2.1` if it is a library project). That is, edit lines of the form:

```
<TargetFramework>netcoreapp3.1</TargetFramework>
```

You can find more details on specifying target frameworks [here](#).

- Replace the reference to the SDK in the project definition. Make sure that the version number corresponds to the value determined in **step 3**.

```
<Project Sdk="Microsoft.Quantum.Sdk/0.17.2105143879">
```

- Remove the reference to package `Microsoft.Quantum.Development.Kit` if present, which will be specified in the following entry:

```
<PackageReference Include="Microsoft.Quantum.Development.Kit" Version="0.17.2105143879" />
```

- Update the version of all the Microsoft Quantum packages to the most recently released version of the QDK (determined in **step 3**). Those packages are named with the following patterns:

```
Microsoft.Quantum.*  
Microsoft.Azure.Quantum.*
```

References to packages have the following format:

```
<PackageReference Include="Microsoft.Quantum.Compiler" Version="0.17.2105143879" />
```

- Save the updated file.
- Restore the dependencies of the project, by doing the following:

```
dotnet restore [project_name].csproj
```

5. Navigate back to the folder containing your main project and run:

```
dotnet build [project_name].csproj
```

With your Q# projects now updated, follow the instructions below to update the QDK itself.

## Update the QDK

The process to update the QDK varies depending on your development language and environment. Select your development environment below.

- [Python: update the `qsharp` package](#)
- [Jupyter Notebooks: update the IQ# kernel](#)
- [Visual Studio: update the QDK extension](#)
- [VS Code: update the QDK extension](#)
- [Command line and C#: update project templates](#)
- [Python: Update the Python SDK for Azure Quantum](#)

## Update the `qsharp` Python package

The update procedure depends on whether you originally installed using conda or using the .NET CLI and pip.

- [Update using conda \(recommended\)](#)
- [Update using .NET CLI and pip \(advanced\)](#)

1. Activate the conda environment where you installed the `qsharp` package, and then run this command to update it:

```
conda update -c quantum-engineering qsharp
```

2. Run the following command from the location of your `.qs` files:

```
python -c "import qsharp; qsharp.reload()"
```

You can now use the updated `qsharp` Python package to run your existing quantum programs.

## Update the IQ# Jupyter kernel

The update procedure depends on whether you originally installed using conda or using the .NET CLI and pip.

- [Update using conda \(recommended\)](#)
- [Update using .NET CLI and pip \(advanced\)](#)

1. Activate the conda environment where you installed the `qsharp` package, and then run this command to update it:

```
conda update -c quantum-engineering qsharp
```

2. Run the following command from a cell in each of your existing Q# Jupyter Notebooks:

```
%workspace reload
```

You can now use the updated IQ# kernel to run your existing Q# Jupyter Notebooks.

## Update the Visual Studio QDK extension

1. Update the Q# Visual Studio extension

- Visual Studio prompts you to accept updates to the [Quantum Visual Studio extension](#)
- Accept the update

### NOTE

The project templates are updated with the extension. The updated templates apply to newly created projects only. The code for your existing projects is not updated when the extension is updated.

## Update the VS Code QDK extension

1. Update the Quantum VS Code extension

- Restart VS Code
- Navigate to the **Extensions** tab
- Select the **Microsoft Quantum Development Kit for Visual Studio Code** extension
- Reload the extension

## C# using the `dotnet` command-line tool

1. Update the Quantum project templates for .NET

From the command prompt:

```
dotnet new -i Microsoft.Quantum.ProjectTemplates
```

Alternatively, if you intend to use the command-line templates, and already have the VS Code QDK extension installed, you can update the project templates from the extension itself:

- [Update the QDK extension](#)
- In VS Code, go to **View -> Command Palette**

- Select Q#: **Install command line project templates**
- After a few seconds you should get a popup confirming "project templates installed successfully"

## Update the Python SDK for Azure Quantum

1. Update to the latest `azure-quantum` Python package by using the Python Package Installer (PIP)

```
pip install --upgrade azure-quantum
```

2. If you encounter any issues please ensure that Python and PIP are up to date. For information on the latest version requirements, please follow our guide for [installing and using the Python SDK](#)

# Tutorial: Implement a Quantum Random Number Generator in Q#

4/6/2021 • 5 minutes to read • [Edit Online](#)

A simple example of a quantum algorithm written in Q# is a quantum random number generator. This algorithm leverages the nature of quantum mechanics to produce a random number.

## Prerequisites

- The Microsoft [Quantum Development Kit](#).
- Create a Q# project for either a [Q# application](#), with a [Python host program](#), or a [C# host program](#).

## Write a Q# operation

### Q# operation code

1. Replace the contents of the Program.qs file with the following code:

```
namespace Qrng {
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    @EntryPoint()
    operation SampleQuantumRandomNumberGenerator() : Result {
        use q = Qubit(); // Allocate a qubit.
        H(q);           // Put the qubit to superposition. It now has a 50% chance of being 0 or 1.
        return MResetZ(q); // Measure the qubit value.
    }
}
```

As mentioned in our [Understanding quantum computing](#) article, a qubit is a unit of quantum information that can be in superposition. When measured, a qubit can only be either 0 or 1. However, before measurement, the state of the qubit represents the probability of reading either a 0 or a 1 with a measurement. This probabilistic state is known as superposition. We can use this probability to generate random numbers.

In our Q# operation, we introduce the `Qubit` datatype, native to Q#. We can only allocate a `Qubit` with a `use` statement. When it gets allocated, a qubit is always in the `Zero` state.

Using the `H` operation, we are able to put our `Qubit` in superposition. To measure a qubit and read its value, you use the `M` intrinsic operation.

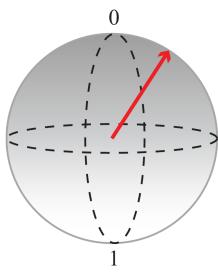
By putting our `Qubit` in superposition and measuring it, our result will be a different value each time the code is invoked.

When a `Qubit` is deallocated it must be explicitly set back to the `Zero` state, otherwise the simulator will report a runtime error. An easy way to achieve this is invoking `Reset`.

### Visualizing the code with the Bloch sphere

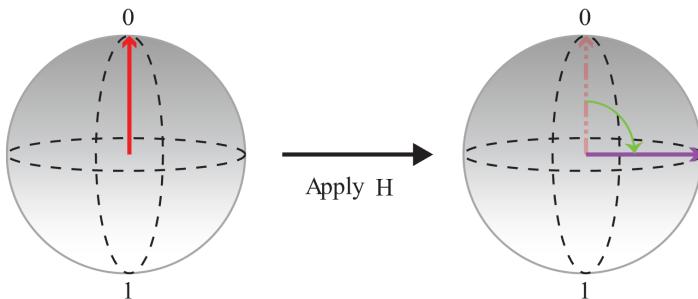
In the Bloch sphere, the north pole represents the classical value 0 and the south pole represents the classical value 1. Any superposition can be represented by a point on the sphere (represented by an arrow). The closer the end of the arrow to a pole the higher the probability the qubit collapses into the classical value assigned to

that pole when measured. For example, the qubit state represented by the red arrow below has a higher probability of giving the value 0 if we measure it.

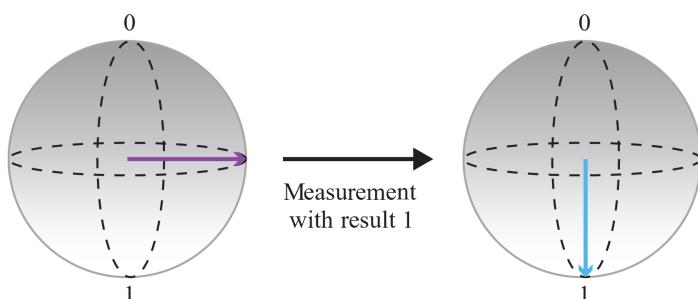


We can use this representation to visualize what the code is doing:

- First we start with a qubit initialized in the state 0 and apply  $\text{H}$  to create a superposition in which the probabilities for 0 and 1 are the same.



- Then we measure the qubit and save the output:



Since the outcome of the measurement is completely random, we have obtained a random bit. We can call this operation several times to create integers. For example, if we call the operation three times to obtain three random bits, we can build random 3-bit numbers (that is, a random number between 0 and 7).

## Creating a complete random number generator

Now that we have a Q# operation that generates random bits, we can use it to build a complete quantum random number generator. We can use a Q# application or use a host program.

- [Q# applications with Visual Studio or Visual Studio Code](#)
- [Python with Visual Studio Code or the command prompt](#)
- [C# with Visual Studio Code or Visual Studio](#)

To create the full Q# application, add the following entry point to your Q# program:

```

namespace Qrng {
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    operation SampleQuantumRandomNumberGenerator() : Result {
        use q = Qubit(); // Allocate a qubit.
        H(q); // Put the qubit to superposition. It now has a 50% chance of being 0 or 1.
        return MResetZ(q); // Measure the qubit value.
    }

    operation SampleRandomNumberInRange(max : Int) : Int {
        mutable bits = new Result[0];
        for idxBit in 1..BitSizeI(max) {
            set bits += [SampleQuantumRandomNumberGenerator()];
        }
        let sample = ResultArrayAsInt(bits);
        return sample > max
            ? SampleRandomNumberInRange(max)
            | sample;
    }
    @EntryPoint()
    operation SampleRandomNumber() : Int {
        let max = 50;
        Message($"Sampling a random number between 0 and {max}: ");
        return SampleRandomNumberInRange(max);
    }
}

```

The program will run the operation or function marked with the `@EntryPoint()` attribute on a simulator or resource estimator, depending on the project configuration and command-line options.

In Visual Studio, simply press Ctrl + F5 to run the script.

In VS Code, build the Program.qs the first time by typing the below in the terminal:

```
dotnet build
```

For subsequent runs, there is no need to build it again. To run it, type the following command and press enter:

```
dotnet run --no-build
```

# Tutorial: Explore entanglement with Q#

5/27/2021 • 13 minutes to read • [Edit Online](#)

In this tutorial, we show you how to write a Q# program that manipulates and measures qubits and demonstrates the effects of superposition and entanglement.

You will write an application called Bell to demonstrate quantum entanglement. The name Bell is in reference to Bell states, which are specific quantum states of two qubits that are used to represent the simplest examples of superposition and quantum entanglement.

## Pre-requisites

If you are ready to start coding, follow these steps before proceeding:

- [Install](#) the Quantum Development Kit (QDK) using your preferred language and development environment.
- If you already have the QDK installed, make sure you have [updated](#) to the latest version

You can also follow along with the narrative without installing the QDK, reviewing the overviews of the Q# programming language and the first concepts of quantum computing.

## In this tutorial, you'll learn how to:

- Create and combine operations in Q#
- Create operations to put qubits in superposition, entangle and measure them.
- Demonstrate quantum entanglement with a Q# program run in a simulator.

## Demonstrating qubit behavior with the QDK

Where classical bits hold a single binary value such as a 0 or 1, the state of a [qubit](#) can be in a [superposition](#) of 0 and 1. Conceptually, the state of a qubit can be thought of as a direction in an abstract space (also known as a vector). A qubit state can be in any of the possible directions. The two [classical states](#) are the two directions; representing 100% chance of measuring 0 and 100% chance of measuring 1.

The act of measurement produces a binary result and changes a qubit state. Measurement produces a binary value, either 0 or 1. The qubit goes from being in superposition (any direction) to one of the classical states. Thereafter, repeating the same measurement without any intervening operations produces the same binary result.

Multiple qubits can be [entangled](#). When we make a measurement of one entangled qubit, our knowledge of the state of the other(s) is updated as well.

Now, we're ready to demonstrate how Q# expresses this behavior. You start with the simplest program possible and build it up to demonstrate quantum superposition and quantum entanglement.

## Creating a Q# project

The first thing we have to do is to create a new Q# project. In this tutorial we are going to use the environment based on [Q# applications with VS Code](#).

To create a new project, in VS Code:

1. Click **View -> Command Palette** and select **Q#: Create New Project**.

2. Click **Standalone console application**.
3. Navigate to the location to save the project and click **Create Project**.
4. When the project is successfully created, click **Open new project...** in the lower right.

In this case we called the project `Bell`. This generates two files: `Bell.csproj`, the project file and `Program.qs`, a template of a Q# application that we will use to write our application. The content of `Program.qs` should be:

```
namespace Bell {

    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;

    @EntryPoint()
    operation HelloQ() : Unit {
        Message("Hello quantum world!");
    }
}
```

## Write the Q# application

Our goal is to prepare two qubits in a specific quantum state, demonstrating how to operate on qubits with Q# to change their state and demonstrate the effects of superposition and entanglement. We will build this up piece by piece to introduce qubit states, operations, and measurement.

### Initialize qubit using measurement

In the first code snippet below, we show you how to work with qubits in Q#. We'll introduce two operations, `M` and `X` that transform the state of a qubit. In this code snippet, an operation `SetQubitState` is defined that takes as a parameter a qubit and another parameter, `desired`, representing the state that we would like the qubit to be in. The operation `SetQubitState` performs a measurement on the qubit using the operation `M`. In Q#, a qubit measurement always returns either `Zero` or `One`. If the measurement returns a value not equal to the desired value, `SetQubitState` "flips" the qubit; that is, it runs an `X` operation, which changes the qubit state to a new state in which the probabilities of a measurement returning `Zero` and `One` are reversed. This way, `SetQubitState` always puts the target qubit in the desired state.

Replace the contents of `Program.qs` with the following code:

```
namespace Bell {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Canon;

    operation SetQubitState(desired : Result, q1 : Qubit) : Unit {
        if desired != M(q1) {
            X(q1);
        }
    }
}
```

This operation may now be called to set a qubit to a classical state, either returning `Zero` 100% of the time or returning `One` 100% of the time. `Zero` and `One` are constants that represent the only two possible results of a measurement of a qubit.

The operation `SetQubitState` measures the qubit. If the qubit is in the state we want, `SetQubitState` leaves it alone; otherwise, by running the `X` operation, we change the qubit state to the desired state.

### About Q# operations

A Q# operation is a quantum subroutine. That is, it is a callable routine that contains calls to other quantum

operations.

The arguments to an operation are specified as a tuple, within parentheses.

The return type of the operation is specified after a colon. In this case, the `SetQubitState` operation has no return type, so it is marked as returning `Unit`. This is the Q# equivalent of `unit` in F#, which is roughly analogous to `void` in C#, and an empty tuple in Python (`()`, represented by the type hint `Tuple[()]`).

You have used two quantum operations in your first Q# operation:

- The `M` operation, which measures the state of the qubit
- The `X` operation, which flips the state of a qubit

A quantum operation transforms the state of a qubit. Sometime people talk about quantum gates instead of operations, in analogy to classical logic gates. This is rooted in the early days of quantum computing when algorithms were merely a theoretical construct and visualized as diagrams similarly to circuit diagrams in classical computing.

### Counting measurement outcomes

To demonstrate the effect of the `SetQubitState` operation, a `TestBellState` operation is then added. This operation takes as input a `Zero` or `One`, and calls the `SetQubitState` operation some number of times with that input, and counts the number of times that `Zero` was returned from the measurement of the qubit and the number of times that `One` was returned. Of course, in this first simulation of the `TestBellState` operation, we expect that the output will show that all measurements of the qubit set with `Zero` as the parameter input will return `Zero`, and all measurements of a qubit set with `One` as the parameter input will return `One`. Further on, we'll add code to `TestBellState` to demonstrate superposition and entanglement.

Add the following operation to the `Program.qs` file, inside the namespace, after the end of the `SetQubitState` operation:

```
operation TestBellState(count : Int, initial : Result) : (Int, Int) {

    mutable numOnes = 0;
    use qubit = Qubit();
    for test in 1..count {
        SetQubitState(initial, qubit);
        let res = M(qubit);

        // Count the number of ones we saw:
        if res == One {
            set numOnes += 1;
        }
    }

    SetQubitState(Zero, qubit);

    // Return number of times we saw a |0> and number of times we saw a |1>
    Message("Test results (# of 0s, # of 1s): ");
    return (count - numOnes, numOnes);
}
```

Note that we added a line before the `return` to print an explanatory message in the console with the function (`Message`)`[microsoft.quantum.intrinsic.message]`

This operation (`TestBellState`) will loop for `count` iterations, set a specified `initial` value on a qubit and then measure (`M`) the result. It will gather statistics on how many zeros and ones we've measured and return them to the caller. It performs one other necessary operation. It resets the qubit to a known state (`Zero`) before

returning it allowing others to allocate this qubit in a known state. This is required by the `use` statement.

#### About variables in Q#

By default, variables in Q# are immutable; their value may not be changed after they are bound. The `let` keyword is used to indicate the binding of an immutable variable. Operation arguments are always immutable.

If you need a variable whose value can change, such as `numOnes` in the example, you can declare the variable with the `mutable` keyword. A mutable variable's value may be changed using a `set` statement.

In both cases, the type of a variable is inferred by the compiler. Q# doesn't require any type annotations for variables.

#### About `use` statements in Q#

The `use` statement is also special to Q#. It is used to allocate qubits for use in a block of code. In Q#, all qubits are dynamically allocated and released, rather than being fixed resources that are there for the entire lifetime of a complex algorithm. A `use` statement allocates a set of qubits at the start, and releases those qubits at the end of the block.

## Run the code from the command prompt

In order to run the code we need to tell the compiler *which* callable to run when we provide the `dotnet run` command. This is done with a simple change in the Q# file by adding a line with `@EntryPoint()` directly preceding the callable: the `TestBellState` operation in this case. The full code should be:

```
namespace Bell {
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;

    operation SetQubitState(desired : Result, target : Qubit) : Unit {
        if desired != M(target) {
            X(target);
        }
    }

    @EntryPoint()
    operation TestBellState(count : Int, initial : Result) : (Int, Int) {

        mutable numOnes = 0;
        use qubit = Qubit();
        for test in 1..count {
            SetQubitState(initial, qubit);
            let res = M(qubit);

            // Count the number of ones we saw:
            if res == One {
                set numOnes += 1;
            }
        }

        SetQubitState(Zero, qubit);

        // Return number of times we saw a |0> and number of times we saw a |1>
        Message("Test results (# of 0s, # of 1s): ");
        return (count - numOnes, numOnes);
    }
}
```

To run the program we need to specify `count` and `initial` arguments from the command prompt. Let's choose for example `count = 1000` and `initial = One`. Enter the following command:

```
dotnet run --count 1000 --initial One
```

And you should observe the following output:

```
Test results (# of 0s, # of 1s):  
(0, 1000)
```

If you try with `initial = Zero` you should observe:

```
dotnet run --count 1000 --initial Zero
```

```
Test results (# of 0s, # of 1s):  
(1000, 0)
```

## Prepare superposition

Now let's look at how Q# expresses ways to put qubits in superposition. Recall that the state of a qubit can be in a superposition of 0 and 1. We'll use the `Hadamard` operation to accomplish this. If the qubit is in either of the classical states (where a measurement returns `Zero` always or `One` always), then the `Hadamard` or `H` operation will put the qubit in a state where a measurement of the qubit will return `Zero` 50% of the time and return `One` 50% of the time. Conceptually, the qubit can be thought of as halfway between the `Zero` and `One`. Now, when we simulate the `TestBellState` operation, we will see the results will return roughly an equal number of `zero` and `one` after measurement.

### `X` flips qubit state

First we'll just try to flip the qubit (if the qubit is in `Zero` state it will flip to `One` and vice versa). This is accomplished by performing an `X` operation before we measure it in `TestBellState`:

```
X(qubit);  
let res = M(qubit);
```

Now the results are reversed:

```
dotnet run --count 1000 --initial One
```

```
Test results (# of 0s, # of 1s):  
(1000, 0)
```

```
dotnet run --count 1000 --initial Zero
```

```
Test results (# of 0s, # of 1s):  
(0, 1000)
```

Now let's explore the quantum properties of the qubits.

### `H` prepares superposition

All we need to do is replace the `X` operation in the previous run with an `H` or Hadamard operation. Instead of

flipping the qubit all the way from 0 to 1, we will only flip it halfway. The replaced lines in `TestBellState` now look like:

```
H(qubit);
let res = M(qubit);
```

Now the results get more interesting:

```
dotnet run --count 1000 --initial One
```

```
Test results (# of 0s, # of 1s):
(496, 504)
```

```
dotnet run --count 1000 --initial Zero
```

```
Test results (# of 0s, # of 1s):
(506, 494)
```

Every time we measure, we ask for a classical value, but the qubit is halfway between 0 and 1, so we get (statistically) 0 half the time and 1 half the time. This is known as **superposition** and gives us our first real view into a quantum state.

## Prepare entanglement

Now let's look at how Q# expresses ways to entangle qubits. First, we set the first qubit to the initial state and then use the `H` operation to put it into superposition. Then, before we measure the first qubit, we use a new operation (`CNOT`), which stands for *Controlled-NOT*. The result of running this operation on two qubits is to flip the second qubit if the first qubit is `One`. Now, the two qubits are entangled. Our statistics for the first qubit haven't changed (50-50 chance of a `Zero` or a `One` after measurement), but now when we measure the second qubit, it is **always** the same as what we measured for the first qubit. Our `CNOT` has entangled the two qubits, so that whatever happens to one of them, happens to the other. If you reversed the measurements (did the second qubit before the first), the same thing would happen. The first measurement would be random and the second would be in lock step with whatever was discovered for the first.

The first thing we'll need to do is allocate two qubits instead of one in `TestBellState`:

```
use (q0, q1) = (Qubit(), Qubit());
```

This will allow us to add a new operation (`CNOT`) before we measure (`M`) in `TestBellState`:

```
SetQubitState(initial, q0);
SetQubitState(Zero, q1);

H(q0);
CNOT(q0, q1);
let res = M(q0);
```

We've added another `SetQubitState` operation to initialize the first qubit to make sure that it's always in the `Zero` state when we start.

We also need to reset the second qubit before releasing it.

```
SetQubitState(Zero, q0);
SetQubitState(Zero, q1);
```

The full routine now looks like this:

```
operation TestBellState(count : Int, initial : Result) : (Int, Int) {

    mutable numOnes = 0;
    use (q0, q1) = (Qubit(), Qubit());
    for test in 1..count {
        SetQubitState(initial, q0);
        SetQubitState(Zero, q1);

        H(q0);
        CNOT(q0,q1);
        let res = M(q0);

        // Count the number of ones we saw:
        if res == One {
            set numOnes += 1;
        }
    }

    SetQubitState(Zero, q0);
    SetQubitState(Zero, q1);

    // Return number of times we saw a |0> and number of times we saw a |1>
    return (count-numOnes, numOnes);
}
```

If we run this, we'll get exactly the same 50-50 result we got before. However, what we're interested in is how the second qubit reacts to the first being measured. We'll add this statistic with a new version of the `TestBellState` operation:

```

operation TestBellState(count : Int, initial : Result) : (Int, Int, Int) {
    mutable numOnes = 0;
    mutable agree = 0;
    use (q0, q1) = (Qubit(), Qubit());
    for test in 1..count {
        SetQubitState(initial, q0);
        SetQubitState(Zero, q1);

        H(q0);
        CNOT(q0, q1);
        let res = M(q0);

        if M(q1) == res {
            set agree += 1;
        }

        // Count the number of ones we saw:
        if res == One {
            set numOnes += 1;
        }
    }

    SetQubitState(Zero, q0);
    SetQubitState(Zero, q1);

    // Return times we saw |0>, times we saw |1>, and times measurements agreed
    Message("Test results (# of 0s, # of 1s, # of agreements)");
    return (count-numOnes, numOnes, agree);
}

```

The new return value (`agree`) keeps track of every time the measurement from the first qubit matches the measurement of the second qubit.

Running the code we obtain:

```
dotnet run --count 1000 --initial One
```

```
(505, 495, 1000)
```

```
dotnet run --count 1000 --initial Zero
```

```
Test results (# of 0s, # of 1s, # of agreements)
(507, 493, 1000)
```

As stated in the overview, our statistics for the first qubit haven't changed (50-50 chance of a 0 or a 1), but now when we measure the second qubit, it is **always** the same as what we measured for the first qubit, because they are entangled!

## Next steps

The tutorial [Grover's search](#) shows you how to build and run Grover search, one of the most popular quantum computing algorithms and offers a nice example of a Q# program that can be used to solve real problems with quantum computing.

[Get Started with the Quantum Development Kit](#) recommends more ways to learn Q# and quantum

programming.

# Tutorial: Implement Grover's search algorithm in Q#

5/27/2021 • 16 minutes to read • [Edit Online](#)

In this tutorial you'll learn to implement Grover's algorithm in Q# to solve search based problems.

Grover's algorithm is one of the most famous algorithms in quantum computing. The problem it solves is often referred to as "searching a database", but it's more accurate to think of it in terms of the *search problem*.

Any search task can be mathematically formulated with an abstract function  $f(x)$  that accepts search items  $x$ . If the item  $x$  is a solution to the search task, then  $f(x)=1$ . If the item  $x$  isn't a solution, then  $f(x)=0$ . The search problem consists of finding any item  $x_0$  such that  $f(x_0)=1$ .

## NOTE

This tutorial is intended for people who are already familiar with Grover's algorithm that want to learn how to implement it in Q#. For a more slow paced tutorial we recommend the Microsoft Learn module [Solve graph coloring problems by using Grover's search](#). For a detailed explanation on the theory behind Grover's algorithm, check the conceptual article [Theory of Grover's algorithm](#).

## Grover's algorithm task

Given a classical function  $f(x):\{0,1\}^n \rightarrow \{0,1\}$ , where  $n$  is the bit-size of the search space, find an input  $x_0$  for which  $f(x_0)=1$ .

To implement Grover's algorithm to solve a problem you need to:

1. **Transform the problem to the form of a Grover's task:** for example, suppose we want to find the factors of an integer  $M$  using Grover's algorithm. You can transform the integer factorization problem to a Grover's task by creating a function  $f_M(x)=1[r]$  where  $f_M(x)=1$  if  $r=0$  and  $f_M(x)=0$  if  $r \neq 0$  and  $r$  is the remainder of  $M/x$ . This way, the integers  $x_i$  that make  $f_M(x_i)=1$  are the factors of  $M$  and we transformed the problem to a Grover's task.
2. **Implement the function of the Grover's task as a quantum oracle:** to implement Grover's algorithm, you need to implement the function  $f(x)$  of your Grover's task as a [quantum oracle](#).
3. **Use Grover's algorithm with your oracle to solve the task:** once you have a quantum oracle, you can plug it into your Grover's algorithm implementation to solve the problem and interpret the output.

## Quick overview of Grover's algorithm

Suppose we have  $N=2^n$  eligible items for the search task and we index them by assigning each item an integer from  $0$  to  $N-1$ . The steps of the algorithm are:

1. Start with a register of  $n$  qubits initialized in the state  $\ket{0}$ .
2. Prepare the register into a uniform superposition by applying  $H$  to each qubit in the register:  
$$\frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} \ket{x}$$
3. Apply the following operations to the register  $N_{\text{optimal}}$  times:
  - a. The phase oracle  $O_f$  that applies a conditional phase shift of  $-1$  for the solution items.
  - b. Apply  $H$  to each qubit in the register.
  - c. Apply  $-O_0$ , a conditional phase shift of  $-1$  to every computational basis state except  $\ket{0}$ .
  - d. Apply  $H$  to each qubit in the register.
4. Measure the register to obtain the index of an item that's a solution with very high probability.

5. Check if it's a valid solution. If not, start again.

## Write the code for Grover's algorithm

Now let's see how to implement the algorithm in Q#.

### Grover's diffusion operator

First, we are going to write an operation that applies the steps **b**, **c** and **d** from the loop above. Together, these steps are also known as the Grover diffusion operator  $\text{H}^{\otimes n} \text{O}_0 \text{H}^{\otimes n}$

```
operation ReflectAboutUniform(inputQubits : Qubit[]) : Unit {  
  
    within {  
        ApplyToEachA(H, inputQubits);  
        ApplyToEachA(X, inputQubits);  
    } apply {  
        Controlled Z(Most(inputQubits), Tail(inputQubits));  
    }  
  
}
```

In this operation we use the [within-apply](#) statement that implements the automatic conjugation of operations that occur in Grover's diffusion operator.

#### NOTE

To learn more about conjugations in Q#, check the [conjugations article in the Q# language guide](#).

A good exercise to understand the code and the operations is to check with pen and paper that the operation `ReflectAboutUniform` applies Grover's diffusion operator. To see it note that the operation `Controlled Z(Most(inputQubits), Tail(inputQubits))` only has an effect different than the identity if and only if all qubits are in the state  $|\text{ket}{1}\rangle$ .

You can check what each of the operations and functions used is by looking into the API documentation:

- [ApplyToEachA](#)
- [Most](#)
- [Tail](#)

The operation is called `ReflectAboutUniform` because it can be geometrically interpreted as a reflection in the vector space about the uniform superposition state.

### Number of iterations

Grover's search has an optimal number of iterations that yields the highest probability of measuring a valid output. If the problem has  $N=2^n$  possible eligible items, and  $M$  of them are solutions to the problem, the optimal number of iterations is:

$$N_{\text{optimal}} \approx \frac{\pi}{4} \sqrt{\frac{N}{M}}$$

Continuing to iterate past that number starts reducing that probability until we reach nearly-zero success probability on iteration  $N_{\text{optimal}}$ . After that, the probability grows again and until  $N_{\text{optimal}}$  and so on.

In practical applications, you don't usually know how many solutions your problem has before you solve it. An efficient strategy to handle this issue is to "guess" the number of solutions  $M$  by progressively increasing the guess in powers of two (i.e.  $1, 2, 4, 8, 16, \dots, 2^n$ ). One of these guesses will be sufficiently close that the

algorithm will still find the solution with an average number of iterations around  $\sqrt{\frac{N}{M}}$ .

### Complete Grover's operation

Now we are ready to write a Q# operation for Grover's search algorithm. It will have three inputs:

- A qubit array `register : Qubit[]` that should be initialized in the all `Zero` state. This register will encode the tentative solution to the search problem. After the operation it will be measured.
- An operation `phaseOracle : (Qubit[]) => Unit` is `Adj` that represents the phase oracle for the Grover's task. This operation applies an unitary transformation over a generic qubit register.
- An integer `iterations : Int` to represent the iterations of the algorithm.

```
operation RunGroversSearch(register : Qubit[], phaseOracle : (Qubit[]) => Unit is Adj, iterations : Int) : Unit {
    // Prepare register into uniform superposition.
    ApplyToEach(H, register);
    // Start Grover's loop.
    for _ in 1 .. iterations {
        // Apply phase oracle for the task.
        phaseOracle(register);
        // Apply Grover's diffusion operator.
        ReflectAboutUniform(register);
    }
}
```

This code is generic - it can be used to solve any search problem. We pass the quantum oracle - the only operation that relies on the knowledge of the problem instance we want to solve - as a parameter to the search code.

## Implement the oracle

One of the key properties that makes Grover's algorithm faster is the ability of quantum computers to perform calculations not only on individual inputs but also on superpositions of inputs. We need to compute the function  $f(x)$  that describes the instance of a search problem using only quantum operations. This way we can compute it over a superposition of inputs.

Unfortunately there isn't an automatic way to translate classical functions to quantum operations. It's an open field of research in computer science called *reversible computing*.

However, there are some guidelines that might help you to translate your function  $f(x)$  into a quantum oracle:

1. **Break down the classical function into small building blocks that are easy to implement.** For example, you can try to decompose your function  $f(x)$  into a series of arithmetic operations or Boolean logic gates.
2. **Use the higher-level building blocks of the Q# library to implement the intermediate operations.** For instance, if you decomposed your function into a combination of simple arithmetic operations, you can use the [Numerics library](#) to implement the intermediate operations.

The following equivalence table might prove useful when implementing Boolean functions in Q#.

CLASSICAL LOGIC GATE	Q# OPERATION
\$NOT\$	X
\$XOR\$	CNOT
\$AND\$	CCNOT with an auxiliary qubit

## Example: Quantum operation to check if a number is a divisor

### IMPORTANT

In this tutorial we are going to factorize a number using Grover's search algorithm as a didactic example to show how to translate a simple mathematical problem into a Grover's task. However, **Grover's algorithm is NOT an efficient algorithm to solve the integer factorization problem**. To explore a quantum algorithm that does solve the integer factorization problem faster than any classical algorithm, check the [Shor's algorithm sample](#).

As an example, let's see how we would express the function  $f_M(x) = 1[r]$  of the factoring problem as a quantum operation in Q#.

Classically, we would compute the remainder of the division  $M/x$  and check if it's equal to zero. If it is, the program outputs `1`, and if it's not, the program outputs `0`. We need to:

- Compute the remainder of the division.
- Apply a controlled operation over the output bit so that it's `1` if the remainder is `0`.

So we need to calculate a division of two numbers with a quantum operation. Fortunately, you don't need to write the circuit implementing the division from scratch, you can use the `DivideI` operation from the Numerics library instead.

If we look into the description of `DivideI`, we see that it needs three qubit registers: the  $n$ -bit dividend `xs`, the  $n$ -bit divisor `ys`, and the  $n$ -bit `result` that must be initialized in the state `Zero`. The operation is `Adj + Ctl`, so we can conjugate it and use it in *within-apply* statements. Also, in the description it says that the dividend in the input register `xs` is replaced by the remainder. This is perfect since we are interested exclusively in the remainder, and not in the result of the operation.

We can then build a quantum operation that does the following:

1. Takes three inputs:
  - The dividend, `number : Int`. This is the  $M$  in  $f_M(x)$ .
  - A qubit array encoding the divisor, `divisorRegister : Qubit[]`. This is the  $x$  in  $f_M(x)$ , possibly in a superposition state.
  - A target qubit, `target : Qubit`, that flips if the output of  $f_M(x)$  is `1`.
2. Calculates the division  $M/x$  using only reversible quantum operations, and flips the state of `target` if and only if the remainder is zero.
3. Reverts all operations except the flipping of `target`, so as to return the used auxiliary qubits to the zero state without introducing irreversible operations, such as measurement. This step is important in order to preserve entanglement and superposition during the process.

The code to implement this quantum operation is:

```

operation MarkDivisor (
    dividend : Int,
    divisorRegister : Qubit[],
    target : Qubit
) : Unit is Adj + Ctl {
    // Calculate the bit-size of the dividend.
    let size = BitSizeI(dividend);
    // Allocate two new qubit registers for the dividend and the result.
    use dividendQubits = Qubit[size];
    use resultQubits = Qubit[size];
    // Create new LittleEndian instances from the registers to use DivideI
    let xs = LittleEndian(dividendQubits);
    let ys = LittleEndian(divisorRegister);
    let result = LittleEndian(resultQubits);

    // Start a within-apply statement to perform the operation.
    within {
        // Encode the dividend in the register.
        ApplyXorInPlace(dividend, xs);
        // Apply the division operation.
        DivideI(xs, ys, result);
        // Flip all the qubits from the remainder.
        ApplyToEachA(X, xs!);
    } apply {
        // Apply a controlled NOT over the flipped remainder.
        Controlled X(xs!, target);
        // The target flips if and only if the remainder is 0.
    }
}

```

#### NOTE

We take advantage of the statement *within-apply* to achieve step 3. Alternatively, we could explicitly write the adjoints of each of the operations inside the `within` block after the controlled flipping of `target`. The *within-apply* statement does it for us, making the code shorter and more readable. One of the main goals of Q# is to make quantum programs easy to write and read.

#### Transform the operation into a phase oracle

The operation `MarkDivisor` is what's known as a *marking oracle*, since it marks the valid items with an entangled auxiliary qubit (`target`). However, Grover's algorithm needs a *phase oracle*, that is, an oracle that applies a conditional phase shift of  $-1\$$  for the solution items. But don't panic, the operation above wasn't written in vain. It's very easy to switch from one oracle type to the other in Q#.

We can apply any marking oracle as a phase oracle with the following operation:

```

operation ApplyMarkingOracleAsPhaseOracle(
    markingOracle : (Qubit[], Qubit) => Unit is Adj,
    register : Qubit[]
) : Unit is Adj {
    use target = Qubit();
    within {
        X(target);
        H(target);
    } apply {
        markingOracle(register, target);
    }
}

```

This famous transformation is often known as the *phase kickback* and it's widely used in many quantum computing algorithms. You can find a detailed explanation of this technique in this [Microsoft Learn module](#).

# Factoring numbers with Grover's search

Now we have all the ingredients to implement a particular instance of Grover's search algorithm and solve our factoring problem.

Let's use the program below to find a factor of 21. To simplify the code, let's assume that we know the number \$M\$ of valid items. In this case, \$M=4\$, since there are two factors, 3 and 7, plus 1 and 21 itself.

```
namespace GroversTutorial {
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Arithmetic;
    open Microsoft.Quantum.Arrays;
    open Microsoft.Quantum.Preparation;

    @EntryPoint()
    operation FactorizeWithGrovers(number : Int) : Unit {

        // Define the oracle that for the factoring problem.
        let markingOracle = MarkDivisor(number, _, _);
        let phaseOracle = ApplyMarkingOracleAsPhaseOracle(markingOracle, _);
        // Bit-size of the number to factorize.
        let size = BitSizeI(number);
        // Estimate of the number of solutions.
        let nSolutions = 4;
        // The number of iterations can be computed using the formula.
        let nIterations = Round(PI() / 4.0 * Sqrt(IntAsDouble(size) / IntAsDouble(nSolutions)));

        // Initialize the register to run the algorithm
        use (register, output) = (Qubit[size], Qubit());
        mutable isCorrect = false;
        mutable answer = 0;
        // Use a Repeat-Until-Succeed loop to iterate until the solution is valid.
        repeat {
            RunGroversSearch(register, phaseOracle, nIterations);
            let res = MultiM(register);
            set answer = BoolArrayToInt(ResultArrayAsBoolArray(res));
            // Check that if the result is a solution with the oracle.
            markingOracle(register, output);
            if MResetZ(output) == One and answer != 1 and answer != number {
                set isCorrect = true;
            }
            ResetAll(register);
        } until isCorrect;

        // Print out the answer.
        Message($"The number {answer} is a factor of {number}.");
    }

    operation MarkDivisor (
        dividend : Int,
        divisorRegister : Qubit[],
        target : Qubit
    ) : Unit is Adj+Ctl {
        let size = BitSizeI(dividend);
        use (dividendQubits, resultQubits) = (Qubit[size], Qubit[size]);
        let xs = LittleEndian(dividendQubits);
        let ys = LittleEndian(divisorRegister);
        let result = LittleEndian(resultQubits);
        within{
            ApplyXorInPlace(dividend, xs);
            DivideI(xs, ys, result);
            AnnInvToEachA(X, xs!);
        }
    }
}
```

```

        ApplyToEach(X, xs!),
    }
    apply{
        Controlled X(xs!, target);
    }
}

operation PrepareUniformSuperpositionOverDigits(digitReg : Qubit[]) : Unit is Adj + Ctl {
    PrepareArbitraryStateCP(ConstantArray(10, ComplexPolar(1.0, 0.0)), LittleEndian(digitReg));
}

operation ApplyMarkingOracleAsPhaseOracle(
    markingOracle : (Qubit[], Qubit) => Unit is Adj,
    register : Qubit[]
) : Unit is Adj {
    use target = Qubit();
    within {
        X(target);
        H(target);
    } apply {
        markingOracle(register, target);
    }
}

operation RunGroversSearch(register : Qubit[], phaseOracle : ((Qubit[]) => Unit is Adj), iterations : Int) : Unit {
    ApplyToEach(H, register);
    for _ in 1 .. iterations {
        phaseOracle(register);
        ReflectAboutUniform(register);
    }
}

operation ReflectAboutUniform(inputQubits : Qubit[]) : Unit {
    within {
        ApplyToEachA(H, inputQubits);
        ApplyToEachA(X, inputQubits);
    } apply {
        Controlled Z(Most(inputQubits), Tail(inputQubits));
    }
}
}

```

### IMPORTANT

In order to be able to use operations from the numerics library (or any other library besides the standard library), we need to make sure the corresponding package has been [added to our project](#). For a quick way to do so in VS Code, open the terminal from within your project and run the following command:

```
dotnet add package Microsoft.Quantum.Numerics
```

### Run it with Visual Studio or Visual Studio Code

The program above will run the operation or function marked with the `@EntryPoint()` attribute on a simulator or resource estimator, depending on the project configuration and command-line options.

- [Visual Studio](#)
- [VS Code](#)

In general, running a Q# program in Visual Studio is as simple as pressing `ctrl + F5`. But first, we need to provide the right command-line arguments to our program.

Command-line arguments can be configured via the debug page of your project properties. You can visit the [Visual Studio reference guide](#) for more information about this, or follow the steps below:

1. In the solution explorer on the right, right-click the name of your project (the project node, one level below the solution) and select **Properties**.
2. From the new window that opens, navigate to the **Debug** tab.
3. In the field **Application arguments**, you can enter any arguments you wish to pass to the entry point of your program. Enter `--number 21` in the arguments field.

Now press `Ctrl + F5` to run the program.

With either environment, you should now see the following message displayed in the terminal:

```
The number 7 is a factor of 21.
```

## Extra: check the statistics with Python

How can you check that the algorithm is behaving correctly? For example, if we substituted Grover's search by a random number generator in the code above, after  $\sim \$N\$$  attempts it will also find a factor.

Let's write a small Python script to check that the program is working as it should.

### TIP

If you need help running Q# applications within Python, you can take a look at our guide about the [ways to run a Q# program](#) and the [installation guide for Python](#).

First, we are going to modify our main operation to get rid of the repeat-until-success loop, instead outputting the first measurement result after running Grover's search:

```
@EntryPoint()
operation FactorizeWithGrovers2(number : Int) : Int {

    let markingOracle = MarkDivisor(number, _, _);
    let phaseOracle = ApplyMarkingOracleAsPhaseOracle(markingOracle, _);
    let size = BitSizeI(number);
    let nSolutions = 4;
    let nIterations = Round(PI() / 4.0 * Sqrt(IntAsDouble(size) / IntAsDouble(nSolutions)));

    use register = Qubit[size] {
        RunGroversSearch(register, phaseOracle, nIterations);
        let res = MultiM(register);
        return ResultArrayAsInt(res);
        // Check whether the result is correct.
    }

}
```

Note that we changed the output type from `Unit` to `Int`. This will be useful for the Python program.

The Python program is very simple. It just calls the operation `FactorizeWithGrovers2` several times and plots the results in a histogram.

The code is the following:

```

import qsharp
qsharp.packages.add("Microsoft.Quantum.Numerics")
qsharp.reload()
from GroversTutorial import FactorizeWithGrovers2
import matplotlib.pyplot as plt
import numpy as np

def main():

    # Instantiate variables
    frequency = {}
    N_Experiments = 1000
    results = []
    number = 21

    # Run N_Experiments times the Q# operation.
    for i in range(N_Experiments):
        print(f'Experiment: {i} of {N_Experiments}')
        results.append(FactorizeWithGrovers2.simulate(number = number))

    # Store the results in a dictionary
    for i in results:
        if i in frequency:
            frequency[i]=frequency[i]+1
        else:
            frequency[i]=1

    # Sort and print the results
    frequency = dict(reversed(sorted(frequency.items(), key=lambda item: item[1])))
    print('Output, Frequency' )
    for k, v in frequency.items():
        print(f'{k:<8} {v}')

    # Plot an histogram with the results
    plt.bar(frequency.keys(), frequency.values())
    plt.xlabel("Output")
    plt.ylabel("Frequency of the outputs")
    plt.title("Outputs for Grover's factoring. N=21, 1000 iterations")
    plt.xticks(np.arange(1, 33, 2.0))
    plt.show()

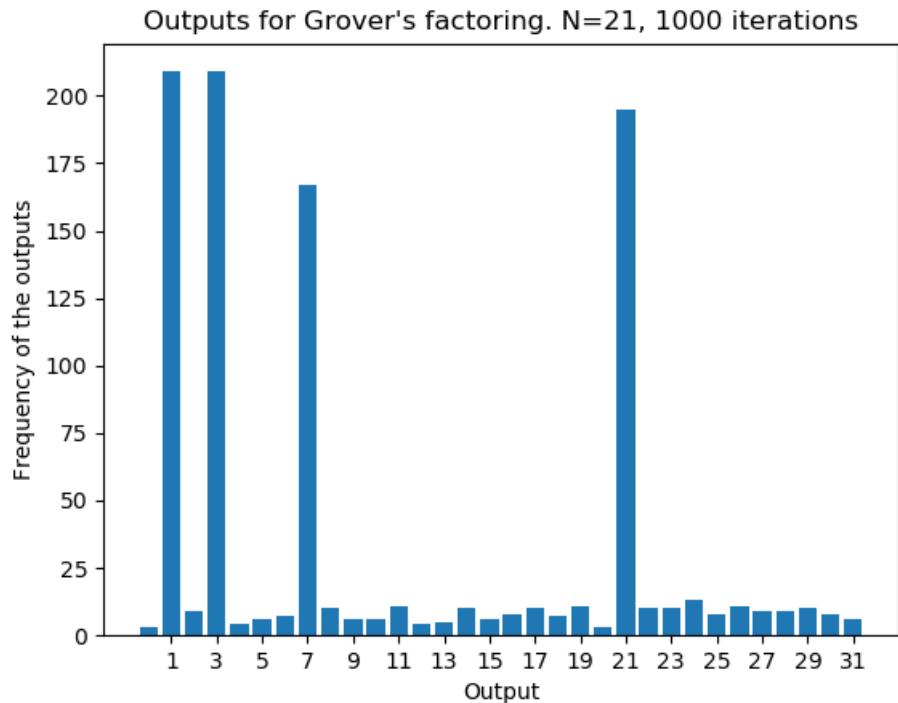
if __name__ == "__main__":
    main()

```

#### NOTE

The line `from GroversTutorial import FactorizeWithGrovers2` in the Python program imports the Q# code we've written previously. Note that the Python module name (`GroversTutorial`) needs to be identical to the Namespace of the operation we want to import (in this case, `FactorizeWithGrovers2`).

The program generates the following histogram:



As you can see in the histogram, the algorithm outputs the solutions to the search problem (1, 3, 7 and 21) with much higher probability than the non-solutions. You can think of Grover's algorithm as a quantum random generator that is purposefully biased towards those indices that are solutions to the search problem.

## Next steps

Now that you know how to implement Grover's algorithm, try to transform a mathematical problem into a search task and solve it with Q# and Grover's algorithm.

# Tutorial: Write and simulate qubit-level programs in Q#

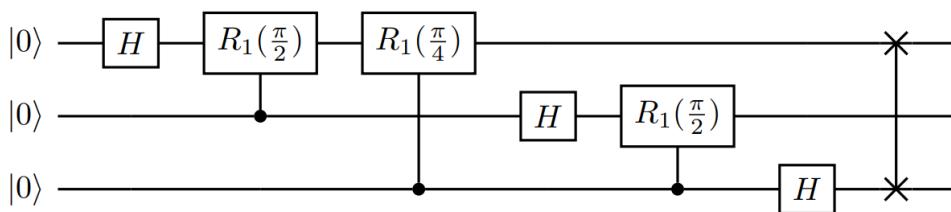
5/27/2021 • 21 minutes to read • [Edit Online](#)

Welcome to the Quantum Development Kit (QDK) tutorial on writing and simulating a basic quantum program that operates on individual qubits.

Although Q# was primarily created as a high-level programming language for large-scale quantum programs, it can just as easily be used to explore the lower level of quantum programs: directly addressing specific qubits. The flexibility of Q# allows users to approach quantum systems from any such level of abstraction, and in this tutorial we dive into the qubits themselves. Specifically, we take a look under the hood of the [quantum Fourier transform](#), a subroutine that is integral to many larger quantum algorithms.

Note that this low-level view of quantum information processing is often described in terms of "[quantum circuits](#)," which represent the sequential application of gates to specific qubits of a system.

Thus, the single- and multi-qubit operations we sequentially apply can be readily represented in a "circuit diagram." In our case, we will define a Q# operation to perform the full three-qubit quantum Fourier transform, which has the following representation as a circuit:



## Prerequisites

- [Install](#) the QDK using your preferred language and development environment.
- If you already have the QDK installed, make sure you have [updated](#) to the latest version

## In this tutorial, you'll learn how to:

- Define quantum operations in Q#
- Call Q# operations directly from the command prompt or using a classical host program
- Simulate a quantum operation from qubit allocation to measurement output
- Observe how the quantum system's simulated wavefunction evolves throughout the operation

Running a quantum program with the QDK typically consists of two parts:

1. The program itself, which is implemented using the Q# quantum programming language, and then invoked to run on a quantum computer or quantum simulator. These consist of
  - Q# operations: subroutines acting on quantum registers, and
  - Q# functions: classical subroutines used within the quantum algorithm.
2. The entry point used to call the quantum program and specify the target machine on which it should be run. This can be done directly from the command prompt, or through a host program written in a classical programming language like Python or C#. This tutorial includes instructions for whichever method you prefer.

# Allocate qubits and define quantum operations

The first part of this tutorial consists of defining the Q# operation `Perform3qubitQFT`, which performs the quantum Fourier transform on three qubits.

In addition, we will use the `DumpMachine` function to observe how the simulated wavefunction of our three qubit system evolves across the operation.

The first step is creating your Q# project and file. The steps for this depend on the environment you will use to call the program, and you can find the details in the respective [installation guides](#).

We will walk you through the components of the file step-by-step, but the code is also available as a full block below.

## Namespaces to access other Q# operations

Inside the file, we first define the namespace `NamespaceQFT` which will be accessed by the compiler. For our operation to make use of existing Q# operations, we open the relevant `Microsoft.Quantum.<>` namespaces.

```
namespace NamespaceQFT {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Diagnostics;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Arrays;

    // operations go here
}
```

## Define operations with arguments and returns

Next, we define the `Perform3qubitQFT` operation:

```
operation Perform3qubitQFT() : Unit {
    // do stuff
}
```

For now, the operation takes no arguments and does not return anything---in this case we write that it returns a `Unit` object, which is akin to `void` in C# or an empty tuple, `Tuple[()]`, in Python. Later, we will modify it to return an array of measurement results, at which point `Unit` will be replaced by `Result[]`.

## Allocate qubits with `use`

Within our Q# operation, we first allocate a register of three qubits with the `use` keyword:

```
use qs = Qubit[3];

Message("Initial state |000>:");
DumpMachine();
```

With `use`, the qubits are automatically allocated in the  $|\text{ket}{0}\rangle$  state. We can verify this by using `Message(<string>)` and `DumpMachine()`, which print a string and the system's current state to the console.

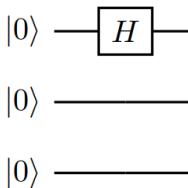
## NOTE

The `Message(<string>)` and `DumpMachine()` functions (from `Microsoft.Quantum.Intrinsic` and `Microsoft.Quantum.Diagnostics`, respectively) both print directly to the console. Just like a real quantum computation, Q# does not allow us to directly access qubit states. However, as `DumpMachine` prints the target machine's current state, it can provide valuable insight for debugging and learning when used in conjunction with the full state simulator.

## Applying single-qubit and controlled gates

Next, we apply the gates which comprise the operation itself. Q# already contains many basic quantum gates as operations in the `Microsoft.Quantum.Intrinsic` namespace, and these are no exception.

Within a Q# operation, the statements invoking callables will, of course, be run in sequential order. Hence, the first gate to apply is the `H` (Hadamard) to the first qubit:



To apply an operation to a specific qubit from a register (for example, a single `Qubit` from an array `Qubit[]`) we use standard index notation. So, applying the `H` to the first qubit of our register `qs` takes the form:

```
H(qs[0]);
```

Besides applying the `H` (Hadamard) gate to individual qubits, the QFT circuit consists primarily of controlled `R1` rotations. An `R1(<theta>, <qubit>)` operation in general leaves the  $\lvert \text{ket}0 \rangle$  component of the qubit unchanged, while applying a rotation of  $e^{i\theta}$  to the  $\lvert \text{ket}1 \rangle$  component.

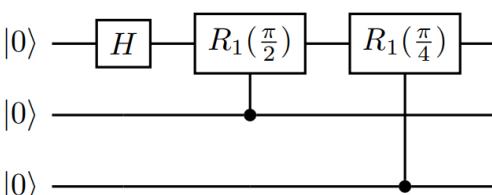
### Controlled operations

Q# makes it extremely easy to condition the run of an operation upon one or multiple control qubits. In general, we merely preface the call with `Controlled`, and the operation arguments change as:

```
Op(<normal args>) $\to$ Controlled Op([<control qubits>], (<normal args>)).
```

Note that the control qubits must be provided as an array, even if it is a single qubit.

After the `H`, we see that the next gates are the `R1` gates acting on the first qubit (and controlled by the second/third):



We call these with

```
Controlled R1([qs[1]], (PI()/2.0, qs[0]));
Controlled R1([qs[2]], (PI()/4.0, qs[0]));
```

Note that we use the `PI()` function from the `Microsoft.Quantum.Math` namespace to define the rotations in terms of pi radians. Additionally, we divide by a `Double` (for example, `2.0`) because dividing by an integer `2`

would throw a type error.

#### TIP

`R1(π/2)` and `R1(π/4)` are equivalent to the `s` and `t` operations (also in `Microsoft.Quantum.Intrinsic`).

After applying the relevant `H` operations and controlled rotations to the second and third qubits:

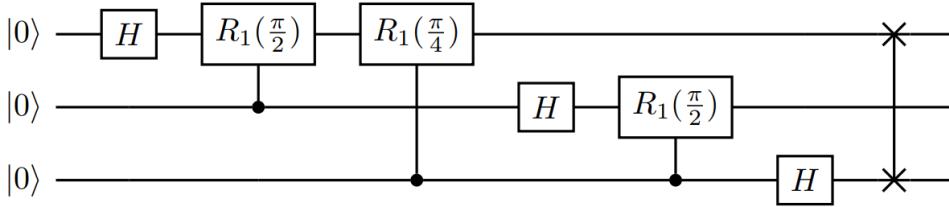
```
//second qubit:  
H(qs[1]);  
Controlled R1([qs[2]], (PI()/2.0, qs[1]));  
  
//third qubit:  
H(qs[2]);
```

we need only apply a `SWAP` gate to complete the circuit:

```
SWAP(qs[2], qs[0]);
```

This is necessary because the nature of the quantum Fourier transform outputs the qubits in reverse order, so the swaps allow for seamless integration of the subroutine into larger algorithms.

Hence we have finished writing the qubit-level operations of the quantum Fourier transform into our Q# operation:



However, we can't call it a day just yet. Our qubits were in state  $\lvert \text{ket}0 \rangle$  when we allocated them, and much like in life, in Q# we should leave things the same way we found them (or better!).

#### Deallocate qubits

We call `DumpMachine()` again to see the post-operation state, and finally apply `ResetAll` to the qubit register to reset our qubits to  $\lvert \text{ket}0 \rangle$  before completing the operation:

```
Message("After:");  
DumpMachine();  
  
ResetAll(qs);
```

Requiring that all deallocated qubits be explicitly set to  $\lvert \text{ket}0 \rangle$  is a basic feature of Q#, as it allows other operations to know their state precisely when they begin using those same qubits (a scarce resource). Additionally, this assures that they not be entangled with any other qubits in the system. If the reset is not performed at the end of a `use` allocation block, a runtime error might be thrown.

Your full Q# file should now look like this:

```

namespace NamespaceQFT {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Diagnostics;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Arrays;

    operation Perform3qubitQFT() : Unit {

        use qs = Qubit[3];

        Message("Initial state |000>:");
        DumpMachine();

        //QFT:
        //first qubit:
        H(qs[0]);
        Controlled R1([qs[1]], (PI()/2.0, qs[0]));
        Controlled R1([qs[2]], (PI()/4.0, qs[0]));

        //second qubit:
        H(qs[1]);
        Controlled R1([qs[2]], (PI()/2.0, qs[1]));

        //third qubit:
        H(qs[2]);

        SWAP(qs[2], qs[0]);

        Message("After:");
        DumpMachine();

        ResetAll(qs);
    }
}

```

With the Q# file and operation complete, our quantum program is ready to be called and simulated.

## Run the program

Having defined our Q# operation in a `.qs` file, we now need to call that operation and observe any returned classical data. For now, there isn't anything returned (recall that our operation defined above returns `Unit`), but when we later modify the Q# operation to return an array of measurement results (`Result[]`), we will address this.

While the Q# program is ubiquitous across the environments used to call it, the manner of doing so will of course vary. As such, simply follow the instructions in the tab corresponding to your setup: working from the Q# application or using a host program in Python or C#.

- [Command prompt](#)
- [Python](#)
- [C#](#)

Running the Q# program from the command prompt requires only a small change to the Q# file.

Simply add `@EntryPoint()` to a line preceding the operation definition:

```

@EntryPoint()
operation Perform3qubitQFT() : Unit {
    // ...

```

To run the program, open the terminal in the folder of your project and enter

```
dotnet run
```

Upon completion, you should see the `Message` and `DumpMachine` outputs below printed in your console.

```
Initial state |000>:  
# wave function for qubits with ids (least to most significant): 0;1;2  
|0>: 1.000000 + 0.000000 i == ***** [ 1.000000 ] --- [ 0.00000 rad ]  
|1>: 0.000000 + 0.000000 i == [ 0.000000 ]  
|2>: 0.000000 + 0.000000 i == [ 0.000000 ]  
|3>: 0.000000 + 0.000000 i == [ 0.000000 ]  
|4>: 0.000000 + 0.000000 i == [ 0.000000 ]  
|5>: 0.000000 + 0.000000 i == [ 0.000000 ]  
|6>: 0.000000 + 0.000000 i == [ 0.000000 ]  
|7>: 0.000000 + 0.000000 i == [ 0.000000 ]  
After:  
# wave function for qubits with ids (least to most significant): 0;1;2  
|0>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]  
|1>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]  
|2>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]  
|3>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]  
|4>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]  
|5>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]  
|6>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]  
|7>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]
```

When called on the full-state simulator, `DumpMachine()` provides these multiple representations of the quantum state's wavefunction. The possible states of an  $n$ -qubit system can be represented by  $2^n$  computational basis states, each with a corresponding complex coefficient (simply an amplitude and a phase). The computational basis states correspond to all the possible binary strings of length  $n$ —that is, all the possible combinations of qubit states  $\lvert \text{ket}{0} \rangle$  and  $\lvert \text{ket}{1} \rangle$ , where each binary digit corresponds to an individual qubit.

The first row provides a comment with the IDs of the corresponding qubits in their significant order. Qubit `2` being the "most significant" simply means that in the binary representation of basis state vector  $\lvert \text{ket}{i} \rangle$ , the state of qubit `2` corresponds to the left-most digit. For example,  $\lvert \text{ket}{6} \rangle = \lvert \text{ket}{110} \rangle$  comprises qubits `2` and `1` both in  $\lvert \text{ket}{1} \rangle$  and qubit `0` in  $\lvert \text{ket}{0} \rangle$ .

The rest of the rows describe the probability amplitude of measuring the basis state vector  $\lvert \text{ket}{i} \rangle$  in both Cartesian and polar formats. In detail for the first row of our input state  $\lvert \text{ket}{000} \rangle$ :

- `|0>`: this row corresponds to the `0` computational basis state (given that our initial state post-allocation was  $\lvert \text{ket}{000} \rangle$ , we would expect this to be the only state with probability amplitude at this point).
- `1.000000 + 0.000000 i`: the probability amplitude in Cartesian format.
- `==`: the `equal` sign separates both equivalent representations.
- `*****`: A graphical representation of the magnitude, the number of `*` is proportionate to the probability of measuring this state vector.
- `[ 1.000000 ]`: the numeric value of the magnitude
- `---`: A graphical representation of the amplitude's phase.
- `[ 0.000 rad ]`: the numeric value of the phase (in radians).

Both the magnitude and the phase are displayed with a graphical representation. The magnitude representation is straightforward: it shows a bar of `*`, and the higher the probability, the larger the bar will be. For the phase, see [Testing and debugging: dump functions](#) for the possible symbol representations based on angle ranges.

So, the printed output is illustrating that our programmed gates transformed our state from

```
$$ \ket{\psi}_{\text{initial}} = \ket{000} $$
```

to

```
$$ \begin{aligned} \ket{\psi}_{\text{final}} &= \frac{1}{\sqrt{8}} (\ket{000} + \ket{001} + \ket{010} + \ket{011} + \\ &\quad \ket{100} + \ket{101} + \ket{110} + \ket{111}) \\ &= \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} \ket{j}, \end{aligned} $$
```

which is precisely the behavior of the 3-qubit Fourier transform.

If you are curious about how other input states are affected, we encourage you to play around with applying qubit operations before the transform.

## Adding measurements

Unfortunately, a cornerstone of quantum mechanics tells us that a real quantum system cannot have such a `DumpMachine` function. Instead, we're forced to extract information through measurements, which in general not only fail to provide us the full quantum state, but can also drastically alter the system itself. There are many sorts of quantum measurements, but we will focus on the most basic: projective measurements on single qubits. Upon measurement in a given basis (for example, the computational basis  $\{\ket{0}, \ket{1}\}$ ), the qubit state is projected onto whichever basis state was measured---hence destroying any superposition between the two.

To implement measurements within a Q# program, we use the `M` operation (from `Microsoft.Quantum.Intrinsic`), which returns a `Result` type.

First, we modify our `Perform3QubitQFT` operation to return an array of measurement results, `Result[]`, instead of `Unit`.

```
operation Perform3QubitQFT() : Result[] {
```

**Define and initialize `Result[]` array**

Before even allocating qubits (for example, before the `use` statement), we declare and bind this length-3 array (one `Result` for each qubit):

```
mutable resultArray = new Result[3];
```

The `mutable` keyword prefacing `resultArray` allows the variable to be rebound later in the code---for example, when adding our measurement results.

**Perform measurements in a `for` loop and add results to array**

After the Fourier transform operations insert the following code:

```
for i in IndexRange(qs) {
    set resultArray w/= i -> M(qs[i]);
}
```

The `IndexRange` function called on an array (for example, our array of qubits, `qs`) returns a range over the indices of the array. Here, we use it in our `for` loop to sequentially measure each qubit using the `M(qs[i])` statement. Each measured `Result` type (either `Zero` or `One`) is then added to the corresponding index position in `resultArray` with an update-and-reassign statement.

## NOTE

The syntax of this statement is unique to Q#, but corresponds to the similar variable reassignment

```
resultArray[i] <- M(qs[i])
```

seen in other languages such as F# and R.

The keyword `set` is always used to reassign variables bound using `mutable`.

**Return** `resultArray`

With all three qubits measured and the results added to `resultArray`, we are safe to reset and deallocate the qubits as before. To return the measurements, insert:

```
return resultArray;
```

## Understanding the effects of measurement

Let's change the placement of our `DumpMachine` functions to output the state before and after the measurements. The final operation code should look like:

```
operation Perform3QubitQFT() : Result[] {  
  
    mutable resultArray = new Result[3];  
  
    use qs = Qubit[3];  
  
    //QFT:  
    //first qubit:  
    H(qs[0]);  
    Controlled R1([qs[1]], (PI()/2.0, qs[0]));  
    Controlled R1([qs[2]], (PI()/4.0, qs[0]));  
  
    //second qubit:  
    H(qs[1]);  
    Controlled R1([qs[2]], (PI()/2.0, qs[1]));  
  
    //third qubit:  
    H(qs[2]);  
  
    SWAP(qs[2], qs[0]);  
  
    Message("Before measurement: ");  
    DumpMachine();  
  
    for i in IndexRange(qs) {  
        set resultArray w/= i <- M(qs[i]);  
    }  
  
    Message("After measurement: ");  
    DumpMachine();  
  
    ResetAll(qs);  
  
    return resultArray;  
}
```

If you are working from the command prompt, the returned array will simply be displayed directly to the console at the end of the run. Otherwise, update your host program to process the returned array.

- [Command prompt](#)
- [Python](#)

- C#

To have more understanding of the returned array which will be printed in the console, we can add another `Message` in the Q# file just before the `return` statement:

```
Message("Post-QFT measurement results [qubit0, qubit1, qubit2]: ");
return resultArray;
```

Run the project, and your output should look similar to the following:

```
Before measurement:
# wave function for qubits with ids (least to most significant): 0;1;2
|0>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]
|1>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]
|2>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]
|3>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]
|4>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]
|5>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]
|6>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]
|7>: 0.353553 + 0.000000 i == *** [ 0.125000 ] --- [ 0.00000 rad ]
After measurement:
# wave function for qubits with ids (least to most significant): 0;1;2
|0>: 0.000000 + 0.000000 i == [ 0.000000 ]
|1>: 0.000000 + 0.000000 i == [ 0.000000 ]
|2>: 0.000000 + 0.000000 i == [ 0.000000 ]
|3>: 1.000000 + 0.000000 i == **** [ 1.000000 ] --- [ 0.00000 rad ]
|4>: 0.000000 + 0.000000 i == [ 0.000000 ]
|5>: 0.000000 + 0.000000 i == [ 0.000000 ]
|6>: 0.000000 + 0.000000 i == [ 0.000000 ]
|7>: 0.000000 + 0.000000 i == [ 0.000000 ]

Post-QFT measurement results [qubit0, qubit1, qubit2]:
[One,One,Zero]
```

This output illustrates a few different things:

1. Comparing the returned result to the pre-measurement `DumpMachine`, it clearly does *not* illustrate the post-QFT superposition over basis states. A measurement only returns a single basis state, with a probability determined by the amplitude of that state in the system's wavefunction.
2. From the post-measurement `DumpMachine`, we see that measurement *changes* the state itself, projecting it from the initial superposition over basis states to the single basis state that corresponds to the measured value.

If we were to repeat this operation many times, we would see the result statistics begin to illustrate the equally weighted superposition of the post-QFT state that gives rise to a random result on each shot. *However*, besides being inefficient and still imperfect, this would nevertheless only reproduce the relative amplitudes of the basis states, not the relative phases between them. The latter is not an issue in this example, but we would see relative phases appear if given a more complex input to the QFT than  $\lvert 000 \rangle$ .

#### Partial measurements

To explore how measuring only some qubits of the register can affect the system's state, try adding the following inside the `for` loop, after the measurement line:

```
let iString = IntAsString(i);
Message("After measurement of qubit " + iString + ":");
DumpMachine();
```

Note that to access the `IntAsString` function you will have to add

```
open Microsoft.Quantum.Convert;
```

with the rest of the namespace `open` statements.

In the resulting output, you will see the gradual projection into subspaces as each qubit is measured.

## Use the Q# libraries

As we mentioned in the introduction, much of Q#'s power rests in the fact that it allows you to abstract-away the worries of dealing with individual qubits. Indeed, if you want to develop full-scale, applicable quantum programs, worrying about whether an `H` operation goes before or after a particular rotation would only slow you down.

The Q# libraries contain the `QFT` operation, which you can simply take and apply for any number of qubits. To give it a try, define a new operation in your Q# file which has the same contents of `Perform3QubitQFT`, but with everything from the first `H` to the `SWAP` replaced by two easy lines:

```
let register = BigEndian(qs);    //from Microsoft.Quantum.Arithmetic  
QFT(register);                 //from Microsoft.Quantum.Canon
```

The first line simply creates a `BigEndian` expression of the allocated array of qubits, `qs`, which is what the `QFT` operation takes as an argument. This corresponds to index ordering of the qubits in the register.

To have access to these operations, add `open` statements for their respective namespaces at the beginning of the Q# file:

```
open Microsoft.Quantum.Canon;  
open Microsoft.Quantum.Arithmetic;
```

Now, adjust your host program to call the name of your new operation (e.g. `PerformIntrinsicQFT`), and give it a whirl.

To see the real benefit of using the Q# library operations, change the number of qubits to something other than `3`:

```
mutable resultArray = new Result[4];  
  
use qs = Qubit[4];  
//...
```

You can thus apply the proper QFT for any given number of qubits, without having to worry about the mess of new `H` operations and rotations on each qubit.

Note that the quantum simulator takes exponentially more time to run as you increase the number of qubits---precisely why we look forward to real quantum hardware!

# Learn quantum computing with the Quantum Katas

3/5/2021 • 4 minutes to read • [Edit Online](#)

The [Quantum Katas](#) are open-source, self-paced tutorials and programming exercises aimed at teaching the elements of quantum computing and Q# programming at the same time.

## Learning by doing

The tutorials and exercises collected in this project emphasize learning by doing: they offer programming tasks that cover certain topics which progress from very simple to quite challenging. Each task asks you to fill in some code; the first tasks might require just one line, and the later ones might require a sizable fragment of code.

Most importantly, the katas include testing frameworks that set up, run and validate the solutions to the tasks. This allows you to get immediate feedback on your solution and to reconsider your approach if it is incorrect.

You can use the katas for learning in your environment of choice:

- Jupyter Notebooks online within the Binder environment
- Jupyter Notebooks running on your local machine
- Visual Studio
- Visual Studio Code

## What can I learn with the Quantum Katas?

Explore the basics and fundamentals of quantum computing or dive deeper into quantum algorithms and protocols. We recommend you to follow this learning path in the beginning to make sure you have a solid grasp on the fundamental concepts of quantum computing. Of course, you can skip the topics you're comfortable with, such as complex arithmetic, and learn the algorithms in any order you want.

### Introduction to quantum computing concepts

KATA	DESCRIPTION
<a href="#">Complex arithmetic</a>	This tutorial explains some of the mathematical background required to work with quantum computing, such as imaginary and complex numbers.
<a href="#">Linear algebra</a>	Linear algebra is used to represent quantum states and operations in quantum computing. This tutorial covers the basics, including matrices and vectors.
<a href="#">The concept of a qubit</a>	Learn about qubits - one of the core concepts of quantum computing.
<a href="#">Single-qubit quantum gates</a>	This tutorial introduces single-qubit quantum gates, which act as the building blocks of quantum algorithms and transform quantum qubit states in various ways.
<a href="#">Multi-qubit systems</a>	This tutorial introduces multi-qubit systems, their representation in mathematical notation and in Q# code, and the concept of entanglement.

KATA	DESCRIPTION
<a href="#">Multi-qubit quantum gates</a>	This tutorial follows the <a href="#">Single-qubit quantum gates</a> tutorial, and focuses on applying quantum gates to multi-qubit systems.

## Quantum computing fundamentals

KATA	DESCRIPTION
<a href="#">Recognizing quantum gates</a>	A series of exercises designed to get you familiar with the basic quantum gates in Q#. Includes exercises for basic single-qubit and multi-qubit gates, adjoint and controlled gates, and how to use gates to modify the state of a qubit.
<a href="#">Creating quantum superposition</a>	Use these exercises to get familiar with the concept of superposition and programming in Q#. Includes exercises for basic single-qubit and multi-qubit gates, superposition, and flow control and recursion in Q#.
<a href="#">Distinguishing quantum states using measurements</a>	Solve these exercises while learning about quantum measurement and orthogonal and non-orthogonal states.
<a href="#">Joint measurements</a>	Learn about joint parity measurements and how to use the <a href="#">Measure</a> operation to distinguish quantum states.

## Algorithms

KATA	DESCRIPTION
<a href="#">Quantum teleportation</a>	This kata explores quantum teleportation - a protocol which allows communicating a quantum state using only classical communication and previously shared quantum entanglement.
<a href="#">Superdense coding</a>	Superdense coding is a protocol that allows transmission of two bits of classical information by sending just one qubit using previously shared quantum entanglement.
<a href="#">Deutsch–Jozsa algorithm</a>	This algorithm is famous for being one of the first examples of a quantum algorithm that is exponentially faster than any deterministic classical algorithm.
<a href="#">Exploring high-level properties of Grover's search algorithm</a>	A high-level introduction to one of the most famous algorithms in quantum computing. It solves the problem of finding an input to a black box (oracle) that produces a particular output.
<a href="#">Implementing Grover's search algorithm</a>	This kata dives deeper into Grover's search algorithm, and covers writing oracles, performing steps of the algorithm, and finally putting it all together.
<a href="#">Solving real problems using Grover's algorithm: SAT problems</a>	A series of exercises that uses Grover's algorithm to solve realistic problems, using <a href="#">boolean satisfiability problems</a> (SAT) as an example.

KATA	DESCRIPTION
Solving real problems using Grover's algorithm: Graph coloring problems	This kata further explores Grover's algorithm, this time to solve <a href="#">constraint satisfaction problems</a> , using a graph coloring problem as an example.

## Protocols and libraries

KATA	DESCRIPTION
BB84 protocol for quantum key distribution	Learn about and implement a quantum key distribution protocol, <a href="#">BB84</a> , using qubits to exchange cryptographic keys.
Bit-flip error correcting code	Explore quantum error correction with the simplest of the quantum error-correction (QEC) codes - the three-qubit bit-flip code.
Phase estimation	Phase estimation algorithms are some of the most fundamental building blocks of quantum computing. Learn about phase estimation with these exercises that cover quantum phase estimation and how to prepare and run phase estimation routines in Q#.
Quantum arithmetic: Building ripple-carry adders	An in-depth series of exercises that explores <a href="#">ripple carry</a> addition on a quantum computer. Build an in-place quantum adder, expand on it with a different algorithm, and finally, build an in-place quantum subtractor.

## Entanglement games

KATA	DESCRIPTION
CHSH game	Explore quantum entanglement with an implementation of the <a href="#">CHSH</a> game. This <a href="#">nonlocal</a> game shows how quantum entanglement can be used to increase the players' chance of winning beyond what would be possible with a purely classical strategy.
GHZ game	The GHZ game is another nonlocal game, but involves three players.
Mermin-Peres magic square game	A series of exercises that explores <a href="#">quantum pseudo-telepathy</a> to solve a magic square game.

## Resources

See the full series of [Quantum Katas](#)

[Run the katas online](#)

# Quantum computing history and background

3/5/2021 • 4 minutes to read • [Edit Online](#)

A host of new computer technologies has emerged within the last few years, and quantum computing is arguably the technology requiring the greatest paradigm shift on the part of developers. Quantum computers were proposed in the 1980s by [Richard Feynman](#) and [Yuri Manin](#). The intuition behind quantum computing stemmed from what was often seen as one of the greatest embarrassments of physics: remarkable scientific progress faced with an inability to model even simple systems. You see, quantum mechanics was developed between 1900 and 1925 and it remains the cornerstone on which chemistry, condensed matter physics, and technologies ranging from computer chips to LED lighting ultimately rests. Yet despite these successes, even some of the simplest systems seemed to be beyond the human ability to model with quantum mechanics. This is because simulating systems of even a few dozen interacting particles requires more computing power than any conventional computer can provide over thousands of years!

There are many ways to understand why quantum mechanics is hard to simulate. Perhaps the simplest is to see that quantum theory can be interpreted as saying that matter, at a quantum level, is in a multitude of possible configurations (known as *states*). Unlike classical probability theory, these many configurations of the quantum state, which can be potentially observed, may interfere with each other like waves in a tide pool. This interference prevents the use of statistical sampling to obtain the quantum state configurations. Rather, we have to track *every possible* configuration a quantum system could be in if we want to understand the quantum evolution.

Consider a system of electrons where electrons can be in any of say \$40\$ positions. The electrons therefore may be in any of  $2^{40}$  configurations (since each position can either have or not have an electron). To store the quantum state of the electrons in a conventional computer memory would require in excess of \$130\$ GB of memory! This is substantial, but within the reach of some computers. If we allowed the particles to be in any of  $41$  positions, there would be twice as many configurations at  $2^{41}$  which in turn would require more than \$260\$ GB of memory to store the quantum state. This game of increasing the number of positions cannot be played indefinitely if we want to store the state conventionally as we quickly exceed memory capacities of the world's most powerful machines. At a few hundred electrons the memory required to store the system exceeds the number of particles in the universe; thus there is no hope with our conventional computers to ever simulate their quantum dynamics. And yet in nature, such systems readily evolve in time according to quantum mechanical laws, blissfully unaware of the inability to engineer and simulate their evolution with conventional computing power.

This observation led those with an early vision of quantum computing to ask a simple yet powerful question: can we turn this difficulty into an opportunity? Specifically, if quantum dynamics are hard to simulate what would happen if we were to build hardware that had quantum effects as fundamental operations? Could we simulate systems of interacting particles using a system that exploits exactly the same laws that govern them naturally? Could we investigate tasks that are entirely absent from nature, yet follow or benefit from quantum mechanical laws? These questions led to the genesis of quantum computing.

The foundational core of quantum computing is to store information in quantum states of matter and to use quantum gate operations to compute on that information, by harnessing and learning to "program" quantum interference. An early example of programming interference to solve a problem thought to be hard on our conventional computers was done by [Peter Shor](#) in 1994 for a problem known as factoring. Solving factoring brings with it the ability to break many of our public key cryptosystems underlying the security of e-commerce today, including RSA and Elliptic Curve Cryptography. Since that time, fast and efficient quantum computer algorithms have been developed for many of our hard classical tasks: simulating physical systems in chemistry, physics, and materials science, searching an unordered database, solving systems of linear equations, and

machine learning.

Designing a quantum program to harness interference may sound like a daunting challenge, and while it is, many techniques and tools, including the Quantum Development Kit (QDK), have been introduced to make quantum programming and algorithm development more accessible. There are a handful of basic strategies that can be used to manipulate quantum interference in a way useful for computing, while at the same time not causing the solution to be lost in a tangle of quantum possibilities. Quantum programming is a distinct art from classical programming requiring very different tools to understand and express quantum algorithmic thinking. Indeed, without general tools to aid a quantum developer in tackling the art of quantum programming, quantum algorithmic development is not so easy.

The Quantum Development Kit empowers a growing community with tools to unlock the quantum revolution for their tasks, problems, and solutions. Our high-level programming language, Q#, was designed to address the challenges of quantum information processing; it is integrated in a software stack that enables a quantum algorithm to be compiled down to the primitive operations of a quantum computer. Before approaching the programming language, it's helpful to review the basic principles on which quantum computing is based. We will take the fundamental rules of quantum computing to be axioms, rather than detailing their foundations in quantum mechanics. Additionally, we will assume basic familiarity with linear algebra (vectors, matrices, and so on). If a deeper study of quantum computing history and principles is desired, we refer you to the [reference section](#) containing more information.

# Work with vectors and matrices in quantum computing

3/26/2021 • 6 minutes to read • [Edit Online](#)

Some familiarity with vectors and matrices is essential to understand quantum computing. We provide a brief introduction below and interested readers are recommended to read a standard reference on linear algebra such as *Strang, G. (1993). Introduction to linear algebra (Vol. 3). Wellesley, MA: Wellesley-Cambridge Press* or an online reference such as [Linear Algebra](#).

A column vector (or simply *vector*)  $v$  of dimension (or size)  $n$  is a collection of  $n$  complex numbers  $(v_1, v_2, \dots, v_n)$  arranged as a column:

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

The norm of a vector  $v$  is defined as  $\sqrt{\sum_i |v_i|^2}$ . A vector is said to be of unit norm (or alternatively it is called a *unit vector*) if its norm is 1. The *adjoint of a vector*  $v$  is denoted  $v^\dagger$  and is defined to be the following row vector where  $*$  denotes the complex conjugate,

$$\begin{bmatrix} v_1 & \dots & v_n \end{bmatrix}^\dagger = \begin{bmatrix} v_1^* & \dots & v_n^* \end{bmatrix}$$

Notice that we distinguish between a column vector  $v$  and a row vector  $v^\dagger$ .

## Inner product

We can multiply two vectors together through the *inner product*, also known as a *dot product* or *scalar product*. As the name implies, the result of the inner product of two vectors is a scalar. The inner product gives the projection of one vector onto another and is invaluable in describing how to express one vector as a sum of other simpler vectors. The inner product between two column vectors  $u=(u_1, u_2, \dots, u_n)$  and  $v=(v_1, v_2, \dots, v_n)$ , denoted  $\langle u, v \rangle$  is defined as

$$\langle u, v \rangle = u^\dagger v = \begin{bmatrix} u_1^* & \dots & u_n^* \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = u_1^* v_1 + \dots + u_n^* v_n.$$

This notation also allows the norm of a vector  $v$  to be written as  $\sqrt{\langle v, v \rangle}$ .

We can multiply a vector with a number  $c$  to form a new vector whose entries are multiplied by  $c$ . We can also add two vectors  $u$  and  $v$  to form a new vector whose entries are the sum of the entries of  $u$  and  $v$ . These operations are depicted below:

$$\text{If } u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}, v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, \text{ then } au + bv = \begin{bmatrix} au_1 + bv_1 \\ au_2 + bv_2 \\ \vdots \\ au_n + bv_n \end{bmatrix}.$$

A *matrix* of size  $m \times n$  is a collection of  $mn$  complex numbers arranged in  $m$  rows and  $n$  columns as shown below:

$$M = \begin{bmatrix} M_{11} & M_{12} & \dots & M_{1n} \\ M_{21} & M_{22} & \dots & M_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ M_{m1} & M_{m2} & \dots & M_{mn} \end{bmatrix}$$

Note that a vector of dimension  $n$  is simply a matrix of size  $n \times 1$ . As with vectors, we can multiply a matrix with a number  $c$  to obtain a new matrix where every entry is multiplied with  $c$ , and we can add two matrices of the same size to produce a new matrix whose entries are the sum of the respective entries of the two

matrices.

## Matrix multiplication

We can also multiply two matrices  $M$  of dimension  $m \times n$  and  $N$  of dimension  $n \times p$  to get a new matrix  $P$  of dimension  $m \times p$  as follows:

$$\begin{aligned} & \begin{aligned} & \begin{aligned} & M_{11} \sim M_{12} \sim \cdots M_{1n} \\ & M_{21} \sim M_{22} \sim \cdots M_{2n} \\ & \vdots \\ & M_{m1} \sim M_{m2} \sim \cdots M_{mn} \end{aligned} \end{aligned} \begin{aligned} & \begin{aligned} & N_{11} \sim N_{12} \\ & N_{21} \sim N_{22} \sim \cdots N_{2p} \\ & \vdots \\ & N_{n1} \sim N_{n2} \sim \cdots N_{np} \end{aligned} \end{aligned} = \begin{aligned} & \begin{aligned} & P_{11} \sim P_{12} \sim \cdots P_{1p} \\ & P_{21} \sim P_{22} \sim \cdots P_{2p} \\ & \vdots \\ & P_{m1} \sim P_{m2} \sim \cdots P_{mp} \end{aligned} \end{aligned}$$

where the entries of  $P$  are  $P_{ik} = \sum_j M_{ij}N_{jk}$ . For example, the entry  $P_{11}$  is the inner product of the first row of  $M$  with the first column of  $N$ . Note that since a vector is simply a special case of a matrix, this definition extends to matrix-vector multiplication.

All the matrices we consider will either be square matrices, where the number of rows and columns are equal, or vectors, which corresponds to only 1 column. One special square matrix is the *identity matrix*, denoted  $\boldsymbol{\text{I}}$ , which has all its diagonal elements equal to 1 and the remaining elements equal to 0:

$$\boldsymbol{\text{I}} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

For a square matrix  $A$ , we say a matrix  $B$  is its *inverse* if  $AB = BA = \boldsymbol{\text{I}}$ . The inverse of a matrix need not exist, but when it exists it is unique and we denote it  $A^{-1}$ .

For any matrix  $M$ , the adjoint or conjugate transpose of  $M$  is a matrix  $N$  such that  $N_{ij} = M_{ji}^*$ . The adjoint of  $M$  is usually denoted  $M^\dagger$ . We say a matrix  $U$  is *unitary* if  $UU^\dagger = U^\dagger U = \boldsymbol{\text{I}}$  or equivalently,  $U^{-1} = U^\dagger$ . Perhaps the most important property of unitary matrices is that they preserve the norm of a vector. This happens because

$$\langle v, v \rangle = v^\dagger v = v^\dagger U U^{-1} v = v^\dagger U^{-1} v = \langle U v, v \rangle$$

A matrix  $M$  is said to be *Hermitian* if  $M = M^\dagger$ .

## Tensor product

Finally, another important operation is the *Kronecker product*, also called the *matrix direct product* or *tensor product*. Note that the Kronecker product is distinguished from matrix multiplication, which is an entirely different operation. In quantum computing theory, *tensor product* is commonly used to denote the Kronecker product.

Consider the two vectors  $v = \begin{bmatrix} a \\ b \end{bmatrix}$  and  $u = \begin{bmatrix} c \\ d \\ e \end{bmatrix}$ . Their tensor product is denoted as  $v \otimes u$  and results in a block matrix.

$$v \otimes u = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \\ 0 & 0 & d \\ 0 & 0 & e \end{bmatrix}$$

Notice that tensor product is an operation on two matrices or vectors of arbitrary size. The tensor product of two matrices  $M$  of size  $m \times n$  and  $N$  of size  $p \times q$  is a larger matrix  $P = M \otimes N$  of size  $mp \times nq$ , and is obtained from  $M$  and  $N$  as follows:

$$M \otimes N = \begin{bmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \ddots & \vdots \\ M_{m1} & \cdots & M_{mn} \end{bmatrix} \otimes \begin{bmatrix} N_{11} & \cdots & N_{1q} \\ \vdots & \ddots & \vdots \\ N_{p1} & \cdots & N_{pq} \end{bmatrix} = \begin{bmatrix} M_{11}N_{11} & \cdots & M_{11}N_{1q} & \cdots & M_{11}N_{p1} & \cdots & M_{11}N_{pq} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ M_{m1}N_{11} & \cdots & M_{m1}N_{1q} & \cdots & M_{m1}N_{p1} & \cdots & M_{m1}N_{pq} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ M_{1n}N_{11} & \cdots & M_{1n}N_{1q} & \cdots & M_{1n}N_{p1} & \cdots & M_{1n}N_{pq} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ M_{mn}N_{11} & \cdots & M_{mn}N_{1q} & \cdots & M_{mn}N_{p1} & \cdots & M_{mn}N_{pq} \end{bmatrix}$$

```

N_{p1} ~ ~ \cdots ~ ~ N_{pq} \end{bmatrix} ~ ~ \cdots ~ ~ M_{1n} \begin{bmatrix} N_{11} ~ ~ \cdots ~ ~ N_{1q} \\ \ddots \\ N_{p1} ~ ~ \cdots ~ ~ N_{pq} \end{bmatrix} \\ \ddots \\ M_{m1} \begin{bmatrix} N_{11} ~ ~ \cdots ~ ~ N_{1q} \\ \ddots \\ N_{p1} ~ ~ \cdots ~ ~ N_{pq} \end{bmatrix} ~ ~ \cdots ~ ~ M_{mn} \begin{bmatrix} N_{11} ~ ~ \cdots ~ ~ N_{1q} \\ \ddots \\ N_{p1} ~ ~ \cdots ~ ~ N_{pq} \end{bmatrix} \end{bmatrix}. \end{aligned}

```

This is better demonstrated with an example:

$$\begin{aligned}
& \begin{bmatrix} a & b & c & d \end{bmatrix} \otimes \\
& \begin{bmatrix} e & f & g & h \end{bmatrix} = \begin{bmatrix} a & \begin{bmatrix} e & f & g & h \end{bmatrix} & \dots & d \end{bmatrix} \\
& \begin{bmatrix} e & f & g & h \end{bmatrix} \otimes \begin{bmatrix} e & f & g & h \end{bmatrix} = \begin{bmatrix} ae & af & be & bf & \dots & ag & ah & bg & bh & \dots & ce & cf & de & df & \dots & cg & ch & dg & dh \end{bmatrix}.
\end{aligned}$$

A final useful notational convention surrounding tensor products is that, for any vector  $\mathbf{v}$  or matrix  $\mathbf{M}$ ,  $\mathbf{v}^{\otimes n}$  or  $\mathbf{M}^{\otimes n}$  is short hand for an  $n$ -fold repeated tensor product. For example:

$$\begin{aligned}
& \begin{aligned} & \begin{bmatrix} 1 & 0 \end{bmatrix}^{\otimes 1} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \\ & \begin{bmatrix} 1 & 0 \end{bmatrix}^{\otimes 2} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}, \\ & \begin{bmatrix} 1 & -1 \end{bmatrix}^{\otimes 2} = \begin{bmatrix} 1 & -1 & -1 & 1 \end{bmatrix}, \\ & \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^{\otimes 1} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}, \\ & \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^{\otimes 2} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{aligned} \\
& \begin{aligned} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}^{\otimes 1} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \\ & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}^{\otimes 2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{aligned} \\
& \begin{aligned} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^{\otimes 1} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \\ & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^{\otimes 2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{aligned} \\
& \vdots
\end{aligned}$$

# Advanced matrix concepts

3/5/2021 • 4 minutes to read • [Edit Online](#)

We now extend our manipulation of matrices to *eigenvalues*, *eigenvectors* and *exponentials* which form a fundamental set of tools we need to describe and implement quantum algorithms.

## Eigenvalues and eigenvectors

Let  $M$  be a square matrix and  $v$  be a vector that is not the all zeros vector (for example, the vector with all entries equal to  $0$ ).

We say  $v$  is an *eigenvector* of  $M$  if  $Mv = cv$  for some number  $c$ . We say  $c$  is the *eigenvalue* corresponding to the eigenvector  $v$ . In general a matrix  $M$  may transform a vector into any other vector, but an eigenvector is special because it is left unchanged except for being multiplied by a number. Note that if  $v$  is an eigenvector with eigenvalue  $c$ , then  $av$  is also an eigenvector (for any nonzero  $a$ ) with the same eigenvalue.

For example, for the identity matrix, every vector  $v$  is an eigenvector with eigenvalue  $1$ .

As another example, consider a *diagonal matrix*  $D$  which only has nonzero entries on the diagonal:

$$\begin{bmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & 0 & d_3 \end{bmatrix}$$

The vectors

$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ ,  $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$  and  $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

are eigenvectors of this matrix with eigenvalues  $d_1$ ,  $d_2$ , and  $d_3$ , respectively. If  $d_1$ ,  $d_2$ , and  $d_3$  are distinct numbers, then these vectors (and their multiples) are the only eigenvectors of the matrix  $D$ . In general, for a diagonal matrix it is easy to read off the eigenvalues and eigenvectors. The eigenvalues are all the numbers appearing on the diagonal, and their respective eigenvectors are the unit vectors with one entry equal to  $1$  and the remaining entries equal to  $0$ .

Note in the above example that the eigenvectors of  $D$  form a basis for  $3$ -dimensional vectors. A basis is a set of vectors such that any vector can be written as a linear combination of them. More explicitly,  $v_1$ ,  $v_2$ , and  $v_3$  form a basis if any vector  $v$  can be written as  $v = a_1 v_1 + a_2 v_2 + a_3 v_3$  for some numbers  $a_1$ ,  $a_2$ , and  $a_3$ .

Recall that a Hermitian matrix (also called self-adjoint) is a complex square matrix equal to its own complex conjugate transpose, while a unitary matrix is a complex square matrix whose inverse is equal to its adjoint or complex conjugate transpose. For Hermitian and unitary matrices, which are essentially the only matrices encountered in quantum computing, there is a general result known as the *spectral theorem*, which asserts the following: For any Hermitian or unitary matrix  $M$ , there exists a unitary  $U$  such that  $M = U^\dagger D U$  for some diagonal matrix  $D$ . Furthermore, the diagonal entries of  $D$  will be the eigenvalues of  $M$ .

We already know how to compute the eigenvalues and eigenvectors of a diagonal matrix  $D$ . Using this theorem we know that if  $v$  is an eigenvector of  $D$  with eigenvalue  $c$ , for example,  $Dv = cv$ , then  $U^\dagger v$  will be an eigenvector of  $M$  with eigenvalue  $c$ . This is because

$$M(U^\dagger v) = U^\dagger D U (U^\dagger v) = U^\dagger D (U U^\dagger v) = U^\dagger D v = U^\dagger c v = c U^\dagger v$$

## Matrix exponentials

A *matrix exponential* can also be defined in exact analogy to the exponential function. The matrix exponential of a matrix  $A$  can be expressed as

$$\$e^A = \boldsymbol{1} + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \dots \quad \$$$

This is important because quantum mechanical time evolution is described by a unitary matrix of the form  $e^{iB}$  for Hermitian matrix  $B$ . For this reason, performing matrix exponentials is a fundamental part of quantum computing and as such Q# offers intrinsic routines for describing these operations. There are many ways in practice to compute a matrix exponential on a classical computer, and in general numerically approximating such an exponential is fraught with peril. See ["Nineteen dubious ways to compute the exponential of a matrix."](#) *SIAM review* 20.4 (1978): 801-836 for more information about the challenges involved.

The easiest way to understand how to compute the exponential of a matrix is through the eigenvalues and eigenvectors of that matrix. Specifically, the spectral theorem discussed above says that for every Hermitian or unitary matrix  $A$  there exists a unitary matrix  $U$  and a diagonal matrix  $D$  such that  $A = U^\dagger D U$ . Because of the properties of unitarity we have that  $A^2 = U^\dagger D^2 U$  and similarly for any power  $p$ :  $A^p = U^\dagger D^p U$ . If we substitute this into the operator definition of the operator exponential we obtain:

$$\begin{aligned} \$e^A = U^\dagger & \left( \boldsymbol{1} + D + \frac{D^2}{2!} + \dots \right) U = U^\dagger \\ & \begin{bmatrix} \exp(D_{11}) & 0 & \cdots & 0 \\ 0 & \exp(D_{22}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \exp(D_{NN}) \end{bmatrix} U. \end{aligned} \quad \$$$

In other words, if you transform to the eigenbasis of the matrix  $A$  then computing the matrix exponential is equivalent to computing the ordinary exponential of the eigenvalues of the matrix. As many operations in quantum computing involve performing matrix exponentials, this trick of transforming into the eigenbasis of a matrix to simplify performing the operator exponential appears frequently and is the basis behind many quantum algorithms such as Trotter–Suzuki-style quantum simulation methods discussed later in this guide.

Another useful property is if  $B$  is both unitary and Hermitian, i.e.,  $B = B^\dagger = B^{-1}$  then  $B^2 = \boldsymbol{1}$ . By applying this rule to the above expansion of the matrix exponential and by grouping the  $\boldsymbol{1}$  and the  $B$  terms together, it can be seen that for any real value  $x$  the identity

$$\$e^{iBx} = \boldsymbol{1} \cos(x) + iB \sin(x) \quad \$$$

holds. This trick is especially useful because it allows to reason about the actions matrix exponentials have, even if the dimension of  $B$  is exponentially large, for the special case when  $B$  is both unitary and Hermitian.

# The qubit in quantum computing

3/5/2021 • 9 minutes to read • [Edit Online](#)

Just as bits are the fundamental object of information in classical computing, *qubits* (quantum bits) are the fundamental object of information in quantum computing. To understand this correspondence, this article looks at the simplest example: a single qubit.

## Representing a qubit

While a bit, or binary digit, can have value either \$0\$ or \$1\$, a qubit can have a value that is either of these or a quantum superposition of \$0\$ and \$1\$.

The state of a single qubit can be described by a two-dimensional column vector of unit norm, that is, the magnitude squared of its entries must sum to \$1\$. This vector, called the quantum state vector, holds all the information needed to describe the one-qubit quantum system just as a single bit holds all of the information needed to describe the state of a binary variable.

Any two-dimensional column vector of real or complex numbers with norm \$1\$ represents a possible quantum state held by a qubit. Thus  $\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$  represents a qubit state if  $|\alpha|^2 + |\beta|^2 = 1$ . Some examples of valid quantum state vectors representing qubits include

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$$

The quantum state vectors  $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$  and  $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$  take a special role. These two vectors form a basis for the vector space that describes the qubit's state. This means that any quantum state vector can be written as a sum of these basis vectors. Specifically, the vector  $\begin{bmatrix} x \\ y \end{bmatrix}$  can be written as  $x \begin{bmatrix} 1 \\ 0 \end{bmatrix} + y \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ . While any rotation of these vectors would serve as a perfectly valid basis for the qubit, we choose to privilege this one, by calling it the *computational basis*.

We take these two quantum states to correspond to the two states of a classical bit, namely \$0\$ and \$1\$. The standard convention is to choose

$$|0\rangle \equiv \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle \equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

although the opposite choice could equally well be taken. Thus, out of the infinite number of possible single-qubit quantum state vectors, only two correspond to states of classical bits; all other quantum states do not.

## Measuring a qubit

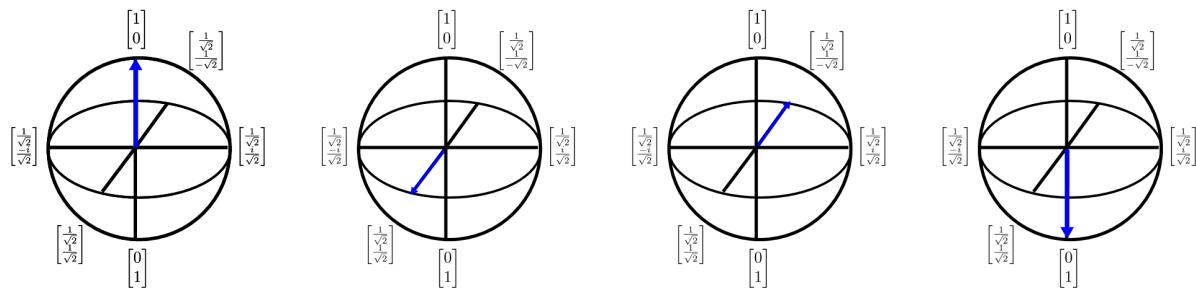
Now that we know how to represent a qubit, we can gain some intuition for what these states represent by discussing the concept of *measurement*. A measurement corresponds to the informal idea of "looking" at a qubit, which immediately collapses the quantum state to one of the two classical states  $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$  or  $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ . When a qubit given by the quantum state vector  $\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$  is measured, we obtain the outcome \$0\$ with probability  $|\alpha|^2$  and the outcome \$1\$ with probability  $|\beta|^2$ . On outcome \$0\$, the qubit's new state is  $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ ; on outcome \$1\$ its state is  $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ . Note that these probabilities sum up to \$1\$ because of the normalization condition  $|\alpha|^2 + |\beta|^2 = 1$ .

The properties of measurement also mean that the overall sign of the quantum state vector is irrelevant. Negating a vector is equivalent to  $\alpha \rightarrow -\alpha$  and  $\beta \rightarrow -\beta$ . Because the probability of measuring \$0\$ and \$1\$ depends on the magnitude squared of the terms, inserting such signs does not change the probabilities whatsoever. Such phases are commonly called “*global phases*” and more generally can be of the form  $e^{i\phi}$  rather than just  $\pm 1$ .

A final important property of measurement is that it does not necessarily damage all quantum state vectors. If we start with a qubit in the state  $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ , which corresponds to the classical state \$0\$, measuring this state will always yield the outcome \$0\$ and leave the quantum state unchanged. In this sense, if we only have classical bits (for example, qubits that are either  $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$  or  $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ ) then measurement does not damage the system. This means that we can replicate classical data and manipulate it on a quantum computer just as one could do on a classical computer. The ability, however, to store information in both states at once is what elevates quantum computing beyond what is possible classically and further robs quantum computers of the ability to copy quantum data indiscriminately, see also [the no-cloning theorem](#).

## Visualizing qubits and transformations using the bloch sphere

Qubits may also be pictured in 3D using the *Bloch sphere* representation. The Bloch sphere gives a way of describing a single-qubit quantum state (which is a two-dimensional complex vector) as a three-dimensional real-valued vector. This is important because it allows us to visualize single-qubit states and thereby develop reasoning that can be invaluable in understanding multi-qubit states (where sadly the Bloch sphere representation breaks down). The Bloch sphere can be visualized as follows:



The arrows in this diagram show the direction in which the quantum state vector is pointing and each transformation of the arrow can be thought of as a rotation about one of the cardinal axes. While thinking about a quantum computation as a sequence of rotations is a powerful intuition, it is challenging to use this intuition to design and describe algorithms. Q# alleviates this issue by providing a language for describing such rotations.

## Single-qubit operations

Quantum computers process data by applying a universal set of quantum gates that can emulate any rotation of the quantum state vector. This notion of universality is akin to the notion of universality for traditional (for example, classical) computing where a gate set is considered to be universal if every transformation of the input bits can be performed using a finite length circuit. In quantum computing, the valid transformations that we are allowed to perform on a qubit are unitary transformations and measurement. The *adjoint operation* or the complex conjugate transpose is of crucial importance to quantum computing because it is needed to invert quantum transformations.

There are only four functions that map one bit to one bit on a classical computer. In contrast, there are an infinite number of unitary transformations on a single qubit on a quantum computer. Therefore, no finite set of primitive quantum operations, called *gates*, can exactly replicate the infinite set of unitary transformations allowed in quantum computing. This means, unlike classical computing, it is impossible for a quantum computer to implement every possible quantum program exactly using a finite number of gates. Thus quantum computers cannot be universal in the same sense of classical computers. As a result, when we say that a set of

gates is *universal* for quantum computing we actually mean something slightly weaker than we mean with classical computing. For universality, we require that a quantum computer only *approximate* every unitary matrix within a finite error using a finite length gate sequence. In other words, a set of gates is a universal gate set if any unitary transformation can be approximately written as a product of gates from this set. We require that for any prescribed error bound, there exist gates  $G_1, G_2, \dots, G_N$  from the gate set such that

$$G_N G_{N-1} \cdots G_2 G_1 \approx U.$$

Note that because the convention for matrix multiplication is to multiply from right to left the first gate operation in this sequence,  $G_N$ , is actually the last one applied to the quantum state vector. More formally, we say that such a gate set is universal if for every error tolerance  $\epsilon$  there exists  $G_1, \dots, G_N$  such that the distance between  $G_N \cdots G_1$  and  $U$  is at most  $\epsilon$ . Ideally the value of  $N$  needed to reach this distance of  $\epsilon$  should scale poly-logarithmically with  $1/\epsilon$ .

What does such a universal gate set look like in practice? The simplest such universal gate set for single-qubit gates consists of only two gates: the Hadamard gate  $H$  and the so-called  $T$ -gate (also known as the  $i\pi/8$  gate):

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}.$$

However, for practical reasons related to quantum error correction it can be more convenient to consider a larger gate set, namely one that can be generated using  $H$  and  $T$ . We can classify the quantum gates into two categories: Clifford gates and the  $T$ -gate. This subdivision is useful because in many quantum error correction schemes the so-called Clifford gates are easy to implement, that is they require very few resources in terms of operations and qubits to implement fault tolerantly, whereas non-Clifford gates are quite costly when requiring fault tolerance. The standard set of single-qubit Clifford gates, [included by default in Q#](#), include

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \quad T^2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad HT^4 = H.$$

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = T^2 H T^4, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = T^4.$$

Here the operations  $X$ ,  $Y$  and  $Z$  are used especially frequently and are named *Pauli operators* after their creator Wolfgang Pauli. Together with the non-Clifford gate (the  $T$ -gate), these operations can be composed to approximate any unitary transformation on a single qubit.

For more information on these operations, their Bloch sphere representations and Q# implementations, see [Intrinsic Operations and Functions](#).

As an example of how unitary transformations can be built from these primitives, the three transformations pictured in the Bloch spheres above correspond to the gate sequence  $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ .

While the previous constitute the most popular primitive gates for describing operations on the logical level of the stack (think of the logical level as the level of the quantum algorithm), it is often convenient to consider less basic operations at the algorithmic level, for example operations closer to a function description level. Fortunately, Q# also has methods available for implementing higher-level unitaries, which in turn allow high-level algorithms to be implemented without explicitly decomposing everything down to Clifford and  $T$ -gates.

The simplest such primitive is the single qubit-rotation. Three single-qubit rotations are typically considered:  $R_x$ ,  $R_y$  and  $R_z$ . To visualize the action of the rotation  $R_x(\theta)$ , for example, imagine pointing your right thumb along the direction of the  $x$ -axis of the Bloch sphere and rotating the vector with your hand through an angle of  $\theta/2$  radians. This confusing factor of  $2$  arises from the fact that orthogonal vectors are  $180^\circ$  apart when plotted on the Bloch sphere, yet are actually  $90^\circ$  degrees apart geometrically. The corresponding unitary matrices are:

```

\begin{align*}
& R_z(\theta) = e^{-i\theta/2} \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \\
& R_x(\theta) = e^{-i\theta/2} H = \begin{bmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ i\sin(\theta/2) & \cos(\theta/2) \end{bmatrix} \\
& R_y(\theta) = e^{-i\theta/2} \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}
\end{align*}

```

Just as any three rotations can be combined together to perform an arbitrary rotation in three dimensions, it can be seen from the Bloch sphere representation that any unitary matrix can be written as a sequence of three rotations as well. Specifically, for every unitary matrix  $U$  there exists  $\alpha, \beta, \gamma, \delta$  such that  $U = e^{i\alpha} R_x(\beta) R_z(\gamma) R_x(\delta)$ . Thus  $R_z(\theta)$  and  $H$  also form a universal gate set although it is not a discrete set because  $\theta$  can take any value. For this reason, and due to applications in quantum simulation, such continuous gates are crucial for quantum computation, especially at the quantum algorithm design level. To achieve fault-tolerant hardware implementation, they will ultimately be compiled into discrete gate sequences that closely approximate these rotations.

# Operations on multiple qubits

5/27/2021 • 12 minutes to read • [Edit Online](#)

This article reviews the rules used to build multi-qubit states out of single-qubit states and discusses the gate operations needed to include in a gate set to form a universal many-qubit quantum computer. These tools are absolutely necessary to understand the gate sets that are commonly used in Q# code, and also to gain intuition about why quantum effects such as entanglement or interference render quantum computing more powerful than classical computing.

## Single-qubit vs. multi-qubit gates

The true power of quantum computing only becomes evident as we increase the number of qubits. Single-qubit gates possess some counter-intuitive features, such as the ability to be in more than one state at a given time. However, if all we had in a quantum computer were single-qubit gates, then a calculator and certainly a classical supercomputer would dwarf its computational power.

Quantum computing power arises, in part, because the dimension of the vector space of quantum state vectors grows exponentially with the number of qubits. This means that while a single qubit can be trivially modeled, simulating a fifty-qubit quantum computation would arguably push the limits of existing supercomputers. Increasing the size of the computation by only one additional qubit *doubles* the memory required to store the state and roughly *doubles* the computational time. This rapid doubling of computational power is why a quantum computer with a relatively small number of qubits can far surpass the most powerful supercomputers of today, tomorrow, and beyond for some computational tasks.

## Representing two qubits

If we are given two separate qubits, one in the state  $\psi = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$  and the other in the state  $\phi = \begin{bmatrix} \gamma \\ \delta \end{bmatrix}$ , the corresponding two-qubit state is given by the tensor product (or [Kronecker product](#)) of vectors, which is defined as follows

$$\begin{aligned} \psi \otimes \phi &= \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \otimes \begin{bmatrix} \gamma \\ \delta \end{bmatrix} = \begin{bmatrix} \alpha \begin{bmatrix} \gamma \\ \delta \end{bmatrix} \\ \beta \begin{bmatrix} \gamma \\ \delta \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{bmatrix} \end{aligned}$$

Therefore, given two single-qubit states  $\psi$  and  $\phi$ , each of dimension 2, the corresponding two-qubit state  $\psi \otimes \phi$  is 4-dimensional. The vector

$$\begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{bmatrix}$$

represents a quantum state on two qubits if

$|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$ . More generally, you can see that the quantum state of  $n$  qubits is represented by a unit vector  $v_1 \otimes v_2 \otimes \dots \otimes v_n$  of dimension  $2^n$  using this construction. Just as with single qubits, the quantum state vector of multiple qubits holds all the information needed to describe the system's behavior. For more information about vectors and tensor products, see [Vectors and Matrices in Quantum Computing](#).

The computational basis for two-qubit states is formed by the tensor products of one-qubit states. For example, we have

$$\begin{aligned} 00 &\equiv \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\ 01 &\equiv \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \\ 10 &\equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \\ 11 &\equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \end{aligned}$$

$$\begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad 10 \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Note that while we can always take the tensor product of two single-qubit states to form a two-qubit state, not all two-qubit quantum states can be written as the tensor product of two single-qubit states. For example, there are no states  $\psi = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$  and  $\phi = \begin{bmatrix} \gamma \\ \delta \end{bmatrix}$  such that their tensor product is the state

$$\psi \otimes \phi = \begin{bmatrix} 1/\sqrt{2} & 0 & 0 & 1/\sqrt{2} \end{bmatrix}$$

Such a two-qubit state, which cannot be written as the tensor product of single-qubit states, is called an "entangled state"; the two qubits are said to be *entangled*. Loosely speaking, because the quantum state cannot be thought of as a tensor product of single qubit states, the information that the state holds is not confined to either of the qubits individually. Rather, the information is stored non-locally in the correlations between the two states. This non-locality of information is one of the major distinguishing features of quantum computing over classical computing and is essential for a number of quantum protocols including [quantum teleportation](#) and [quantum error correction](#).

## Measure two-qubit states

Measuring two-qubit states is very similar to single-qubit measurements. Measuring the state

$$\begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{10} & \alpha_{11} \end{bmatrix}$$

yields \$00\$ with probability  $|\alpha_{00}|^2$ , \$01\$ with probability  $|\alpha_{01}|^2$ , \$10\$ with probability  $|\alpha_{10}|^2$ , and \$11\$ with probability  $|\alpha_{11}|^2$ . The variables  $\alpha_{00}$ ,  $\alpha_{01}$ ,  $\alpha_{10}$ , and  $\alpha_{11}$  were deliberately named to make this connection clear. After the measurement, if the outcome is \$00\$ then the quantum state of the two-qubit system has collapsed and is now

$$00 \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

It is also possible to measure just one qubit of a two-qubit quantum state. In cases where you measure only one of the qubits, the impact of measurement is subtly different because the entire state is not collapsed to a computational basis state, rather it is collapsed to only one sub-system. In other words, in such cases measuring only one qubit only collapses one of the subsystems but not all of them.

To see this consider measuring the first qubit of the following state, which is formed by applying the Hadamard transform  $H$  on two qubits initially set to the "0" state:

$$H^{\otimes 2} \left( \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \right) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$

Both outcomes have 50% probability of occurring. The outcome being 50% probability for both can be intuited from the fact that the initial quantum state vector is invariant under swapping \$0\$ with \$1\$ on the first qubit.

The mathematical rule for measuring the first or second qubit is simple. If we let  $e_k$  be the  $k^{\text{th}}$  computational basis vector and let  $S$  be the set of all  $e_k$  such that the qubit in question takes the value \$1\$ for that value of  $k$ . For example, if we are interested in measuring the first qubit then  $S$  would consist of  $e_1 \equiv 10$  and  $e_3 \equiv 11$ . Similarly, if we are interested in the second qubit  $S$  would consist of  $e_2 \equiv 01$  and  $e_3 \equiv 11$ . Then the probability of measuring the chosen qubit to be \$1\$ is for state vector  $\psi$

$$P(\text{outcome}=1) = \sum_{e_k \in S} |\psi_e|^2$$

#### NOTE

In this document we are using the little-endian format to label the computational basis. In little endian format, the least significant bits come first. For example, the number four in little-endian format is represented by the string of bits 001.

Since each qubit measurement can only yield \$0\$ or \$1\$, the probability of measuring \$0\$ is simply \$1 - P(\text{outcome}=1)\$. This is why we only explicitly give a formula for the probability of measuring \$1\$.

The action that such a measurement has on the state can be expressed mathematically as

$$\$ \$ \psi \mapsto \frac{\sum_{e_k \in \text{the set } S} e_k e_k^\dagger \psi}{\sqrt{P(\text{outcome}=1)}}. \$ \$$$

The cautious reader may worry about what happens when the probability of the measurement is zero. While the resultant state is technically undefined in this case, we never need to worry about such eventualities because the probability is zero!

If we take \$\psi\$ to be the uniform state vector given above and are interested in measuring the first qubit then

$$\$ \$ P(\text{measurement of first qubit}=1) = (\psi^\dagger e_1)(e_1^\dagger \psi) + (\psi^\dagger e_3)(e_3^\dagger \psi) = |e_1^\dagger \psi|^2 + |e_3^\dagger \psi|^2. \$ \$$$

Note that this is just the sum of the two probabilities that would be expected for measuring the results \$10\$ and \$11\$ were all the qubits to be measured. For our example, this evaluates to

$$\$ \$ \frac{1}{4}(|\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}|^2 + |\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}|^2) = \frac{1}{4}(\sqrt{2})^2 = \frac{1}{2}. \$ \$$$

which perfectly matches what our intuition tells us the probability should be. Similarly, the state can be written as

$$\$ \$ \frac{1}{\sqrt{2}}(\begin{bmatrix} e_1 \\ e_3 \end{bmatrix}) = \frac{1}{\sqrt{2}}\begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix} \$ \$$$

again in accordance with our intuition.

## Two-qubit operations

As in the single-qubit case, any unitary transformation is a valid operation on qubits. In general, a unitary transformation on \$n\$ qubits is a matrix \$U\$ of size \$2^n \times 2^n\$ (so that it acts on vectors of size \$2^n\$), such that \$U^{-1} = U^\dagger\$. For example, the CNOT (controlled-NOT) gate is a commonly used two-qubit gate and is represented by the following unitary matrix:

$$\$ \$ \text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \$ \$$$

We can also form two-qubit gates by applying single-qubit gates on both qubits. For example, if we apply the gates

$$\$ \$ \begin{bmatrix} a & b & c & d \end{bmatrix} \$ \$$$

and

$$\$ \$ \begin{bmatrix} e & f & g & h \end{bmatrix} \$ \$$$

to the first and second qubits, respectively, this is equivalent to applying the two-qubit unitary given by their tensor product: \$ \\$ \begin{bmatrix} a & b & c & d \end{bmatrix} \otimes \begin{bmatrix} e & f & g & h \end{bmatrix} = \begin{bmatrix} ae & af & be & bf & ag & ah & bg & bh & ce & cf & de & df & cg & ch & dg & dh \end{bmatrix} \\$ \\$ Thus we can form two-qubit gates by taking the tensor product of some known single-qubit gates. Some examples of two-qubit gates include \$H \otimes H\$, \$X \otimes \text{ctrl}\$, and \$X \otimes Z\$.

Note that while any two single-qubit gates define a two-qubit gate by taking their tensor product, the converse is not true. Not all two-qubit gates can be written as the tensor product of single-qubit gates. Such a gate is called an *entangling* gate. One example of an entangling gate is the CNOT gate.

The intuition behind a controlled-not gate can be generalized to arbitrary gates. A controlled gate in general is a gate that acts as identity (ie it has no action) unless a specific qubit is \$1\$. We denote a controlled unitary, controlled in this case on the qubit labeled \$x\$, with a \$\Lambda\_x(U)\$. As an example \$\Lambda\_0(U) e\_1 \otimes e\_1 = e\_1 \otimes U e\_1\$ and \$\Lambda\_0(U) e\_0 \otimes e\_0 = e\_0 \otimes e\_0\$, where \$e\_0\$ and \$e\_1\$ are the computational basis vectors for a single qubit corresponding to the values \$0\$ and \$1\$. For example, consider the following controlled-\$Z\$ gate then we can express this as 
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} = (\text{CNOT})(H)$$

Building controlled unitaries in an efficient manner is a major challenge. The simplest way to implement this requires forming a database of controlled versions of fundamental gates and replacing every fundamental gate in the original unitary operation with its controlled counterpart. This is often quite wasteful and clever insight often can be used to just replace a few gates with controlled versions to achieve the same impact. For this reason, we provide in our framework the ability to perform either the naive method of controlling or allow the user to define a controlled version of the unitary if an optimized hand-tuned version is known.

Gates can also be controlled using classical information. A classically controlled not-gate, for example, is just an ordinary not-gate but it is only applied if a classical bit is \$1\$ as opposed to a quantum bit. In this sense, a classically controlled gate can be thought of as an if statement in the quantum code wherein the gate is applied only in one branch of the code.

As in the single-qubit case, a two-qubit gate set is universal if any \$4\times 4\$ unitary matrix can be approximated by a product of gates from this set to arbitrary precision. One example of a universal gate set is the Hadamard gate, the T gate, and the CNOT gate. By taking products of these gates, we can approximate any unitary matrix on two qubits.

## Many-qubit systems

We follow exactly the same patterns explored in the two-qubit case to build many-qubit quantum states from smaller systems. Such states are built by forming tensor products of smaller states. For example, consider encoding the bit string \$1011001\$ in a quantum computer. We can encode this as

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \equiv \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Quantum gates work in exactly the same way. For example, if we wish to apply the \$X\$ gate to the first qubit and then perform a CNOT between the second and third qubits we may express this transformation as

$$\begin{aligned} & \begin{aligned} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \otimes X \otimes I \end{aligned} \\ & \begin{aligned} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \otimes I \end{aligned} \\ & \begin{aligned} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \otimes I \end{aligned} \\ & \begin{aligned} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \otimes I \end{aligned} \\ & \begin{aligned} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \otimes I \end{aligned} \end{aligned}$$

In many qubit systems, there is often a need to allocate and deallocate qubits that serve as temporary memory for the quantum computer. Such a qubit is said to be *auxiliary*. By default we assume the qubit state is initialized to \$e\_0\$ upon allocation. We further assume that it is returned again to \$e\_0\$ before deallocation. This assumption is important because if an auxiliary qubit becomes entangled with another qubit register when it becomes deallocated then the process of deallocation will damage the auxiliary qubit. For this reason, we always assume that such qubits are reverted to their initial state before being released.

Finally, although new gates needed to be added to our gate set to achieve universal quantum computing for two qubit quantum computers, no new gates need to be introduced in the multi-qubit case. The gates  $\$H\$$ ,  $\$T\$$  and CNOT form a universal gate set on many qubits because any general unitary transformation can be broken into a series of two qubit rotations. We then can leverage the theory developed for the two-qubit case and use it again here when we have many qubits.

While the linear algebraic notation that we have been using thus far can certainly be used to describe multi-qubit states, it becomes increasingly cumbersome as we increase the size of the states. The resulting column-vector for a length 7 bit string, for example, is  $\$128\$$  dimensional, which makes expressing it using the notation described previously very cumbersome. For this reason, we next present a common notation in quantum computing that allows us to concisely describe these high-dimensional vectors.

# Dirac notation

3/5/2021 • 10 minutes to read • [Edit Online](#)

*Dirac notation* is a language to fit the precise needs of expressing states in quantum mechanics. The examples in this article are suggestions that can be used to concisely express quantum ideas.

## Limitations of column vector notation

While column vector notation is ubiquitous in linear algebra, it is often cumbersome in quantum computing especially when dealing with multiple qubits. For example, when we define  $\psi$  to be a vector it is not explicitly clear whether  $\psi$  is a row or a column vector. Thus if  $\phi$  and  $\psi$  are vectors then it is equally unclear if  $\phi \psi$  is even defined because the shapes of  $\phi$  and  $\psi$  may be unclear in the context. Beyond the ambiguity about the shapes of vectors, expressing even simple vectors using the linear algebraic notation introduced earlier can be very cumbersome. For example, if we wish to describe an  $n$ -qubit state where each qubit takes the value  $0$  then we would formally express the state as

$$|\begin{bmatrix} 1 \\ 0 \end{bmatrix}\rangle \otimes \cdots \otimes |\begin{bmatrix} 1 \\ 0 \end{bmatrix}\rangle$$

Of course evaluating this tensor product is impractical because the vector lies in an exponentially large space and so this notation is in fact the best description of the state that can be given using the previous notation.

## Types of vectors in Dirac notation

There are two types of vectors in Dirac notation: the *bra* vector and the *ket* vector, so named because when put together they form a *braket* or inner product. If  $\psi$  is a column vector then we can write it in Dirac notation as  $|\psi\rangle$ , where the  $|\cdot\rangle$  denotes that it is a unit column vector, for example, a *ket* vector. Similarly, the row vector  $\psi^\dagger$  is expressed as  $\langle\psi|$ . In other words,  $\psi^\dagger$  is obtained by applying entry-wise complex conjugation to the elements of the transpose of  $\psi$ . The bra-ket notation directly implies that  $\langle\psi|\psi\rangle$  is the inner product of vector  $\psi$  with itself, which is by definition  $1$ .

More generally, if  $\psi$  and  $\phi$  are quantum state vectors their inner product is  $\langle\phi|\psi\rangle$  which implies that the probability of measuring the state  $|\psi\rangle$  to be  $|\phi\rangle$  is  $|\langle\phi|\psi\rangle|^2$ .

The following convention is used to describe the quantum states that encode the values of zero and one (the single-qubit computational basis states):

$$|\begin{bmatrix} 1 \\ 0 \end{bmatrix}\rangle = |\psi_0\rangle, \quad |\begin{bmatrix} 0 \\ 1 \end{bmatrix}\rangle = |\psi_1\rangle$$

## Example: Represent the Hadamard operation with Dirac notation

The following notation is often used to describe the states that result from applying the Hadamard gate to  $|\psi_0\rangle$  and  $|\psi_1\rangle$  (which correspond to the unit vectors in the  $+x$  and  $-x$  directions on the Bloch sphere):

$$\frac{1}{\sqrt{2}}(|\psi_0\rangle + |\psi_1\rangle) = H|\psi_0\rangle, \quad \frac{1}{\sqrt{2}}(|\psi_0\rangle - |\psi_1\rangle) = H|\psi_1\rangle$$

These states can also be expanded using Dirac notation as sums of  $|\psi_0\rangle$  and  $|\psi_1\rangle$ :

$$|\psi_+\rangle = \frac{1}{\sqrt{2}}(|\psi_0\rangle + |\psi_1\rangle), \quad |\psi_-\rangle = \frac{1}{\sqrt{2}}(|\psi_0\rangle - |\psi_1\rangle)$$

## Computational basis vectors

This demonstrates why these states are often called a *computational basis*: every quantum state can always be expressed as sums of computational basis vectors and such sums are easily expressed using Dirac notation. The converse is also true in that the states  $|\psi+\rangle$  and  $|\psi-\rangle$  also form a basis for quantum states. You can see this from the fact that

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|\psi+\rangle + |\psi-\rangle), \quad |\psi\rangle = \frac{1}{\sqrt{2}}(|\psi+\rangle - |\psi-\rangle).$$

As an example of Dirac notation, consider the bracket  $\langle 0 | 1 \rangle$ , which is the inner product between  $|0\rangle$  and  $|1\rangle$ . It can be written as

$$\langle 0 | 1 \rangle = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 0.$$

This says that  $|\psi\rangle$  and  $|\psi'\rangle$  are orthogonal vectors, meaning that  $\langle \psi | \psi' \rangle = \langle \psi' | \psi \rangle = 0$ . Also by definition  $\langle 0 | 0 \rangle = \langle 1 | 1 \rangle = 1$ , which means that the two computational basis vectors can also be called *orthonormal*.

These orthonormal properties will be useful in the following example. If we have a state  $|\psi\rangle = \frac{3}{5}|0\rangle + \frac{4}{5}|1\rangle$  then because  $\langle 1 | 0 \rangle = 0$  the probability of measuring  $|1\rangle$  is

$$|\langle 1 | \psi \rangle|^2 = \left| \frac{3}{5} \langle 0 | 1 \rangle + \frac{4}{5} \langle 1 | 1 \rangle \right|^2 = \frac{9}{25}.$$

## Tensor product notation

Dirac notation also includes an implicit tensor product structure within it. This is important because in quantum computing, the state vector described by two uncorrelated quantum registers is the tensor products of the two state vectors. Concisely describing the tensor product structure, or lack thereof, is vital if you want to explain a quantum computation. The tensor product structure implies that we can write  $|\psi\rangle \otimes |\phi\rangle$  for any two quantum state vectors  $|\phi\rangle$  and  $|\psi\rangle$  as  $|\psi\rangle \otimes |\phi\rangle = |\psi\rangle |\phi\rangle$ , however by convention writing  $\otimes$  in between the vectors is unnecessary, and we can simply write  $|\psi\rangle |\phi\rangle = |\psi\rangle \phi$ . To further information about vectors and tensor products, see [Vectors and Matrices in Quantum Computing](#). For example, the state with two qubits initialized to the zero state is given by

$$|\psi\rangle = |\psi\rangle \otimes |\psi\rangle = |\psi\rangle |\psi\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}.$$

Similarly, the state  $|\psi_p\rangle$  for integer  $p$  represents a quantum state that encodes in binary representation the integer  $p$ . For example, if we wish to express the number  $5$  using an unsigned binary encoding we could equally express it as

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_0\rangle \otimes |\psi_1\rangle = |\psi_1\rangle |\psi_0\rangle |\psi_1\rangle = |\psi_1\rangle |\psi\rangle.$$

Within this notation  $|\psi\rangle$  need not refer to a single-qubit state but rather a *qubit register* storing a binary encoding of  $0$ . The differences between these two notations is usually clear from the context. This convention is useful for simplifying the first example which can be written in any of the following ways:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \cdots \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |\psi\rangle \otimes \cdots \otimes |\psi\rangle = |\psi\rangle^{\otimes n}$$

where  $|\psi\rangle^{\otimes n}$  represents the tensor product of  $n$   $|\psi\rangle$  quantum states.

## Example: Describe superposition with Dirac notation

As another example of how you can use Dirac notation to describe a quantum state, consider the following equivalent ways of writing a quantum state that is an equal superposition over every possible bit string of length  $n$ :

$\$ \$ H^{\otimes n} |0\rangle = \frac{1}{2^{n/2}} \sum_{j=0}^{2^n-1} |j\rangle = |\psi+\rangle^{\otimes n}.$

Here you may wonder why the sum goes from  $0$  to  $2^n-1$  for  $n$  bits. First note that there are  $2^n$  different configurations that  $n$  bits can take. You can see this by noting that one bit can take  $2$  values but two bits can take  $4$  values and so forth. In general, this means that there are  $2^n$  different possible bit strings but the largest value encoded in any of them is  $1 \dots 1 = 2^n-1$  and hence it is the upper limit for the sum. As a side note, in this example we did not use  $|\psi+\rangle^{\otimes n} = |\psi+\rangle$  in analogy to  $|\psi\rangle^{\otimes n} = |\psi\rangle$  because this notational convention is usually reserved for the computational basis state with every qubit initialized to zero. While such a convention would be sensible in this case, it is not employed in the quantum computing literature.

## Express linearity with Dirac notation

Another nice feature of Dirac notation is the fact that it is linear. For example, for two complex numbers  $\alpha$  and  $\beta$ , we can write

$$\$ \$ |\psi\rangle \otimes (\alpha|\phi\rangle + \beta|\chi\rangle) = \alpha|\psi\rangle\langle\phi| + \beta|\psi\rangle\langle\chi|.$$

That is to say, you can distribute the tensor product notation in Dirac notation so that taking tensor products between state vectors ends up looking just like ordinary multiplication.

Bra vectors follow a similar convention to ket vectors. For example, the vector  $|\psi\rangle\langle\phi|$  is equivalent to the state vector  $|\psi\rangle^\dagger \otimes |\phi\rangle^\dagger = (\langle\psi| \otimes \langle\phi|)^\dagger$ . If the ket vector  $|\psi\rangle$  is  $\alpha|0\rangle + \beta|1\rangle$  then the bra vector version of the vector is  $\langle\psi| = \langle\psi|^\dagger = (\langle 0| \alpha^* + \langle 1| \beta^*)$ .

As an example, imagine that we wish to calculate the probability of measuring the state  $|\psi\rangle = \frac{1}{\sqrt{5}}(|1\rangle + \frac{1}{\sqrt{5}}|0\rangle)$  using a quantum program for measuring states to be either  $|\psi+\rangle$  or  $|\psi-\rangle$ . Then the probability that the device would output that the state is  $|\psi-\rangle$  is

$$\$ \$ |\langle\psi-|\psi\rangle|^2 = |\left(\frac{1}{\sqrt{5}}\langle 1| - \frac{1}{\sqrt{5}}\langle 0|\right)(\frac{1}{\sqrt{5}}|1\rangle + \frac{1}{\sqrt{5}}|0\rangle)|^2 = |\left(-\frac{1}{\sqrt{5}}\langle 1| + \frac{1}{\sqrt{5}}\langle 0|\right)(\frac{1}{\sqrt{5}}|1\rangle + \frac{1}{\sqrt{5}}|0\rangle)|^2 = \frac{1}{5}.$$

The fact that the negative sign appears in the calculation of the probability is a manifestation of quantum interference, which is one of the mechanisms by which quantum computing gains advantages over classical computing.

## ketbra or outer product

The final item worth discussing in Dirac notation is the *ketbra* or outer product. The outer product is represented within Dirac notations as  $|\psi\rangle\langle\phi|$ , and sometimes called ketbras because the bras and kets occur in the opposite order as brackets. The outer product is defined via matrix multiplication as  $|\psi\rangle\langle\phi| = \psi\phi^\dagger$  for quantum state vectors  $|\psi\rangle$  and  $|\phi\rangle$ . The simplest, and arguably most common example of this notation, is

$$\$ \$ |\psi\rangle\langle\phi| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ketbras are often called projectors because they project a quantum state onto a fixed value. Since these operations are not unitary (and do not even preserve the norm of a vector), it should come as no surprise that a quantum computer cannot deterministically apply a projector. However projectors do a beautiful job of describing the action that measurement has on a quantum state. For example, if we measure a state  $|\psi\rangle$  to be  $|0\rangle$  then the resulting transformation that the state experiences as a result of the measurement is

$$\$ \$ |\psi\rangle \rightarrow \frac{|\psi\rangle\langle\psi|}{\langle\psi|\psi\rangle} = |\psi\rangle.$$

as one expects if you were to measure the state and find it to be  $|\psi\rangle$ . To reiterate, such projectors cannot be applied on a state in a quantum computer deterministically. Instead, they can at best be applied randomly with the result  $|\psi\rangle$  appearing with some fixed probability. The probability of such a measurement succeeding can be written as the expectation value of the quantum projector in the state

$$\langle \psi | (\langle 0 | \psi \rangle) |\psi\rangle = |\langle \psi | 0 \rangle|^2,$$

which illustrates that projectors simply give a new way of expressing the measurement process.

If instead we consider measuring the first qubit of a multi-qubit state to be  $|1\rangle$  then we can also describe this process conveniently using projectors and Dirac notation:

$$P(\text{first qubit} = 1) = \langle \psi | \left( \langle 1 | \langle 1 | + \sum_{i=0}^{n-1} \langle i | \langle i | \right) |\psi\rangle.$$

Here the identity matrix can be conveniently written in Dirac notation as

$$\boldsymbol{\mathbb{I}} = |0\rangle\langle 0| + |1\rangle\langle 1| = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

For the case where there are two-qubits the projector can be expanded as

$$\langle \psi | (\langle 1 | \langle 1 | + \langle 0 | \langle 0 |) |\psi\rangle = \langle \psi | (|10\rangle\langle 10| + |11\rangle\langle 11|).$$

We can then see that this is consistent with the discussion about measurement likelihoods for multiqubit states using column-vector notation:

$$P(\text{first qubit} = 1) = |\psi^{\dagger}(e_{10}e_{10}^{\dagger} + e_{11}e_{11}^{\dagger})|^2 = |e_{10}^{\dagger}\psi|^2 + |e_{11}^{\dagger}\psi|^2,$$

which matches the multi-qubit measurement discussion. The generalization of this result to the multi-qubit case, however, is slightly more straightforward to express using Dirac notation than column-vector notation, and is entirely equivalent to the previous treatment.

## Density operators

Another useful operator to express using Dirac notation is a *density operator*, sometimes also known as a *state operator*. A density operator for a quantum state vector takes the form  $\rho = |\psi\rangle\langle\psi|$ . This concept of representing the state as a matrix, rather than a vector, is often convenient because it gives a convenient way of representing probability calculations, and also allows one to describe both statistical uncertainty as well as quantum uncertainty within the same formalism. General quantum state operators, rather than vectors, are ubiquitous in some areas of quantum computing but are not necessary to understand the basics of the field. For the interested reader, we recommend reading one of the reference books provided in [For more information](#).

## Q# gate sequences equivalent to quantum states

A final point worth raising about quantum notation and the Q# programming language: at the onset of this document we mentioned that the quantum state is the fundamental object of information in quantum computing. It may then come as a surprise that in Q# there is no notion of a quantum state. Instead, all states are described only by the operations used to prepare them. The previous example is an excellent illustration of this. Rather than expressing a uniform superposition over every quantum bit string in a register, we can represent the result as  $H^{\otimes n} |0\rangle$ . This exponentially shorter description of the state not only has the advantage that we can classically reason about it, but it also concisely defines the operations needed to be propagated through the software stack to implement the algorithm. For this reason, Q# is designed to emit gate sequences rather than quantum states; however, at a theoretical level the two perspectives are equivalent.

# Single- and multi-qubit Pauli measurement operations

3/5/2021 • 10 minutes to read • [Edit Online](#)

As you work with Q#, *Pauli measurements* are a common kind of measurement, which generalize computational basis measurements to include measurements in other bases and of parity between different qubits. In such cases, it is common to discuss measuring a Pauli operator, in general an operator such as `$X,Y,Z$` or `$Z\otimes X, X\otimes Y$`, and so forth.

Discussing measurement in terms of Pauli operators is especially common in the subfield of quantum error correction. In Q#, we follow a similar convention; we now explain this alternative view of measurements.

## TIP

In Q#, multi-qubit Pauli operators are generally represented by arrays of type `Pauli[]`. For example, to represent `$Z\otimes Z \otimes Y$`, you can use the array `[PauliX, PauliZ, PauliY]`.

Before delving into the details of how to think of a Pauli measurement, it is useful to think about what measuring a single qubit inside a quantum computer does to the quantum state. Imagine that we have an  $n$ -qubit quantum state; then measuring one qubit immediately rules out half of the  $2^n$  possibilities that state could be in. In other words, the measurement projects the quantum state onto one of two half-spaces. We can generalize the way we think about measurement to reflect this intuition.

In order to concisely identify these subspaces, we need a language for describing them. One way to describe the two subspaces is by specifying them through a matrix that just has two unique eigenvalues, taken by convention to be  $\pm 1$ . For a simple example of describing subspaces in this way, consider `$Z$`:

```
$$ \begin{aligned} Z &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \end{aligned} $$
```

By reading the diagonal elements of the Pauli-`$Z$` matrix, we can see that `$Z$` has two eigenvectors, `$\ket{0}$` and `$\ket{1}$`, with corresponding eigenvalues  $\pm 1$ . Thus, if we measure the qubit and obtain `Zero` (corresponding to the state `$\ket{0}$`), we know that the state of our qubit is a  $+1$  eigenstate of the `$Z$` operator. Similarly, if we obtain `One`, we know that the state of our qubit is a  $-1$  eigenstate of `$Z$`. This process is referred to in the language of Pauli measurements as "measuring Pauli `$Z$`," and is entirely equivalent to performing a computational basis measurement.

Any  $2 \times 2$  matrix that is a unitary transformation of `$Z$` also satisfies this criteria. That is, we could also use a matrix `$A = U^\dagger Z U$`, where `$U$` is any other unitary matrix, to give a matrix that defines the two outcomes of a measurement in its  $\pm 1$  eigenvectors. The notation of Pauli measurements references this unitary equivalence by identifying `$X,Y,Z$` measurements as equivalent measurements that one could do to gain information from a qubit. These measurements are given below for convenience.

PAULI MEASUREMENT	UNITARY TRANSFORMATION
<code>\$Z\$</code>	<code>\$\boldsymbol{\text{done}}\$</code>
<code>\$X\$</code>	<code>\$H\$</code>
<code>\$Y\$</code>	<code>\$HS^{\dagger}\$</code>

That is, using this language, "measure  $\$Y\$$ " is equivalent to applying  $\$HS^\dagger\$$  and then measuring in the computational basis, where  $S$  is an intrinsic quantum operation sometimes called the "phase gate," and can be simulated by the unitary matrix

$$\$ \$ \begin{aligned} S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}. \end{aligned} \$ \$$$

It is also equivalent to applying  $\$HS^\dagger\$$  to the quantum state vector and then measuring  $\$Z\$$ , such that the following operation is equivalent to `Measure([PauliY], [q])`:

```
operation MeasureY(qubit : Qubit) : Result {
    mutable result = Zero;
    within {
        Adjoint S(q);
        H(q);
    } apply {
        set result = M(q);
    }
    return result;
}
```

The correct state would then be found by transforming back to the computational basis, which amounts to applying  $\$SH\$$  to the quantum state vector; in the above snippet, the transformation back to the computational basis is handled automatically by the use of the `within ... apply` block.

In Q#, we say the outcome---that is, the classical information extracted from interacting with the state---is given by a `Result` value  $\$j \in \{\text{Zero}, \text{One}\}\$$  indicating if the result is in the  $\$(-1)^j\$$  eigenspace of the Pauli operator measured.

## Multiple-qubit measurements

Measurements of multi-qubit Pauli operators are defined similarly, as seen from:

$$\$ \$ Z \otimes Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \$ \$$$

Thus the tensor products of two Pauli- $\$Z\$$  operators forms a matrix composed of two spaces consisting of  $\$+1\$$  and  $\$-1\$$  eigenvalues. As with the single-qubit case, both constitute a half-space meaning that half of the accessible vector space belongs to the  $\$+1\$$  eigenspace and the remaining half to the  $\$-1\$$  eigenspace. In general, it is easy to see from the definition of the tensor product that any tensor product of Pauli- $\$Z\$$  operators and the identity also obeys this. For example,

$$\$ \$ \begin{aligned} Z \otimes \text{Id} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}. \end{aligned} \$ \$$$

As before, any unitary transformation of such matrices also describes two half-spaces labeled by  $\$|\pm 1\$$  eigenvalues. For example,  $\$X \otimes X = H \otimes H\$$  from the identity that  $\$Z=H\bar{H}\$$ . Similar to the one-qubit case, all two-qubit Pauli-measurements can be written as  $\$U^\dagger (Z \otimes \text{Id}) U\$$  for  $\$4 \times 4\$$  unitary matrices  $\$U\$$ . We enumerate the transformations in the following table.

### NOTE

In the table below, we use `\operatorname{SWAP}` to indicate the matrix  $\$ \$ \begin{aligned} \operatorname{SWAP} &= \\ &\left( \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \right) \end{aligned} \$ \$$  used to simulate the intrinsic operation `SWAP`.

PAULI MEASUREMENT	UNITARY TRANSFORMATION
$Z \otimes \text{boldone}$	$\text{boldone} \otimes \text{boldone}$
$X \otimes \text{boldone}$	$H \otimes \text{boldone}$
$Y \otimes \text{boldone}$	$HS^\dagger \otimes \text{boldone}$
$\text{boldone} \otimes Z$	$\text{operatorname{SWAP}}$
$\text{boldone} \otimes X$	$(H \otimes \text{boldone}) \text{operatorname{SWAP}}$
$\text{boldone} \otimes Y$	$(HS^\dagger \otimes \text{boldone}) \text{operatorname{SWAP}}$
$Z \otimes Z$	$\text{operatorname{CNOT}}_{\{10\}}$
$X \otimes Z$	$\text{operatorname{CNOT}}_{\{10\}}(H \otimes \text{boldone})$
$Y \otimes Z$	$\text{operatorname{CNOT}}_{\{10\}}(HS^\dagger \otimes \text{boldone})$
$Z \otimes X$	$\text{operatorname{CNOT}}_{\{10\}}(\text{boldone} \otimes H)$
$X \otimes X$	$\text{operatorname{CNOT}}_{\{10\}}(H \otimes H)$
$Y \otimes X$	$\text{operatorname{CNOT}}_{\{10\}}(HS^\dagger \otimes H)$
$Z \otimes Y$	$\text{operatorname{CNOT}}_{\{10\}}(\text{boldone} \otimes HS^\dagger)$
$X \otimes Y$	$\text{operatorname{CNOT}}_{\{10\}}(H \otimes HS^\dagger)$
$Y \otimes Y$	$\text{operatorname{CNOT}}_{\{10\}}(HS^\dagger \otimes HS^\dagger)$

Here, the `CNOT` operation appears for the following reason. Each Pauli measurement that does not include the  $\text{boldone}$  matrix is equivalent up to a unitary to  $Z \otimes Z$  by the above reasoning. The eigenvalues of  $Z \otimes Z$  only depend on the parity of the qubits that comprise each computational basis vector, and the controlled-not operations serve to compute this parity and store it in the first bit. Then once the first bit is measured, we can recover the identity of the resultant half-space, which is equivalent to measuring the Pauli operator.

One additional note: while it may be tempting to assume that measuring  $Z \otimes Z$  is the same as sequentially measuring  $Z \otimes \mathbb{1}$  and then  $\mathbb{1} \otimes Z$ , this assumption would be false. The reason is that measuring  $Z \otimes Z$  projects the quantum state into either the  $+1$  or  $-1$  eigenstate of these operators. Measuring  $Z \otimes \mathbb{1}$  and then  $\mathbb{1} \otimes Z$  projects the quantum state vector first onto a half space of  $Z \otimes \mathbb{1}$  and then onto a half space of  $\mathbb{1} \otimes Z$ . As there are four computational basis vectors, performing both measurements reduces the state to a quarter-space and hence reduces it to a single computational basis vector.

## Correlations between qubits

Another way of looking at measuring tensor products of Pauli matrices such as  $X \otimes X$  or  $Z \otimes Z$  is that these measurements let you look at information stored in the correlations between the two qubits.

Measuring  $\otimes \text{id}$  lets you look at information that is locally stored in the first qubit. While both types of measurements are equally valuable in quantum computing, the former illuminates the power of quantum computing. It reveals that in quantum computing, often the information you wish to learn is not stored in any single qubit but rather stored non-locally in all the qubits at once, and therefore only by looking at it through a joint measurement (e.g.  $\otimes Z$ ) does this information become manifest.

For example, in error correction, we often wish to learn what error occurred while learning nothing about the state that we're trying to protect. The [bit-flip code sample](#) shows an example of how you can do that using measurements like  $\otimes Z$  and  $\otimes \text{id}$ .

Arbitrary Pauli operators such as  $\otimes Y$  can also be measured. All such tensor products of Pauli operators have only two eigenvalues  $\pm 1$  and both eigenspaces constitute half-spaces of the entire vector space. Thus they coincide with the requirements stated above.

In Q#, such measurements return  $j$  if the measurement yields a result in the eigenspace of sign  $(-1)^j$ . Having Pauli measurements as a built-in feature in Q# is helpful because measuring such operators requires long chains of controlled-NOT gates and basis transformations to describe the diagonalizing  $U$  gate needed to express the operation as a tensor product of  $Z$  and  $\text{id}$ . By being able to specify that you wish to do one of these pre-defined measurements, you don't need to worry about how to transform your basis such that a computational basis measurement provides the necessary information. Q# handles all the necessary basis transformations for you automatically. For more information, see the [Measure](#) and [MeasurePaulis](#) operations.

## The No-Cloning Theorem

Quantum information is powerful. It enables us to do amazing things such as factor numbers exponentially faster than the best known classical algorithms, or efficiently simulate correlated electron systems that classically require exponential cost to simulate accurately. However, there are limitations to the power of quantum computing. One such limitation is given by the *No-Cloning Theorem*.

The No-Cloning Theorem is aptly named. It disallows cloning of generic quantum states by a quantum computer. The proof of the theorem is remarkably straightforward. While a full proof of the no-cloning theorem is a little too technical for our discussion here, the proof in the case of no additional auxiliary qubits is within our scope.

For such a quantum computer, the cloning operation must be described by a unitary matrix. We disallow measurement, since it would corrupt the quantum state we are trying to clone. To simulate the cloning operation, we want the unitary matrix used to have the property that  $U |\psi\rangle = |\psi\rangle + |\psi\rangle$  for any state  $|\psi\rangle$ . The linearity property of matrix multiplication then implies that for any second quantum state  $|\phi\rangle$ ,

$$\begin{aligned} U &= \frac{1}{\sqrt{2}} (|0\rangle\langle\psi| + |1\rangle\langle\psi|) \\ U^2 &= \frac{1}{2} (|0\rangle\langle\psi| + |1\rangle\langle\psi|)(|0\rangle\langle\psi| + |1\rangle\langle\psi|) \\ &= \frac{1}{2} (|0\rangle\langle\psi| + |1\rangle\langle\psi| + |0\rangle\langle\psi| + |1\rangle\langle\psi|) \\ &= |0\rangle\langle\psi| + |1\rangle\langle\psi| \end{aligned}$$

This provides the fundamental intuition behind the No-Cloning Theorem: any device that copies an unknown quantum state must induce errors on at least some of the states it copies. While the key assumption that the cloner acts linearly on the input state can be violated through the addition and measurement of auxiliary qubits, such interactions also leak information about the system through the measurement statistics and prevent exact cloning in such cases as well. For a more complete proof of the No-Cloning Theorem see [For more information](#).

The No-Cloning Theorem is important for qualitative understanding of quantum computing because if you could clone quantum states inexpensively then you would be granted a near-magical ability to learn from quantum states. Indeed, you could violate Heisenberg's vaunted uncertainty principle. Alternatively, you could use an optimal cloner to take a single sample from a complex quantum distribution and learn everything you could possibly learn about that distribution from just one sample. This would be like you flipping a coin and

observing heads and then upon telling a friend about the result having them respond "Ah the distribution of that coin must be Bernoulli with  $p=0.512643\ldots$ !" Such a statement would be non-sensical because one bit of information (the heads outcome) simply cannot provide the many bits of information needed to encode the distribution without substantial prior information. Similarly, without prior information we cannot perfectly clone a quantum state just as we cannot prepare an ensemble of such coins without knowing  $p$ .

Information is not free in quantum computing. Each qubit measured gives a single bit of information, and the No-Cloning Theorem shows that there is no backdoor that can be exploited to get around the fundamental tradeoff between information gained about the system and the disturbance invoked upon it.

# Quantum circuit diagrams

3/5/2021 • 4 minutes to read • [Edit Online](#)

This article covers conventions for quantum circuit diagrams. Quantum operations are easier to understand in a diagram than in the equivalent written matrix once you understand the visual conventions.

## Example: Unitary transformation

Consider for a moment the unitary transformation  $\text{CNOT}_{01}(H \otimes 1)$ . This gate sequence is of fundamental significance to quantum computing because it creates a maximally entangled two-qubit state:

$$\text{CNOT}_{01}(H \otimes 1) |00\rangle = \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle)$$

Operations with this or greater complexity are ubiquitous in quantum algorithms and quantum error correction. A quantum circuit diagram is a convenient tool for illustrating the operations.

The circuit diagram for preparing this maximally entangled quantum state is:

$$\text{CNOT}_{01}(H \otimes 1) = \begin{array}{c} \square \text{---} \\ | \qquad \oplus \\ \text{---} \end{array}$$

## Quantum circuit diagram conventions

In a circuit diagram, each solid line depicts a qubit or more generally a qubit register. By convention, the top line is qubit register  $|0\rangle$  and the remainder are labeled sequentially. The above example circuit is depicted as acting on two qubits (or equivalently two registers consisting of one qubit). Gates acting on one or more qubit registers are denoted as a box. For example, the symbol

$$\square$$

is a [Hadamard](#) operation acting on a single-qubit register.

Quantum gates are ordered in chronological order with the left-most gate as the gate first applied to the qubits. In other words, if you picture the wires as holding the quantum state, the wires bring the quantum state through each of the gates in the diagram from left to right. That is to say

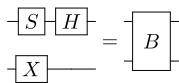
$$\square \square \square$$

is the unitary matrix  $CBA$ . Matrix multiplication obeys the opposite convention: the right-most matrix is applied first. In quantum circuit diagrams, however, the left-most gate is applied first. This difference can at times lead to confusion, so it is important to note this significant difference between the linear algebraic notation and quantum circuit diagrams.

## Inputs and outputs of quantum circuits

All previous examples given have had precisely the same number of wires (qubits) input to a quantum gate as the number of wires out from the quantum gate. It may at first seem reasonable that quantum circuits could have more, or fewer, outputs than inputs in general. This is impossible, however, because all quantum operations, save measurement, are unitary and hence reversible. If they did not have the same number of outputs as inputs they would not be reversible and hence not unitary, which is a contradiction. For this reason any box drawn in a circuit diagram must have precisely the same number of wires entering it as exiting it.

Multi-qubit circuit diagrams follow similar conventions to single-qubit ones. As a clarifying example, we can define a two-qubit unitary operation  $B$  to be  $(H \otimes X)$  and express the circuit equivalently as



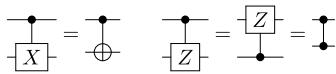
We can also view  $\$B\$$  as having an action on a single two-qubit register rather than two one-qubit registers depending on the context in which the circuit is used. Perhaps the most useful property of such abstract circuit diagrams is that they allow complicated quantum algorithms to be described at a high level without having to compile them down to fundamental gates. This means that you can get an intuition about the data flow for a large quantum algorithm without needing to understand all the details of how each of the subroutines within the algorithm work.

## Controlled gates

The other construct that is built into multi-qubit quantum circuit diagrams is control. The action of a quantum singly controlled gate, denoted  $\$|\Lambda(G)|$ , where a single qubit's value controls the application of  $\$G\$$ , can be understood by looking at the following example of a product state input  $\$|\Lambda(G)| (\alpha |0\rangle + \beta |1\rangle) |\psi\rangle = \alpha |0\rangle |\psi\rangle + \beta |1\rangle G|\psi\rangle$ . That is to say, the controlled gate applies  $\$G\$$  to the register containing  $\$|\psi|\$$  if and only if the control qubit takes the value  $\$1\$$ . In general, we describe such controlled operations in circuit diagrams as



Here the black circle denotes the quantum bit on which the gate is controlled and a vertical wire denotes the unitary that is applied when the control qubit takes the value  $\$1\$$ . For the special cases where  $\$G=X\$$  and  $\$G=Z\$$  we introduce the following notation to describe the controlled version of the gates (note that the controlled-X gate is the [\\$CNOT\\$ gate](#)):



Q# provides methods to automatically generate the controlled version of an operation, which saves the programmer from having to hand code these operations. An example of this is shown below:

```
operation PrepareSuperposition(qubit : Qubit) : Unit
is Ctl { // Auto-generate the controlled specialization of the operation
    H(qubit);
}
```

## Measurement operator

The remaining operation to visualize in circuit diagrams is measurement. Measurement takes a qubit register, measures it, and outputs the result as classical information. A measurement operation is denoted by a meter symbol and always takes as input a qubit register (denoted by a solid line) and outputs classical information (denoted by a double line). Specifically, such a subcircuit looks like:



Q# implements a [Measure operator](#) for this purpose. See the [section on measurements](#) for more information.

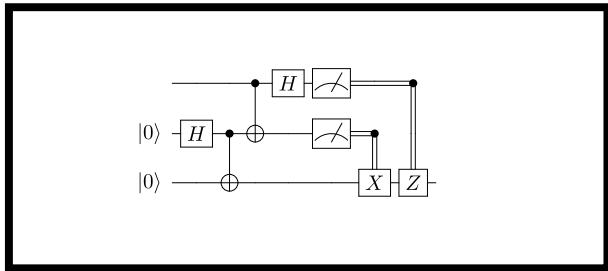
Similarly, the subcircuit



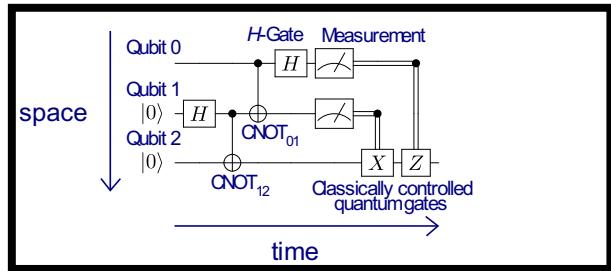
gives a classically controlled gate, where  $\$G\$$  is applied conditioned on the classical control bit being value  $\$1\$$ .

## Teleportation circuit diagram

Quantum teleportation is perhaps the best quantum algorithm for illustrating these components. You can learn hands-on with the corresponding [Quantum Kata](#) Quantum teleportation is a method for moving data within a quantum computer (or even between distant quantum computers in a quantum network) through the use of entanglement and measurement. Interestingly, it is actually capable of moving a quantum state, say the value in a given qubit, from one qubit to another, without even knowing what the qubit's value is! This is necessary for the protocol to work according to the laws of quantum mechanics. The quantum teleportation circuit is given below; we also provide an annotated version of the circuit to illustrate how to read the quantum circuit.



Teleportation Circuit



Annotated Teleportation Circuit

# Work with and define quantum oracles

4/6/2021 • 3 minutes to read • [Edit Online](#)

An oracle  $O$  is a "black box" operation that is used as input to another algorithm. Often, such operations are defined using a classical function  $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$  which takes an  $n$ -bit binary input and produces an  $m$ -bit binary output. To do so, consider a particular binary input  $x = (x_0, x_1, \dots, x_{n-1})$ . We can label qubit states as  $\ket{\vec{x}} = \ket{x_0} \otimes \ket{x_1} \otimes \dots \otimes \ket{x_{n-1}}$ .

We may first attempt to define  $O$  so that  $O\ket{x} = \ket{f(x)}$ , but this has a couple problems. First,  $f$  may have a different size of input and output ( $n \neq m$ ), such that applying  $O$  would change the number of qubits in the register. Second, even if  $n = m$ , the function may not be invertible: if  $f(x) = f(y)$  for some  $x \neq y$ , then  $O\ket{x} = O\ket{y}$  but  $O^\dagger\ket{x} \neq O^\dagger\ket{y}$ . This means we won't be able to construct the adjoint operation  $O^\dagger$ , and oracles have to have an adjoint defined for them.

## Define an oracle by its effect on computational basis states

We can deal with both of these problems by introducing a second register of  $m$  qubits to hold our answer. Then we will define the effect of the oracle on all computational basis states: for all  $x \in \{0, 1\}^n$  and  $y \in \{0, 1\}^m$ ,

$$O(\ket{x} \otimes \ket{y}) = \ket{x} \otimes \ket{y \oplus f(x)}.$$

Now  $O = O^\dagger$  by construction, thus we have resolved both of the earlier problems.

### TIP

To see that  $O = O^\dagger$ , note that  $O^2 = \text{Id}$  since  $a \oplus b \oplus b = a$  for all  $a, b \in \{0, 1\}$ . As a result,  $O\ket{x}\ket{y} = \ket{x}\ket{y \oplus f(x)} = \ket{x}\ket{y \oplus f(x) \oplus f(x)} = \ket{x}\ket{y}$ .

Importantly, defining an oracle this way for each computational basis state  $\ket{x}\ket{y}$  also defines how  $O$  acts for any other state. This follows immediately from the fact that  $O$ , like all quantum operations, is linear in the state that it acts on. Consider the Hadamard operation, for instance, which is defined by  $H\ket{0} = \ket{+}$  and  $H\ket{1} = \ket{-}$ . If we wish to know how  $H$  acts on  $\ket{+}$ , we can use that  $H$  is linear,

$$H\ket{+} = \frac{1}{\sqrt{2}}(H\ket{0} + H\ket{1}) = \frac{1}{\sqrt{2}}(H\ket{0} + H\ket{1}) \\ = \frac{1}{\sqrt{2}}(\ket{+} + \ket{-}) = \frac{1}{\sqrt{2}}(\ket{0} + \ket{1} + \ket{0} - \ket{1}) = \ket{0}.$$

In the case of defining our oracle  $O$ , we can similarly use that any state  $\ket{\psi}$  on  $n + m$  qubits can be written as

$$\ket{\psi} = \sum_{x \in \{0, 1\}^n, y \in \{0, 1\}^m} \alpha(x, y) \ket{x}\ket{y}$$

where  $\alpha : \{0, 1\}^n \times \{0, 1\}^m \rightarrow \mathbb{C}$  represents the coefficients of the state  $\ket{\psi}$ . Thus,

$$O\ket{\psi} = O\sum_{x \in \{0, 1\}^n, y \in \{0, 1\}^m} \alpha(x, y) \ket{x}\ket{y} \\ = \sum_{x \in \{0, 1\}^n, y \in \{0, 1\}^m} \alpha(x, y) O\ket{x}\ket{y} \\ = \sum_{x \in \{0, 1\}^n, y \in \{0, 1\}^m} \alpha(x, y) \ket{x}\ket{y \oplus f(x)}.$$

## Phase oracles

Alternatively, we can encode  $f$  into an oracle  $O$  by applying a *phase* based on the input to  $O$ . For instance, we might define  $O$  such that  $\begin{aligned} O \ket{x} = (-1)^{f(x)} \ket{x}. \end{aligned}$  If a phase oracle acts on a register initially in a computational basis state  $\ket{x}$ , then this phase is a global phase and hence not observable. But such an oracle can be a very powerful resource if applied to a superposition or as a controlled operation. For example, consider a phase oracle  $O_f$  for a single-qubit function  $f$ . Then,

$$\begin{aligned} O_f \ket{+} &= O_f (\ket{0} + \ket{1}) / \sqrt{2} \\ &= ((-1)^{f(0)} \ket{0} + (-1)^{f(1)} \ket{1}) / \sqrt{2} \\ &= (-1)^{f(0)} Z^{f(0)-f(1)} \ket{+}. \end{aligned}$$

#### NOTE

Note that  $Z^{-1} = Z^\dagger = Z$  and therefore  $Z^{f(0)-f(1)} = Z^{f(1)-f(0)}$ .

More generally, both views of oracles can be broadened to represent classical functions which return real numbers instead of only a single bit.

Choosing the best way to implement an oracle depends heavily on how this oracle will be used within a given algorithm. For example, [Deutsch-Jozsa algorithm](#) relies on the oracle implemented in the first way, while [Grover's algorithm](#) relies on the oracle implemented in the second way.

For more details, we suggest the discussion in [Gilyén et al. 1711.00465](#).

# Theory of Grover's search algorithm

5/27/2021 • 7 minutes to read • [Edit Online](#)

In this article you'll find a detailed theoretical explanation of the mathematical principles that make Grover's algorithm work.

For a practical implementation of Grover's algorithm to solve mathematical problems you can read our [guide to implement Grover's search algorithm](#).

## Statement of the problem

Any search task can be expressed with an abstract function  $f(x)$  that accepts search items  $x$ . If the item  $x$  is a solution to the search task, then  $f(x)=1$ . If the item  $x$  isn't a solution, then  $f(x)=0$ . The *search problem* consists of finding any item  $x_0$  such that  $f(x_0)=1$ .

The task that Grover's algorithm aims to solve can be expressed as follows: given a classical function  $f(x):\{0,1\}^n \rightarrow \{0,1\}$ , where  $n$  is the bit-size of the search space, find an input  $x_0$  for which  $f(x_0)=1$ . The complexity of the algorithm is measured by the number of uses of the function  $f(x)$ . Classically, in the worst-case scenario, we have to evaluate  $f(x)$  a total of  $N-1$  times, where  $N=2^n$ , trying out all the possibilities. After  $N-1$  elements, we know it must be the last element. We will see that Grover's quantum algorithm can solve this problem much faster, providing a quadratic speed up. Quadratic here implies that only about  $\sqrt{N}$  evaluations would be required, compared to  $N$ .

## Outline of the algorithm

Suppose we have  $N=2^n$  eligible items for the search task and we index them by assigning each item an integer from  $0$  to  $N-1$ . Further, suppose we know that there are  $M$  different valid inputs, meaning that there are  $M$  inputs for which  $f(x)=1$ . The steps of the algorithm are then as follows:

1. Start with a register of  $n$  qubits initialized in the state  $|\text{ket}{0}\rangle$ .
2. Prepare the register into a uniform superposition by applying  $H$  to each qubit of the register:  
$$|\text{register}\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$$
3. Apply the following operations to the register  $N_{\text{optimal}}$  times:
  - a. The phase oracle  $O_f$  that applies a conditional phase shift of  $-1$  for the solution items.
  - b. Apply  $H$  to each qubit in the register.
  - c. A conditional phase shift of  $-1$  to every computational basis state except  $|\text{ket}{0}\rangle$ . This can be represented by the unitary operation  $O_0$ , as  $O_0$  represents the conditional phase shift to  $|\text{ket}{0}\rangle$  only.
  - d. Apply  $H$  to each qubit in the register.
4. Measure the register to obtain the index of a item that's a solution with very high probability.
5. Check if it's a valid solution. If not, start again.

$N_{\text{optimal}} = \left\lfloor \frac{\pi}{4} \sqrt{\frac{N}{M}} - \frac{1}{2} \right\rfloor$  is the optimal number of iterations that maximizes the likelihood of obtaining the correct item by measuring the register.

### NOTE

The joint application of the steps 3.b, 3.c and 3.d is usually known in the literature as Grover's diffusion operator.

The overall unitary operation applied to the register is:

$$\$(-H^{\otimes n}O_0H^{\otimes n}O_f)^{\otimes N_{\text{optimal}}}H^{\otimes n}$$

## Following the register's state step by step

To illustrate the process, let's follow the mathematical transformations of the state of the register for a simple case in which we have only two qubits and the valid element is  $|\ket{01}|$ .

1. We start with the register in the state:  $|\text{register}\rangle = |00\rangle$
2. After applying  $H$  to each qubit the register's state transforms to:  $|\text{register}\rangle = \frac{1}{\sqrt{4}} \sum_{i \in \{0,1\}^2} |i\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$
3. Then we apply the phase oracle to get:  $|\text{register}\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle + |11\rangle)$
4. Then  $H$  acts on each qubit again to give:  $|\text{register}\rangle = \frac{1}{2}(|00\rangle + |01\rangle - |10\rangle + |11\rangle)$
5. Now we apply the conditional phase shift on every state except  $|00\rangle$ :  $|\text{register}\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$
6. Now we end the first Grover iteration by applying  $H$  again to get:  $|\text{register}\rangle = |\ket{01}|$

We found the valid item in a single iteration. As we will see later, this is because for  $N=4$  and a single valid item,  $N_{\text{optimal}}=1$ .

## Geometrical explanation

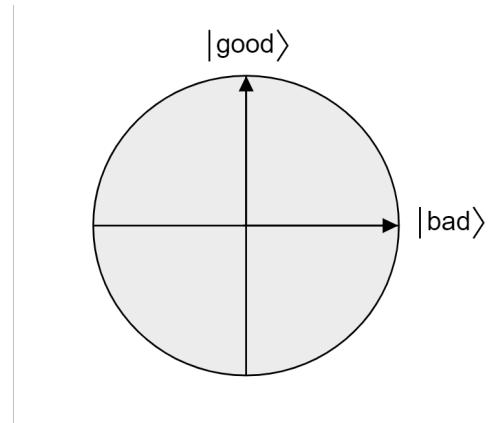
To see why Grover's algorithm works, let's study the algorithm from a geometrical perspective. Let  $|\ket{\text{bad}}|$  be the superposition of all states that aren't a solution to the search problem. Supposing there are  $M$  valid solutions, we get:

$$|\ket{\text{bad}}| = \frac{1}{\sqrt{N-M}} \sum_{x:f(x)=0} |\ket{x}|$$

We define the state  $|\ket{\text{good}}|$  as the superposition of all states that *are* a solution to the search problem:

$$|\ket{\text{good}}| = \frac{1}{\sqrt{M}} \sum_{x:f(x)=1} |\ket{x}|$$

Since *good* and *bad* are mutually exclusive sets because an item cannot be valid and not valid, the states  $|\ket{\text{good}}|$  and  $|\ket{\text{bad}}|$  are orthogonal. Both states form the orthogonal basis of a plane in the vector space. We can use this plane to visualize the algorithm.



Now, suppose  $|\psi\rangle$  is an arbitrary state that lives in the plane spanned by  $|\ket{\text{good}}|$  and  $|\ket{\text{bad}}|$ . Any state living in that plane can be expressed as:

$$|\psi\rangle = \alpha |\text{good}\rangle + \beta |\text{bad}\rangle$$

where  $\alpha$  and  $\beta$  are real numbers. Now, let's introduce the reflection operator  $R_{|\psi\rangle}$ , where  $|\psi\rangle$  is any qubit state living in the plane. The operator is defined as:

$$R_{|\psi\rangle} = 2|\psi\rangle\langle\psi| - I$$

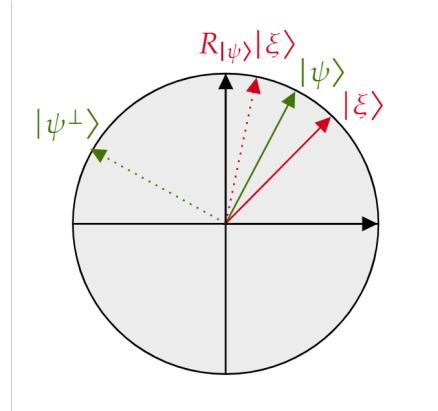
It is called the reflection operator about  $|\psi\rangle$  because it can be geometrically interpreted as reflection about the direction of  $|\psi\rangle$ . To see it, take the orthogonal basis of the plane formed by  $|\psi\rangle$  and its orthogonal complement  $|\psi^\perp\rangle$ . Any state  $|\xi\rangle$  of the plane can be decomposed in such basis:

$$|\xi\rangle = \mu |\psi\rangle + \nu |\psi^\perp\rangle$$

If we apply the operator  $R_{|\psi\rangle}$  to  $|\xi\rangle$  we get:

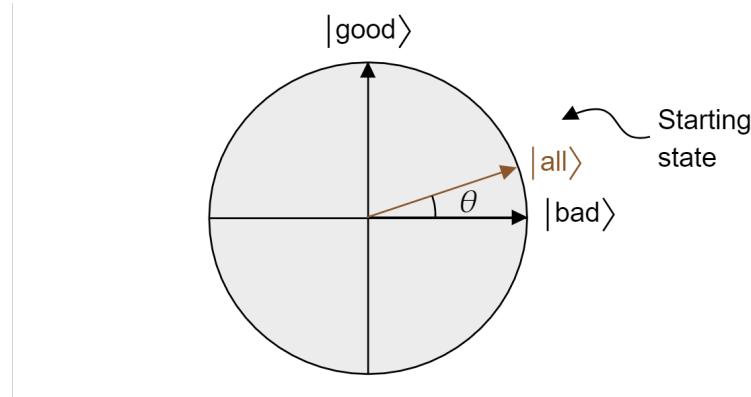
$$R_{|\psi\rangle}|\xi\rangle = \mu R_{|\psi\rangle}|\psi\rangle - \nu R_{|\psi\rangle}|\psi^\perp\rangle$$

The operator  $R_{|\psi\rangle}$  inverts the component orthogonal to  $|\psi\rangle$  but leaves the  $|\psi\rangle$  component unchanged. Therefore,  $R_{|\psi\rangle}$  is a reflection about  $|\psi\rangle$ .



In Grover's algorithm, after the first application of  $H$  to every qubit, we start with an uniform superposition of all states. This can be written as:

$$|\text{all}\rangle = \sqrt{\frac{M}{N}}|\text{good}\rangle + \sqrt{\frac{N-M}{N}}|\text{bad}\rangle$$



And thus the state lives in the plane. Note that the probability of obtaining a correct result when measuring from the equal superposition is just  $|\langle \text{good} | \text{all} \rangle|^2 = M/N$ , that is what we expect from a random guess.

The oracle  $O_f$  adds a negative phase to any solution to the search problem. Therefore, it can be written as a reflection about the  $|\text{bad}\rangle$  axis:

$$O_f = R_{|\text{bad}\rangle} = 2|\text{bad}\rangle\langle\text{bad}| - I$$

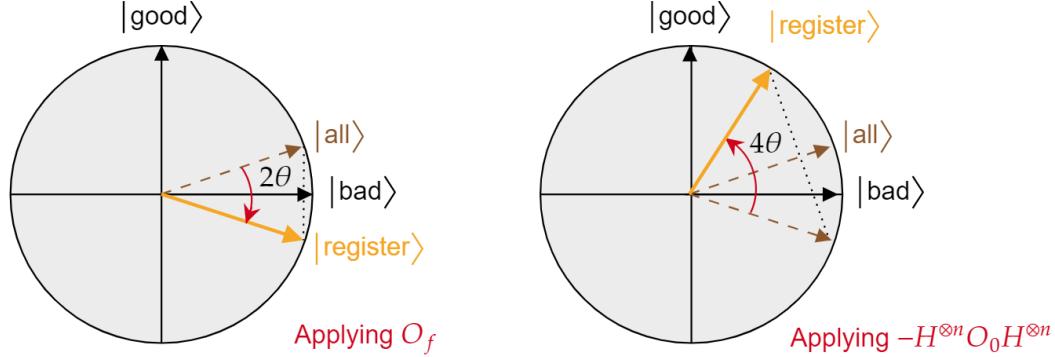
Analogously, the conditional phase shift  $O_0$  is just an inverted reflection about the state  $|\text{ket}0\rangle$ :

$$\$\$O_0 = R_{\{\ket{0}\}} = -2\ket{0}\bra{0} + \mathcal{I} \$\$$$

Knowing this fact, it's easy to check that the Grover diffusion operation  $\text{-H}^{\otimes n} O_0 H^{\otimes n}$  is also a reflection about the state  $\ket{\text{all}}$ . Just do:

$$\$\$-\text{H}^{\otimes n} O_0 H^{\otimes n}=2\text{H}^{\otimes n}\ket{0}\bra{0}\text{H}^{\otimes n}-\text{H}^{\otimes n}\mathcal{I}\text{H}^{\otimes n}=2\ket{\text{all}}\bra{\text{all}}-\mathcal{I}=R_{\{\ket{\text{all}}\}} \$\$$$

We just demonstrated that each iteration of Grover's algorithm is a composition of two reflections  $R_{\{\ket{\text{bad}}\}}$  and  $R_{\{\ket{\text{all}}\}}$ .



The combined effect of each Grover iteration is a counterclockwise rotation of an angle  $2\theta$ . Fortunately, the angle  $\theta$  is easy to find. Since  $\theta$  is just the angle between  $\ket{\text{all}}$  and  $\ket{\text{bad}}$ , we can use the scalar product to find the angle. We know that  $\cos(\theta) = \text{braket}{\text{all}|\text{bad}}$ , so we need to calculate  $\text{braket}{\text{all}|\text{bad}}$ . From the decomposition of  $\ket{\text{all}}$  in terms of  $\ket{\text{bad}}$  and  $\ket{\text{good}}$ , we get that:

$$\$\$ \theta = \arccos(\left| \text{braket}{\text{all}|\text{bad}} \right|) = \arccos(\sqrt{\frac{N-M}{N}}) \$\$$$

The angle between the state of the register and the  $\ket{\text{good}}$  state will decrease with each iteration, resulting in a higher probability of measuring a valid result. To calculate this probability we just need to calculate  $|\text{braket}{\text{good}|\text{register}}|^2$ . Denoting the angle between  $\ket{\text{good}}$  and  $\ket{\text{register}}$  as  $\gamma(k)$ , where  $k$  is the iteration count, we get:

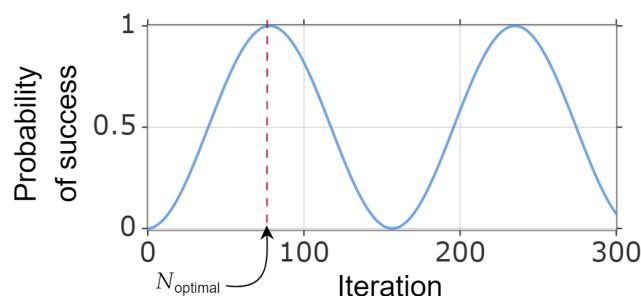
$$\$\$ \gamma(k) = \frac{\pi}{2} - \theta - k\theta = \frac{\pi}{2} - (2k+1)\theta \$\$$$

Therefore, the probability of success is:

$$\$\$ P(\text{success}) = \cos^2(\gamma(k)) = \sin^2(\left( (2k+1) \arccos(\sqrt{\frac{N-M}{N}}) \right)) \$\$$$

## Optimal number of iterations

As the probability of success can be written as a function of the number of iterations, we can find the optimal number of iterations  $N_{\text{optimal}}$  by computing the smallest positive integer that (approximately) maximizes the success probability function.



We know that  $\sin^2(x)$  reaches its first maximum for  $x=\frac{\pi}{2}$ , so we just need to make:

$$\$\$ \frac{\pi}{2} = (2k_{\text{optimal}} + 1) \arccos(\sqrt{\frac{N-M}{N}}) \$\$$$

This gives us:

$$k_{\text{optimal}} = \frac{\pi}{4} \arccos(\sqrt{1 - M/N}) - 1/2 = \frac{\pi}{4} \sqrt{\frac{N}{M}} - \frac{1}{2} - O(\sqrt{\frac{M}{N}})$$

Where in the last step we used the fact that  $\arccos(\sqrt{1-x}) = \sqrt{x} + O(x^{3/2})$ .

Therefore we can pick  $N_{\text{optimal}}$  to be  $N_{\text{optimal}} = \lfloor \lfloor \frac{\pi}{4} \sqrt{\frac{N}{M}} - \frac{1}{2} \rfloor \rfloor$ .

## Complexity analysis

From the previous analysis, we know that we need  $O(\sqrt{\frac{N}{M}})$  queries of the oracle  $O_f$  to find a valid item. However, can the algorithm be implemented efficiently in terms of time complexity?  $O_0$  is based on computing Boolean operations on  $n$  bits and is known to be implementable using  $O(n)$  gates. We also have two layers of  $n$  Hadamard gates. Both of these components thus require only  $O(n)$  gates per iteration. Because  $N=2^n$ , we have that  $O(n)=O(\log(N))$ . Therefore, if we need  $O(\sqrt{\frac{N}{M}}) \log(N)$  iterations and we need  $O(\log(N))$  gates per iteration, the total time complexity (without taking into account the oracle implementation) is  $O(\sqrt{\frac{N}{M}} \log(N))$ .

The overall complexity of the algorithm will ultimately depend on the complexity of the implementation of the oracle  $O_f$ . If a function evaluation is much more complicated on a quantum computer than on a classical one, the overall algorithm runtime will be longer in the quantum case, even though technically, it will use fewer queries.

## References

If you want to continue learning about Grover's algorithm, you can check any of the following sources:

- [Original paper by Lov K. Grover](#)
- Quantum search algorithms section of Nielsen, M. A. & Chuang, I. L. (2010). Quantum Computation and Quantum Information.
- [Grover's algorithm on Arxiv.org](#)

# Contributing to the Quantum Development Kit (QDK)

4/5/2021 • 3 minutes to read • [Edit Online](#)

The Quantum Development Kit is more than a collection of tools for writing quantum programs. It's part of a broad community of people discovering quantum computing, performing research in quantum algorithms, developing new applications for quantum devices, and otherwise working to make the most out of the quantum programming. As a member of that community, the Quantum Development Kit aims to offer quantum developers across a wide range of backgrounds with the features they need. Your contributions to the Quantum Development Kit help in realizing that goal by improving the tools used by other quantum developers, how those tools are documented, and even by creating new features and functionality that helps make the entire quantum programming community a better place to discover and create. We're very thankful for your kind contributions, and for the opportunity to work with you to make our community the best that it can be.

In this guide, we provide some advice on how to make your contribution as useful as possible to the broader quantum programming community.

## Building community

The very first thing about making a contribution is to always keep in mind the community that you are contributing to. By acting respectfully and professionally towards your peers in the quantum programming community and more broadly, you can help to make sure that your efforts build the best and most welcoming community possible.

As a part of that effort, all Quantum Development Kit projects have adopted the [Microsoft Open Source Code of Conduct](#). For more information, please see the [Code of Conduct FAQ](#) or contact [opencode@microsoft.com](mailto:opencode@microsoft.com) with any additional questions or comments.

## What kinds of contributions help the community?

There are lots of different ways to help the quantum programming community through your contributions. In this guide, we'll focus on three ways that are especially relevant to the Quantum Development Kit. All of these ways are critical to building a quantum community that empowers people. That said, this is definitely not an exhaustive list — we encourage you to explore other ways to help the community build on the promise of quantum programming!

- **Reporting bugs.** The first step in fixing bugs and other kinds of problems is to identify them. If you've found a bug in the Quantum Development Kit, letting us know helps us fix it and make a better set of tools for the quantum programming community.
- **Improving documentation.** Any documentation set can always be better, can cover more details, be made more accessible.
- **Contributing code.** Of course, one of the most direct ways to contribute is by adding new code to the Quantum Development Kit.

These different kinds of contributions are all immensely valuable, and are greatly appreciated. In the rest of the guide, we'll offer advice on how to make each kind of contribution.

## Where do contributions go?

The Quantum Development Kit includes a number of different pieces that all work together to realize a platform

for writing quantum programs. Each of these different pieces finds its home on a different repository, so the one of the earlier things to sort out is where each contribution best belongs.

- [microsoft/Quantum](#): Samples and tools to help get started with the Quantum Development Kit.
- [microsoft/QuantumLibraries](#): Standard and domain-specific libraries for the Quantum Development Kit.
- [microsoft/QuantumKatas](#): Self-paced programming exercises for learning quantum computing and the Q# programming language.
- [microsoft/qsharp-compiler](#): The Q# compiler, Visual Studio extension, and Visual Studio Code extension.
- [microsoft/qsharp-runtime](#): Simulation framework, code generation, and simulation target machines for the Quantum Development Kit.
- [microsoft/iqsharp](#): Jupyter kernel and Python host functionality for Q#, as well as Docker images for using IQ# in cloud environments.
- [microsoft/qsharp-language](#): This is where new Q# features are developed and specified, and where you can share ideas and suggestions about the future evolution of the Q# language and core libraries.
- [MicrosoftDocs/quantum-docs](#): Source code for the documentation published at [docs.microsoft.com](https://docs.microsoft.com/quantum).

#### NOTE

We unfortunately cannot accept code and documentation contributions on the [microsoft/Quantum-NC](#) repository at this time, but we still very much appreciate bug reports.

There are also a few other, more specialized repositories focusing on auxiliary functionality related to the Quantum Development Kit.

- [msr-quarc/qsharp.sty](#): LaTeX formatting support for Q# syntax.

## Next steps

Thanks for being a part of the Quantum Development Kit community, we're excited for your contributions! If you'd like to learn more about contributing, please continue with one of the following guides.

[Learn how to report bugs](#)

[Learn how to contribute documentation](#)

[Learn how to open pull requests](#)

# Reporting Bugs

4/5/2021 • 2 minutes to read • [Edit Online](#)

If you think you've found a bug in a component of the Quantum Development Kit, we would very much appreciate a report. After all, someone else may be struggling with the same issue as well; letting us know helps us to get to fixing it for everyone. The first step in reporting a bug is to start by looking through the existing issues found on each repository, as it's very possible that someone else has reported the same problem.

If the issue has already been found before, it may be helpful to provide additional information or context that can help us to diagnose and solve the problem. In that case, please feel free to participate in discussions on the issue.

If your bug hasn't been reported before, then we'd appreciate if you open a new issue on the repository for the component in question. When creating a new issue, you'll be prompted with a template that suggests some information that is often helpful in making informative bug reports:

- Steps to reproduce the problem
- What you expected to happen
- What actually happened
- Information about your software environment (for example: OS, .NET Core, and Quantum Development Kit versions)

You can also [create a pull request](#) to fix the bug directly, if it's very straightforward and is not worth the discussion (for example, a typo).

# Improving documentation

6/1/2021 • 9 minutes to read • [Edit Online](#)

The documentation for the Quantum Development Kit takes on several different forms, such that information is readily available to quantum developers.

Following the principles of [Docs as Code](#), all Quantum Development Kit documentation is formatted as code and is managed using Git in the same way as the source code that is used to build the Quantum Development Kit. For the most part, the code backing documentation consists of various forms of [Markdown](#), a language for writing out richly formatted text in a plain text format that's easy to use at the command line, in IDEs, and with source control. We similarly adopt the [MathJax](#) library to allow for formatting mathematics in documentation using the LaTeX language, as detailed further below.

That said, each form of documentation does vary somewhat in the details:

- The **conceptual documentation** consists of a set of articles that are published to [docs.microsoft.com](#), and that describe everything from the basics of quantum computing to the technical specifications for interchange formats. These articles are written in [DocFX-Flavored Markdown \(DFM\)](#), a Markdown variant used for creating rich documentation sets.
- The **API reference** is a set of pages for each Q# function, operation, and user-defined type, published to [/qsharp/api/](#). These pages document the inputs and operations to each callable, along with examples and links to more information. The API reference is automatically extracted from small DFM documents in Q# source code as a part of each release.
- The **README.md** files included with each sample and kata describe how to use that sample or kata is used, what it covers, and how it relates to the rest of the Quantum Development Kit. These files are written using [GitHub Flavored Markdown \(GFM\)](#), a more lightweight alternative to DFM that's popular for attaching documentation directly to code repositories.

## Contributing to the conceptual documentation

To contribute an improvement to the conceptual or README documentation, then, starts with a pull request onto either [MicrosoftDocs/quantum-docs](#), [Microsoft/Quantum](#), or [Microsoft/QuantumKatas](#), as is appropriate. We'll describe more about pull requests below, but for now there's a few things that are good to keep in mind as you improve documentation:

- Readers come to the Quantum Development Kit documentation from a very wide range of backgrounds. Everyone from high school students looking to learn something new and exciting through to tenured faculty performing quantum computing research should be able to get something out of reading the documentation. To the extent that's possible, please don't assume extensive knowledge on the part of your readers, as they may just be starting out. It's most helpful if you can provide clear and accessible descriptions, or can provide links to other resources for more information.
- Documentation sets aren't laid out like books or papers, in that readers will arrive in what might seem like the "middle." For example, search engines might not suggest the index, or they might have been sent a link by a friend trying to help them out. Try to help your reader by always providing a clear context, along with links where appropriate.
- Some readers will find abstract statements and definitions most helpful, while other readers work best by extrapolating from concrete examples. Providing both the general case and specific examples can help both readers get the most out of quantum programming.
- Especially if you also wrote the code being documented, things may be obvious to you that are not at all obvious to your reader. There's no one unique best way to program, so no matter how clever or experienced

a reader might be, they can't guess at what design patterns you found the most helpful to express your ideas in code. Being clear about how a reader can expect to make use of your code can help provide that context.

- Many members of the quantum programming community are academic researchers, and are recognized mainly through citations for their contributions to the community. In addition to helping readers find additional materials, making sure to properly cite academic outputs such as papers, talks, blog posts, and software tools can help academic contributors to keep doing their best work to improve the community.
- The quantum programming community is a broad and wonderfully diverse community. The use of gendered pronouns in third-person examples (for example: "if a user ..., they will ...") can work to exclude rather than include. Being cognizant of people's names in citations and links, and of the correct inclusion of non-ASCII characters can serve the diversity of the community by showing respect to its members. Similarly, many words in the English language are often used in a hateful manner, such that their use in technical documentation can cause harm both to individual readers and to the community at large.

### Referencing sample code from conceptual articles

If you want to include code from the [samples repository](#), you can do so using a special DocFX-Flavored Markdown command:

```
:::code language="qsharp" source="~/quantum/samples/algorithms/chsh-game/Game.qs" range="4-8":::
```

This command will import lines 4 to 8 of the [Game.qs](#) file from the [chsh-game](#) sample, marking them as Q# code for the purpose of syntax highlighting. Using this command, you can avoid duplicating code between conceptual articles and the samples repository, so that sample code in documentation is always as up to date as possible.

### Contributing image files

**IMPORTANT:** To have the images rendering properly in dark mode you must avoid transparencies.

- For .jpg files, you don't need to do anything as the .jpg format doesn't support transparent elements.
- For .png files, you must add a white background or change the value of the alpha channel to **100**. The easiest way to do this in Windows is to open the file in **Paint** and save it, overwriting the original file.
- For .svg files you must add a white rectangle in the lowest layer. You can do this with **Inkscape**:
  1. Open the .svg file.
  2. Select the square maker tool and draw a white rectangle on top of the original figure.
  3. Select the tool **Select and transform objects** by clicking in the dark arrow or pressing **F1**.
  4. While having the rectangle selected, click the toolbar element **Lower selection to bottom (End)**.
  5. Adjust the rectangle with your mouse or the arrow keys.

## Contributing to the API references

To contribute an improvement to the API references, it's most helpful to open a pull request directly on the code being documented. Each function, operation, or user-defined type supports a documentation comment (denoted with `///` instead of `//`). When we compile each release of the Quantum Development Kit, these comments are used to generate the API reference at [/qsharp/api/](#), including details about the inputs to and outputs from each callable, the assumptions each callable makes, and examples of how to use them.

#### IMPORTANT

Please make sure to not manually edit the generated API documentation, as these files are overwritten with each new release. We value your contribution to the community, and want to make sure that your changes continue to help users release after release.

For example, consider the function

```
ControlledOnBitString<'T> (bits : Bool[], oracle : ('T => Unit is Adj + Ctl)) : ((Qubit[], 'T) => Unit is Adj + Ctl)
```

A documentation comment should help a user learn how to interpret `bits` and `oracle` and what the function is for. Each of these different pieces of information can be provided to the Q# compiler by a specially named Markdown section in the documentation comment. For the example of `ControlledOnBitString`, we might write something like the following:

```

/// # Summary
/// Returns a unitary operation that applies an oracle on the target register if the
/// control register state corresponds to a specified bit mask.
///
/// # Description
/// The output of this function is an operation that can be represented by a
/// unitary transformation $U$ such that
/// \begin{align}
/// U \ket{b_0 b_1 \cdots b_{n - 1}} \ket{\psi} = \ket{b_0 b_1 \cdots b_{n-1}} \otimes
/// \begin{cases}
/// V \ket{\psi} & \text{if } (b_0 b_1 \cdots b_{n - 1}) = \text{bits} \\
/// \ket{\psi} & \text{otherwise}
/// \end{cases}
/// \end{align},
/// \end{align}
/// where $V$ is a unitary transformation that represents the action of the
/// `oracle` operation.
///
/// # Input
/// ## bits
/// The bit string to control the given unitary operation on.
/// ## oracle
/// The unitary operation to be applied on the target register.
///
/// # Output
/// A unitary operation that applies `oracle` on the target register if the control
/// register state corresponds to the bit mask `bits`.
///
/// # Remarks
/// The length of `bits` and `controlRegister` must be equal.
///
/// Given a Boolean array `bits` and a unitary operation `oracle`, the output of this function
/// is an operation that performs the following steps:
/// * apply an `X` operation to each qubit of the control register that corresponds to `false`
/// element of the `bits`;
/// * apply `Controlled oracle` to the control and target registers;
/// * apply an `X` operation to each qubit of the control register that corresponds to `false`
/// element of the `bits` again to return the control register to the original state.
///
/// The output of the `Controlled` functor is a special case of `ControlledOnBitString` where `bits` is
/// equal to `[true, ..., true]`.
///
/// # Example
/// The following code snippets are equivalent:
/// ``qsharp
/// ControlledOnBitString(bits, oracle)(controlRegister, targetRegister);
/// ```
/// and
/// ``qsharp
/// within {
///     ApplyPauliFromBitString(PauliX, false, bits, controlRegister);
/// } apply {
///     Controlled oracle(controlRegister, targetRegister);
/// }
/// ```
///
/// The following code prepares a state  $\frac{1}{2}(\ket{00} - \ket{01} + \ket{10} + \ket{11})$ :
/// ``qsharp
/// use register = Qubit[2];
/// ApplyToEach(H, register);
/// ControlledOnBitString([false], Z)(register[0..0], register[1]);
/// ```
function ControlledOnBitString<'T> (bits : Bool[], oracle : ('T => Unit is Adj + Ctl)) : ((Qubit[], 'T) =>
Unit is Adj + Ctl)
{
    return ControlledOnBitStringImpl(bits, oracle, _,_);
}

```

You can see the rendered version of the code above in the [API documentation for the `controlledOnBitString` function](#).

In addition to the general practice of documentation writing, in writing API documentation comments it helps to keep a few things in mind:

- The format of each documentation comment has to match what the Q# compiler expects for your documentation to appear correctly. Some sections, such as `/// # Remarks` allow for freeform content, while sections such as `/// # See Also` section are more restrictive.
- A reader may read your API documentation on the main API reference site, on the summary for each namespace, or even from within their IDE through the use of hover information. Making sure that `/// # Summary` isn't longer than about a sentence helps your reader quickly sort out whether they need to read further or look elsewhere, and can help in quickly scanning through namespace summaries.
- Your documentation may well wind up being much longer than the code itself, but that's OK! Even a small piece of code can have unexpected effects to users that don't know the context in which that code exists. By providing concrete examples and clear explanations, you can help users make the best use of the code that's available to them.

# Opening Pull Requests

4/5/2021 • 2 minutes to read • [Edit Online](#)

All of the documentation for the Quantum Development Kit is managed using the Git version control system through the use of several repositories hosted on GitHub. Using Git and GitHub together makes it easy to collaborate widely on the Quantum Development Kit. In particular, any Git repository can be cloned or forked to make a completely independent copy of that repository. This allows you to work on your contribution with the tools and at a pace that you prefer.

When you're ready, you can send us a request to include your contribution into our repos, using GitHub's *pull request* functionality. The page for each pull request includes details of all the changes that make your contribution, a list of comments on your contribution, and a set of review tools that other members of the community can use to provide feedback and advice.

## NOTE

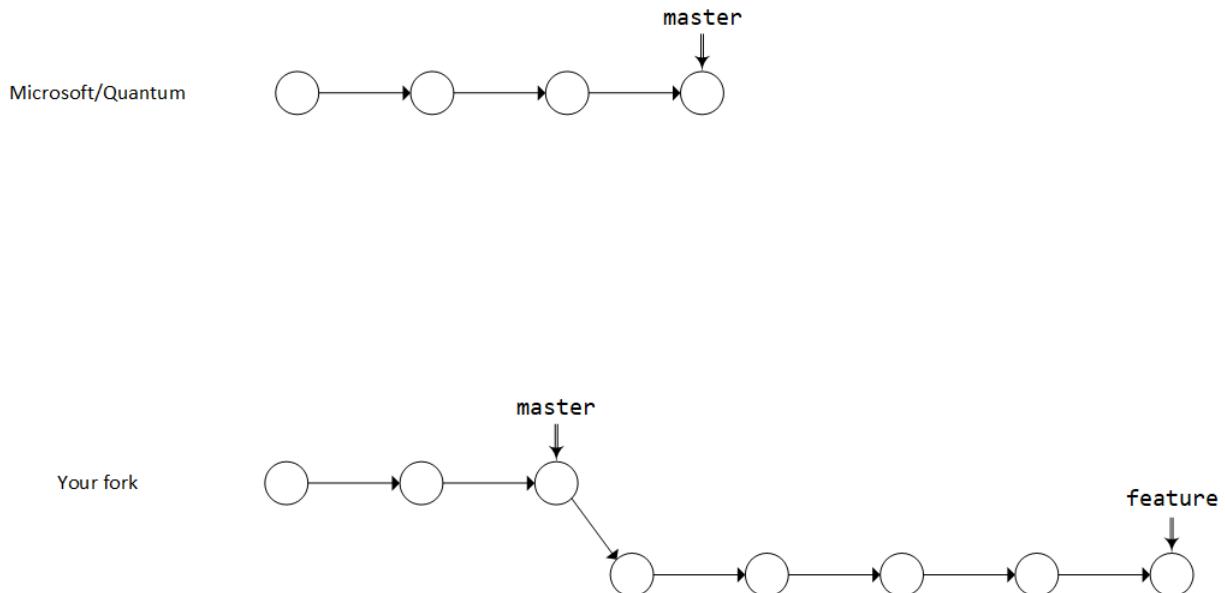
While a full tutorial on Git is beyond the scope of this guide, we can suggest the following links for more resources on learning Git:

- [Learn Git](#): A set of Git tutorials from Atlassian.
- [Version Control in Visual Studio Code](#): A guide on how to use Visual Studio Code as a GUI for Git.
- [GitHub Learning Lab](#): A set of online courses for using Git, GitHub, and related technologies.
- [Visualizing Git](#): An interactive tool for visualizing how Git commands change the state of a repository.
- [Version Control with Git \(EPQIS 2016\)](#): A Git tutorial oriented towards scientific computing.
- [Learn Git Branching](#): An interactive set of branching and rebasing puzzles to help learn new Git features.

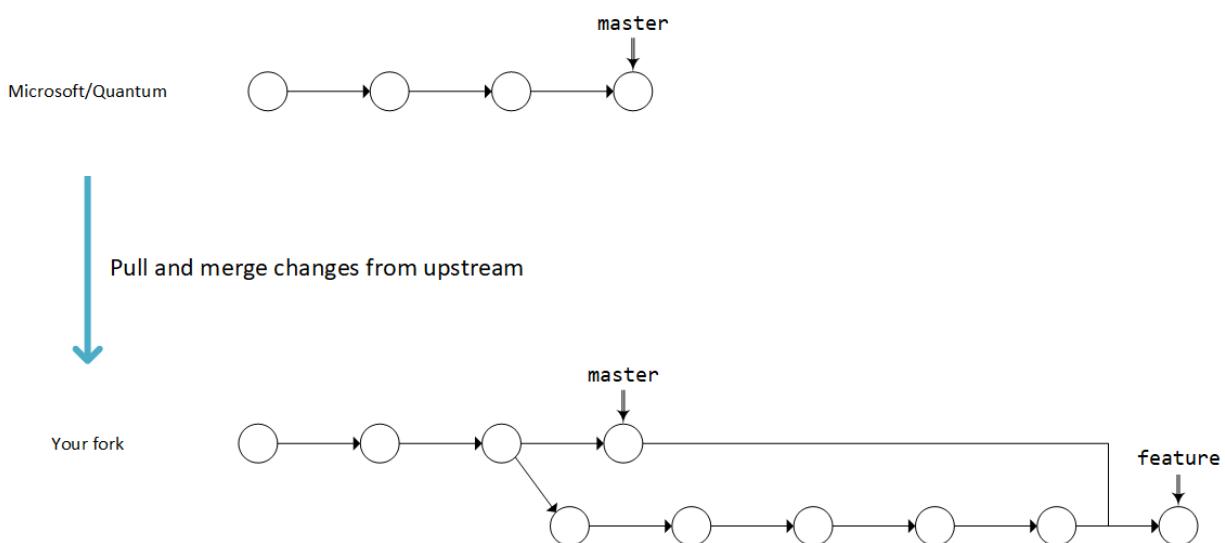
## What is a Pull Request?

Having said the above, it's helpful to take a few moments to say what a pull request *is*. When working with Git, any changes are represented as *commits* that describe how those changes are related to the state of the repository before those changes. We'll often draw diagrams in which commits are drawn as circles with arrows from previous commits.

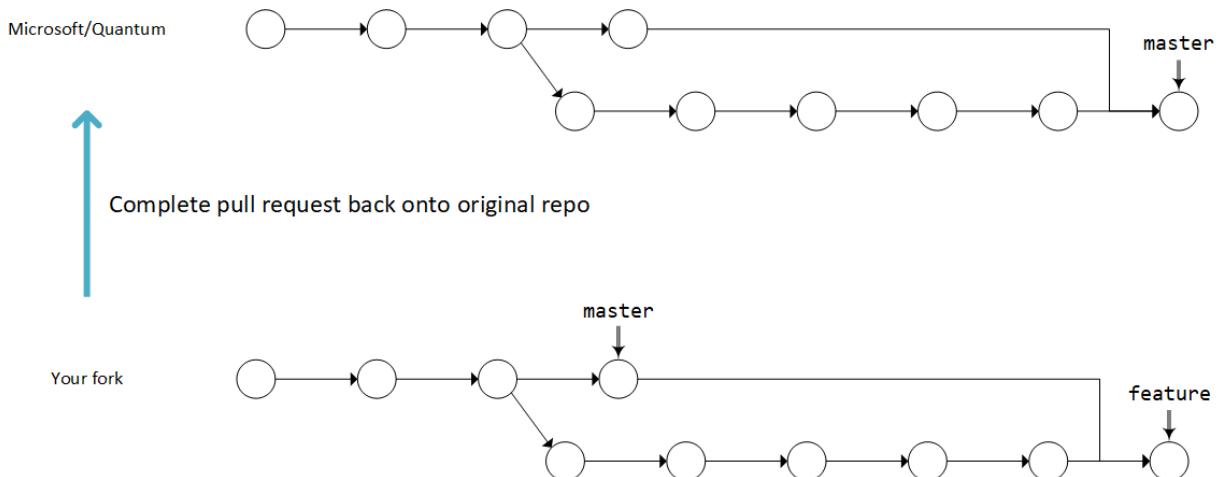
Suppose you have started a contribution in a *branch* called `feature`. Then your fork of Microsoft/Quantum might look something like this:



If you make your changes in your local repository, you can *pull* changes from another repository into yours to catch up to any changes that happened upstream.



Pull requests work the same way, but in reverse: when you open a pull request, you ask for the upstream repository to pull your contribution in.



When you open a pull request to one of our repositories, GitHub will offer an opportunity for others in the community to see a summary of your changes, to comment on them, and to make suggestions for how to help make an even better contribution.

# Updating packages to 0.4.1901.3104 #139

 Open

anpaz-msft wants to merge 3 commits into `master` from `release/v0.4.1901.3104`

 Conversation 0

 Commits 3

 Checks 0

 Files changed 31

Using this process helps us use GitHub functionality to improve contributions and to maintain a high-quality product for the quantum programming community.

## How to Make a Pull Request

There are two main ways to make a pull request.

For small changes that only affect a single file, the GitHub web interface can be used to make a pull request entirely online. Simply navigate to the file you want to edit and use the edit icon.

For more complicated contributions, it's most often easier to clone the repository to your local computer to prepare for a pull request first.

## Next steps

Congratulations on using Git to help out the Quantum Development Kit community! To learn more about how to contribute code, please continue with the following guide.

[Learn how to contribute code](#)

# Contributing Code

4/5/2021 • 5 minutes to read • [Edit Online](#)

In addition to reporting issues and improving documentation, contributing code to the Quantum Development Kit can be a very direct way to help your peers in the quantum programming community. By contributing code, you can help to fix issues, provide new examples, make existing libraries easier to use, or even add entirely new features.

In this guide, we'll detail a bit of what we look for when we review pull requests to help your contribution do the most good.

## What We Look For

An ideal code contribution builds on the existing work in a Quantum Development Kit repository to fix issues, expand existing features, or to add new features that are within the scope of a repository. When we accept a code contribution, it becomes a part of the Quantum Development Kit itself, such that new features will be released, maintained, and developed in the same way as the rest of the Quantum Development Kit. Thus, it is helpful when functionality added by a contribution is well-tested and is documented.

### Unit Tests

The Q# functions, operations, and user-defined types that make up libraries such as the canon are automatically tested as a part of development on the [Microsoft/QuantumLibraries](#) repository. When a new pull request is opened, for instance, our [Azure Pipelines](#) configuration will check that the changes in the pull request do not break any existing functionality that the quantum programming community depends on.

With the latest Q# version, unit tests are defined using the `@Test("QuantumSimulator")` attribute. The argument may be either "QuantumSimulator", "ToffoliSimulator", "TraceSimulator", or any fully qualified name specifying the run target. Several attributes defining different run targets may be attached to the same callable. Some of our tests still use the deprecated [Microsoft.Quantum.Xunit](#) package that exposes all Q# functions and operations ending in `Test` to the [xUnit](#) framework. This package is no longer needed for defining unit tests.

The following function is used to ensure that the [Fst function](#) and [Snd function](#) functions both return the right outputs in a representative example. If the output of `Fst` or `Snd` is incorrect, the `fail` statement is used to cause the test to fail.

```
@Test("QuantumSimulator")
function PairTest () : Unit {
    let pair = (12, PauliZ);

    if (Fst(pair) != 12) {
        let actual = Fst(pair);
        fail $"Expected 12, actual {actual}.";
    }

    if (Snd(pair) != PauliZ) {
        let actual = Snd(pair);
        fail $"Expected PauliZ, actual {actual}.";
    }
}
```

More complicated conditions can be checked using the techniques in the [testing section](#) of the standard libraries guide. For instance, the following test checks that `H(q); X(q); H(q);` as called by [ApplyWith operation](#) does the same thing as `z(q)`.

```
@Test("QuantumSimulator")
operation TestApplyWith() : Unit {
    let actual = ApplyWith(H, X, _);
    let expected = Z;
    AssertOperationsEqualReferenced(ApplyToEach(actual, _), ApplyToEachA(expected, _), 4);
}
```

When adding new features, it's a good idea to also add new tests to make sure that your contribution does what it's supposed to. This helps the rest of the community to maintain and develop your feature, and in particular helps other developers know that they can rely on your feature.

#### NOTE

This works the other way around, too! If there's an existing feature that's missing some tests, helping us add test coverage would make a great contribution to the community.

Locally, unit tests can be run using the Visual Studio Test Explorer or the `dotnet test` command, so that you can check your contribution before opening up a pull request.

## Pull Requests

When you are ready to contribute your work, please send a Pull Request via GitHub to the corresponding repository. The team will review and provide feedback. All comments need to be answered and resolved and all checks need to pass before the code is merged to the `main` branch.

## When We'll Reject a Pull Request

Sometimes, we'll reject the pull request for a contribution. If this happens to you, it doesn't mean that it's bad, as there's a number of reasons why we might not be able to accept a particular contribution. Perhaps most commonly, a contribution to the quantum programming community is a really good one, but the Quantum Development Kit repositories aren't the right place to develop it. In such cases, we strongly encourage you to make your own repository --- part of the strength of the Quantum Development Kit is that it's easy to make and distribute your own libraries using GitHub and NuGet.org, the same way that we distribute the canon and chemistry libraries today.

At other times, we may reject a good contribution simply because we aren't yet ready to maintain and develop it. It can be difficult to do everything, so we plan out what features we're best able to work on as a roadmap. This can be another case where releasing a feature as a third-party library can make a lot of sense. Alternatively, we may ask for your help in modifying a feature to better fit into our roadmap so that we can do the best work we can with it.

We'll also ask for changes to a pull request if it requires more documentation or unit tests to help us make use of it, or if it differs enough in style from the rest of the Q# libraries that it will make it harder for users to find your feature. In these cases, we'll try to offer some advice in code reviews about what can be added or changed to make your contribution easier for us to include.

Finally, we cannot accept contributions that cause harm the quantum computing community, as outlined in the [Microsoft Open Source Code of Conduct](#). We want to ensure that contributions serve the entire quantum computing community, both in its current wonderful diversity, and in the future as it grows to become still more inclusive. We appreciate your help in realizing this goal.

## Next steps

Thanks for helping to make the Quantum Development Kit a great resource for the entire quantum

programming community! To learn more, please continue with the following guide on Q# style.

### [Learn about Q# style guidelines](#)

Depending on what kind of code you're contributing, there may be additional things to keep in mind that can help you make your contribution do as much good for the community as possible.

### [Learn about contributing samples](#)

# Contributing Samples to the Quantum Development Kit

5/27/2021 • 3 minutes to read • [Edit Online](#)

If you're interested in contributing code to the [samples repository](#), thank you for making the quantum development community a better place!

## The Quantum Development Kit Samples Repository

To help you prepare your contribution to help out as much as possible, it's helpful to take a quick look at how the samples repository is laid out:

```
microsoft/Quantum
└─ samples/
    └─ algorithms/
        └─ chsh-game/
            └─ CHSHGame.csproj
            └─ Game.qs
            └─ Host.cs
            └─ host.py
            └─ README.md
            ...
    └─ arithmetic/
    └─ characterization/
    └─ chemistry/
    ...


```

That is, the samples in the [microsoft/Quantum repository](#) are broken down by subject area into different folders such as `algorithms/`, `arithmetic/`, or `characterization/`. Within the folder for each subject area, each sample consists of a single folder that collects everything a user will need to explore and make use of that sample.

## How Samples are Structured

Looking at the files that make up each folder, let's dive into the `algorithms/chsh-game/` sample.

FILE	DESCRIPTION
<code>CHSHGame.csproj</code>	Q# project used to build the sample with the .NET Core SDK
<code>Game.qs</code>	Q# operations and functions for the sample
<code>Host.cs</code>	C# host program used to run the sample
<code>host.py</code>	Python host program used to run the sample
<code>README.md</code>	Documentation on what the sample does and how to use it

Not all samples will have the exact same set of files (for example, some samples may be C#-only, others may not have a Python host, or some samples may require auxillary data files to work).

# Anatomy of a Helpful README File

One especially important file, though, is the `README.md` file, as that's what users need to get started with your sample!

Each `README.md` should start with some metadata that helps [docs.microsoft.com/samples](https://docs.microsoft.com/samples) find your contribution.

## See how the chsh-game sample is rendered

This metadata is provided as a [YAML header](#) that indicates what languages your sample covers (typically, this will be `qsharp`, `csharp`, and `python`), and what products your sample covers (typically, just `qdk`).

```
---
page_type: sample
languages:
- qsharp
- python
- csharp
products:
- qdk
description: "This sample uses the CHSH game to demonstrate how Q# programs can be used to prepare and work with entanglement."
urlFragment: validating-quantum-mechanics
---
```

### IMPORTANT

The `page_type: sample` key in the header is required for your sample to appear at [docs.microsoft.com/samples](https://docs.microsoft.com/samples).

Similarly, the `product` and `language` keys are critical for helping users to find and run your sample.

After that, it's helpful to give a short intro that says what your new sample does:

```
# Validating Quantum Mechanics with the CHSH Game

This sample demonstrates:
- How to prepare entangled states with Q#.
- How to measure part of an entangled register.
- Using Q# to understand superposition and entanglement.

In this sample, you can use Q# to prepare qubits in an entangled state, and to check that measuring these qubits lets you win a game known as the _CHSH game_ more often than you can without entanglement.
This game helps us understand entanglement, and has even been used experimentally to help test that the universe really is quantum mechanical in nature.
```

Users of your sample will also appreciate knowing what they need to run it (for example, do users just need the Quantum Development Kit itself, or do they need additional software such as node.js?):

```
## Prerequisites

- The Microsoft [Quantum Development Kit](/azure/quantum/install-overview-qdk).
```

With all that in place, you can tell users how to run your sample:

```
## Running the Sample

This sample can be run in a number of different ways, depending on your preferred environment.

### Python in Visual Studio Code or the Command Line

At a terminal, run the following command:

```powershell
python host.py
```

### C# in Visual Studio Code or the Command Line

At a terminal, run the following command:

```powershell
dotnet run
```

### C# in Visual Studio 2019

Open the folder containing this sample in Visual Studio ("Open a local folder" from the Getting Started screen or "File → Open → Folder..." from the menu bar) and set `CHSHGame.csproj` as the startup project. Press Start in Visual Studio to run the sample.
```

Finally, it's helpful to tell users what each file in your sample does, and where they can go for more information:

```
## Manifest

- [Game.qs](https://github.com/microsoft/Quantum/blob/main/samples/algorithms/chsh-game/Game.qs): Q# code implementing the game.
- [host.py](https://github.com/microsoft/Quantum/blob/main/samples/algorithms/chsh-game/host.py): Python host program to call into the Q# sample.
- [Host.cs](https://github.com/microsoft/Quantum/blob/main/samples/algorithms/chsh-game/Host.cs): C# code to call the operations defined in Q#.
- [CHSHGame.csproj](https://github.com/microsoft/Quantum/blob/main/samples/algorithms/chsh-game/CHSHGame.csproj): Main C# project for the sample.

## Further resources

- [Measurement concepts](/azure/quantum/concepts-pauli-measurements)
```

### WARNING

Make sure to use absolute URLs here, since your sample will appear at a different URL when rendered at [docs.microsoft.com/samples](https://docs.microsoft.com/samples)!

# Q# style guide

4/5/2021 • 19 minutes to read • [Edit Online](#)

## General conventions

The conventions suggested in this guide are intended to help make programs and libraries written in Q# programming language easier to read and understand.

## Guidance

We suggest:

- Never disregard a convention unless you're doing so intentionally in order to provide more readable and understandable code for your users.

## Naming conventions

In offering the Quantum Development Kit, we strive for function and operation names that help quantum developers write programs that are easy to read and that minimize surprise. An important part of that is that when we choose names for functions, operations, and types, we are establishing the *vocabulary* that programmers use to express quantum concepts; with our choices, we either help or hinder them in their effort to clearly communicate. This places a responsibility on us to make sure that the names we introduce offer clarity rather than obscurity. In this section, we detail how we meet this obligation in terms of explicit guidance that helps us do the best by the Q# development community.

### Operations and functions

One of the first things that a name should establish is whether a given symbol represents a function or an operation. The difference between functions and operations is critical to understanding how a block of code behaves. To communicate the distinction between functions and operations to users, we rely on that Q# models quantum operations through the use of side effects. That is, an operation *does* something.

By contrast, functions describe the mathematical relationships between data. The expression `Sin(PI() / 2.0)` is `1.0`, and implies nothing about the state of a program or its qubits.

Summarizing, operations do things while functions are things. This distinction suggests that we name operations as verbs and functions as nouns.

#### NOTE

When a user-defined type is declared, a new function that constructs instances of that type is implicitly defined at the same time. From that perspective, user-defined types should be named as nouns so that both the type itself and the constructor function have consistent names.

Where reasonable, ensure that operation names begin with verbs that clearly indicate the effect taken by the operation. For example:

- `MeasureInteger`
- `EstimateEnergy`
- `SampleInt`

One case that deserves special mention is when an operation takes another operation as input and calls it. In

such cases, the action taken by the input operation is not clear when the outer operation is defined, such that the right verb is not immediately clear. We recommend the verb `Apply`, as in `ApplyIf`, `ApplyToEach`, and `ApplyToFirst`. Other verbs may be useful in this case as well, as in `IterateThroughCartesianPower`.

| VERB     | EXPECTED EFFECT   |
|----------|---|
| Apply    | An operation provided as input is called  |
| Assert   | A hypothesis about the outcome of a possible quantum measurement is checked by a simulator  |
| Estimate | A classical value is returned, representing an estimate drawn from one or more measurements |
| Measure  | A quantum measurement is performed, and its result is returned to the user                  |
| Prepare  | A given register of qubits is initialized into a particular state                           |
| Sample   | A classical value is returned at random from some distribution                              |

For functions, we suggest avoiding the use of verbs in favor of common nouns (see guidance on proper nouns below) or adjectives:

- `ConstantArray`
- `Head`
- `LookupFunction`

In particular, in almost all cases, we suggest using past participles where appropriate to indicate that a function name is strongly connected to an action or side effect elsewhere in a quantum program. For example, `ControlledOnInt` uses the part participle form of the verb "control" to indicate that the function acts as an adjective to modify its argument. This name has the additional benefit of matching the semantics of the built-in `Controlled` functor, as discussed further below. Similarly, *agent nouns* can be used to construct function and UDT names from operation names, as in the case of the name `Encoder` for a UDT that is strongly associated with `Encode`.

- [Guidance](#)
- [Examples](#)

We suggest:

- Use verbs for operation names.
- Use nouns or adjectives for function names.
- Use nouns for user-defined types and attributes.
- For all callable names, use `CamelCase` in strong preference to `pascalCase`, `snake_case`, or `ANGRY_CASE`. In particular, ensure that callable names start with uppercase letters.
- For all local variables, use `pascalCase` in strong preference to `CamelCase`, `snake_case`, or `ANGRY_CASE`. In particular, ensure that local variables start with lowercase letters.
- Avoid the use of underscores `_` in function and operation names; where additional levels of hierarchy are needed, use namespaces and namespace aliases.

## Entry points

When defining an entry point into a Q# program, the Q# compiler recognizes the `@EntryPoint()` attribute

rather requiring that entry points have a particular name (for example: `main`, `Main`, or `__main__`). That is, from the perspective of a Q# developer, entry points are ordinary operations annotated with `@EntryPoint()`. Moreover, Q# entry points may be entry points for an entire application (for example, in Q# standalone executable programs), or may be an interface between a Q# program and the host program for an application (for example, when using Q# with Python or .NET), such that the name "main" could be misleading when applied to a Q# entry point.

We suggest using naming entry points to reflect the use of the `@EntryPoint()` attribute by using the general advice for naming operations listed above.

- [Guidance](#)
- [Examples](#)

We suggest:

- Do not name entry point operations as "main."
- Name entry point operations as ordinary operations.

### Shorthand and abbreviations

The above advice notwithstanding, there are many forms of shorthand that see common and pervasive use in quantum computing. We suggest using existing and common shorthand where it exists, especially for operations that are intrinsic to the operation of a target machine. For example, we choose the name `X` instead of `ApplyX`, and `Rz` instead of `RotateAboutZ`. When using such shorthand, operation names should be all uppercase (for example, `MAJ`).

Some care is required when applying this convention in the case of commonly used acronyms and initialisms such as "QFT" for "quantum Fourier transform." We suggest following general .NET conventions for the use of acronyms and initialisms in full names, which prescribe that:

- two-letter acronyms and initialisms are named in upper case (for example, `BE` for "big-endian"),
- all longer acronyms and initialisms are named in `CamelCase` (for example, `qft` for "quantum Fourier transform")

Thus, an operation implementing the QFT could either be called `QFT` as shorthand, or written out as `ApplyQft`.

For particularly commonly used operations and functions, it may be desirable to provide a shorthand name as an *alias* for a longer form:

```
operation CCNOT(control0 : Qubit, control1 : Qubit, target : Qubit)
is Adj + Ctl {
    Controlled X([control0, control1], target);
}
```

- [Guidance](#)
- [Examples](#)

We suggest:

- Consider commonly accepted and widely used shorthand names when appropriate.
- Use uppercase for shorthand.
- Use uppercase for short (two-letter) acronyms and initialisms.
- Use `CamelCase` for longer (three or more letter) acronyms and initialisms.

### Proper nouns in names

While in physics it is common to name things after the first person to publish about them, most non-physicists

aren't familiar with everyone's names and all of the history. Relying too heavily on naming conventions from physics can thus put up a substantial barrier to entry, as users from other backgrounds must learn a large number of seemingly opaque names in order to use common operations and concepts.

Thus, we recommend that wherever reasonable, common nouns that describe a concept be adopted in strong preference to proper nouns that describe the publication history of a concept. As a particular example, the singly controlled SWAP and doubly controlled NOT operations are often called the "Fredkin" and "Toffoli" operations in academic literature, but are identified in Q# primarily as `CSWAP` and `CCNOT`. In both cases, the API documentation comments provide synonymous names based on proper nouns, along with all appropriate citations.

This preference is especially important given that some usage of proper nouns will always be necessary — Q# follows the tradition set by many classical languages, for instance, and refers to `Bool` types in reference to Boolean logic, which is in turn named in honor of George Boole. A few quantum concepts similarly are named in a similar fashion, including the `Pauli` type built-in to the Q# language. By minimizing the usage of proper nouns where such usage is not essential, we reduce the impact where proper nouns cannot be reasonably avoided.

- [Guidance](#)
- [Examples](#)

We suggest:

- Avoid the use of proper nouns in names.

### Type conversions

Since Q# is a strongly and statically typed language, a value of one type can only be used as a value of another type by using an explicit call to a type conversion function. This is in contrast to languages which allow for values to change types implicitly (for example, type promotion), or through casting. As a result, type conversion functions play an important role in Q# library development, and comprise one of the more commonly encountered decisions about naming. We note, however, that since type conversions are always *deterministic*, they can be written as functions and thus fall under the advice above. In particular, we suggest that type conversion functions should never be named as verbs (for example, `ConvertToX`) or adverb prepositional phrases (`ToX`), but should be named as adjective prepositional phrases that indicate the source and destination types (`XAsY`). When listing array types in type conversion function names, we recommend the shorthand `Arr`. Barring exceptional circumstances, we recommend that all type conversion functions be named using `As` so that they can be quickly identified.

- [Guidance](#)
- [Examples](#)

We suggest:

- If a function converts a value of type `X` to a value of type `Y`, use either the name `AsY` or `XAsY`.

### Private or internal names

In many cases, a name is intended strictly for use internal to a library or project, and is not a guaranteed part of the API offered by a library. It is helpful to clearly indicate that this is the case when naming functions and operations so that accidental dependencies on internal-only code are made obvious. If an operation or function is not intended for direct use, but rather should be used by a matching callable which acts by partial application, consider using a name starting with the `internal` keyword for the callable that is partially applied.

- [Guidance](#)
- [Examples](#)

We suggest:

- When a function, operation, or user-defined type is not a part of the public API for a Q# library or program, ensure that it is marked as internal by placing the `internal` keyword before the `function`, `operation`, or `newtype` declaration.

## Variants

Though this limitation may not persist in future versions of Q#, it is presently the case that there will often be groups of related operations or functions that are distinguished by which functors their inputs support, or by the concrete types of their arguments. These groups can be distinguished by using the same root name, followed by one or two letters that indicate its variant.

| SUFFIX | MEANING  |
|--------|--|
| A      | Input expected to support <code>Adjoint</code>                             |
| C      | Input expected to support <code>Controlled</code>                          |
| CA     | Input expected to support <code>Controlled</code> and <code>Adjoint</code> |
| I      | Input or inputs are of type <code>Int</code>                               |
| D      | Input or inputs are of type <code>Double</code>                            |
| L      | Input or inputs are of type <code>BigInt</code>                            |

- [Guidance](#)
- [Examples](#)

We suggest:

- If a function or operation is not related to any similar functions or operations by the types and functor support of their inputs, do not use a suffix.
- If a function or operation is related to any similar functions or operations by the types and functor support of their inputs, use suffixes as in the table above to distinguish variants.

## Arguments and variables

A key goal of the Q# code for a function or operation is for it to be easily read and understood. Similarly, the names of inputs and type arguments should communicate how a function or argument will be used once provided.

- [Guidance](#)
- [Examples](#)

We suggest:

- For all variable and input names, use `pascalCase` in strong preference to `CamelCase`, `snake_case`, or `ANGRY_CASE`.
- Input names should be descriptive; avoid one or two letter names where possible.
- Operations and functions accepting exactly one type argument should denote that type argument by `T` when its role is obvious.
- If a function or operation takes multiple type arguments, or if the role of a single type argument is not obvious, consider using a short capitalized word prefaced by `T` (for example, `Toutput`) for each type.

- Do not include type names in argument and variable names; this information can and should be provided by your development environment.
- Denote scalar types by their literal names (`flagQubit`), and array types by a plural (`measResults`). For arrays of qubits in particular, consider denoting such types by `Register` where the name refers to a sequence of qubits that are closely related in some way.
- Variables used as indices into arrays should begin with `idx` and should be singular (for example, `things[ idxThing ]`). In particular, strongly avoid using single-letter variable names as indices; consider using `idx` at a minimum.
- Variables used to hold lengths of arrays should begin with `n` and should be pluralized (for example, `nThings`).

### User-defined type named items

Named items in user-defined types should be named as `CamelCase`, even in input to UDT constructors. This helps in order to clearly separate named items from references to locally scoped variables when using accessor notation (for example, `callable::Apply`) or copy-and-update notation (`set arr w/= Data <- newData`).

- [Guidance](#)
- [Examples](#)

We suggest:

- Named items in UDT constructors should be named as `CamelCase`; that is, they should begin with an initial uppercase.
- Named items that resolve to operations should be named as verb phrases.
- Named items that do not resolve to operations should be named as noun phrases.
- For UDTs that wrap operations, a single named item called `Apply` should be defined.

## Input conventions

When a developer calls into an operation or function, the various inputs to that operation or function must be specified in a particular order, increasing the cognitive load that a developer faces in order to make use of a library. In particular, the task of remembering input orderings is often a distraction from the task at hand: programming an implementation of a quantum algorithm. Though rich IDE support can mitigate this to a large extent, good design and adherence to common conventions can also help to minimize the cognitive load imposed by an API.

Where possible, it can be helpful to reduce the number of inputs expected by an operation or function, so that the role of each input is more immediately obvious both to developers calling into that operation or function, and to developers reading that code later. Especially when it is not possible or reasonable to reduce the number of arguments to an operation or function, it is important to have a consistent ordering that minimizes the surprise that a user faces when predicting the order of inputs.

We recommend an input ordering conventions that largely derives from thinking of partial application as a generalization of currying  $\square(\square, \square) \equiv \square(\square)(\square)$ . Thus, partially applying the first arguments should result in a callable that is useful in its own right whenever that is reasonable. Following this principle, consider using the following order of arguments:

- Classical non-callable arguments such as angles, vectors of powers, etc.
- Callable arguments (functions and arguments). If both functions and operations are taken as arguments, consider placing operations after functions.
- Collections acted upon by callable arguments in a similar way to `Map`, `Iter`, `Enumerate`, and `Fold`.
- Qubit arguments used as controls.
- Qubit arguments used as targets.

Consider an operation `ApplyPhaseEstimationIteration` for use in phase estimation that takes an angle and an oracle, passes the angle to `Rz` modified by an array of different scaling factors, and then controls applications of the oracle. We would order the inputs to `ApplyPhaseEstimationIteration` in the following fashion:

```
operation ApplyPhaseEstimationIteration(
    angle : Double,
    callable : (Qubit => () is Ctl),
    scaleFactors : Double[],
    controlQubit : Qubit,
    targetQubits : Qubit[]
)
: Unit
...
```

As a special case of minimizing surprise, some functions and operations mimic the behavior of the built-in functors `Adjoint` and `Controlled`. For instance, `ControlledOnInt<'T>` has type

`(Int, ('T => Unit is Adj + Ctl)) => ((Qubit[], 'T) => Unit is Adj + Ctl)`, such that

`ControlledOnInt<Qubit[]>(5, _)` acts like the `Controlled` functor, but on the condition that the control register represents the state  $\lvert \text{ket}5 \rangle = \lvert \text{ket}101 \rangle$ . Thus, a developer expects that the inputs to `ControlledOnInt` place the callable being transformed last, and that the resulting operation takes as its input `(Qubit[], 'T)` --- the same order as followed by the output of the `Controlled` functor.

- [Guidance](#)
- [Examples](#)

We suggest:

- Use input orderings consistent with the use of partial application.
- Use input orderings consistent with built-in functors.
- Place all classical inputs before any quantum inputs.

## Documentation conventions

The Q# language allows for attaching documentation to operations, functions, and user-defined types through the use of specially formatted documentation comments. Denoted by triple-slashes (`///`), these documentation comments are small [DocFX-flavored Markdown](#) documents that can be used to describing the purpose of each operation, function, and user-defined type, what inputs each expects, and so forth. The compiler provided with the Quantum Development Kit extracts these comments and uses them to help typeset documentation similar to that at [docs.microsoft.com](#). Similarly, the language server provided with the Quantum Development Kit uses these comments to provide help to users when they hover over symbols in their Q# code. Making use of documentation comments can thus help users to make sense of code by providing a useful reference for details that are not easily expressed using the other conventions in this document.

[Documentation comment syntax reference](#).

In order to effectively use this functionality to help users, we recommend keeping a few things in mind as you write documentation comments.

- [Guidance](#)
- [Examples](#)

We suggest:

- Each public function, operation, and user-defined type should be immediately preceded by a documentation comment.

- At a minimum, each documentation comment should include the following sections:
  - Summary
  - Input
  - Output (if applicable)
- Ensure that all summaries are two sentences or less. If more room is needed, provide a `# Description` section immediately following `# Summary` with complete details.
- Where reasonable, do not include math in summaries, as not all clients support TeX notation in summaries. Note that when writing prose documents (for example, TeX or Markdown), it may be preferable to use longer line lengths.
- Provide all relevant mathematical expressions in the `# Description` section.
- When describing inputs, do not repeat the types of each input as these can be inferred by the compiler and risk introducing inconsistency.
- Provide examples as appropriate, each in their own `# Example` section.
- Briefly describe each example before listing code.
- Cite all relevant academic publications (for example: papers, proceedings, blog posts, and alternative implementations) in a `# References` section as a bulleted list of links.
- Ensure that, where possible, all citation links are to permanent and immutable identifiers (DOIs or versioned arXiv numbers).
- When an operation or function is related to other operations or functions by functor variants, list other variants as bullets in the `# See Also` section.
- Leave a blank comment line between level-1 (`/// #`) sections, but do not leave a blank line between level-2 (`/// ##`) sections.

## Formatting conventions

In addition to the preceding suggestions, it is helpful to help make code as legible as possible to use consistent formatting rules. Such formatting rules by nature tend to be somewhat arbitrary and strongly up to personal aesthetics. Nonetheless, we recommend maintaining a consistent set of formatting conventions within a group of collaborators, and especially for large Q# projects such as the Quantum Development Kit itself. These rules can be automatically applied by using the formatting tool integrated with the Q# compiler.

- [Guidance](#)
- [Examples](#)

We suggest:

- Use four spaces instead of tabs for portability. For instance, in VS Code:

```
"editor.insertSpaces": true,
"editor.tabSize": 4
```

- Line wrap at 79 characters where reasonable.
- Use spaces around binary operators.
- Use spaces on either side of colons used for type annotations.
- Use a single space after commas used in array and tuple literals (for example, in inputs to functions and operations).
- Do not use spaces after function, operation, or UDT names, or after the `@` in attribute declarations.
- Each attribute declaration should be on its own line.

# Q# API Design Principles

4/17/2021 • 14 minutes to read • [Edit Online](#)

## Introduction

As a language and as a platform, Q# empowers users to write, run, understand, and explore quantum applications. In order to empower users, when we design Q# libraries, we follow a set of API design principles to guide our designs and to help us make usable libraries for the quantum development community. This article lists these principles, and gives examples to help guide how to apply them when designing Q# APIs.

### TIP

This is a fairly detailed document that's intended to help guide library development and in-depth library contributions. You'll probably find it most useful if you're writing your own libraries in Q#, or if you're contributing larger features to the [Q# libraries repository](#).

On the other hand, if you're looking to learn how to contribute to the Quantum Development Kit more generally, we suggest starting with the [contribution guide](#). If you're looking for more general information about how we recommend formatting your Q# code, you may be interested in checking out the [style guide](#).

## General Principles

**Key principle:** Expose APIs that places the focus on quantum applications.

- **DO** choose operation and function names that reflect the high-level structure of algorithms and applications.
- **DON'T** expose APIs that focus primarily on low-level implementation details.

**Key principle:** Start each API design with sample use cases to ensure that APIs are intuitive to use.

- **DO** ensure that each component of a public API has a corresponding use case, rather than trying to design for all possible uses from the start. Put differently, don't introduce public APIs in case they are useful, but make sure that each part of an API has a *concrete* example in which it will be useful.

*Examples:*

- [ApplyToEachCA](#) operation can be used as `ApplyToEachCA(H, _)` to prepare registers in a uniform superposition state, a common task in many quantum algorithms. The same operation can also be used for many other tasks in preparation, numerics, and oracle-based algorithms.
- **DO** brainstorm and workshop new API designs to double-check that they are intuitive and meet proposed use cases.

*Examples:*

- Inspect current Q# code to see how new API designs could simplify and clarify existing implementations.
- Review proposed API designs with representatives of primary audiences.

**Key principle:** Design APIs to support and encourage readable code.

- **DO** ensure that code is readable by domain experts and non-experts alike.
- **DO** place the focus on the effects of each operation and function within the high-level algorithm, using

documentation to delve into implementation details as appropriate.

- DO follow the common [Q# style guide](#) whenever applicable.

**Key principle:** Design APIs to be stable and to provide forward compatibility.

- DO deprecate old APIs gracefully when breaking changes are required.
- DO provide "shim" operations and functions that allow existing user code to operate correctly during deprecation.

*Examples:*

- When renaming an operation called `EstimateExpectation` to `EstimateAverage`, introduce a new operation called `EstimateExpectation` that calls the original operation at its new name, so that existing code can continue to work correctly.
- DO use the [Deprecated user defined type](#) attribute to communicate deprecations to the user.
- When renaming an operation or function, DO provide the new name as a string input to `@Deprecated`.
- DON'T remove existing functions or operations without a deprecation period of at least six months for preview releases, or at least two years for supported releases.

## Functions and Operations

**Key principle:** ensure that every function and operation has a single well-defined purpose within the API.

- DON'T expose functions and operations that perform multiple unrelated tasks.

**Key principle:** design functions and operations to be as reusable as possible, and to anticipate future needs.

- DO design functions and operations to compose well with other functions and operations, both in the same API and in previously existing libraries.

*Examples:*

- The [Delay operation](#) operation makes minimal assumptions about its input, and thus can be used to delay applications of either operations across the Q# standard library or as defined by users.
- DO expose purely deterministic classical logic as functions rather than operations.

*Examples:*

- A subroutine which squares its floating-point input can be written deterministically, and so should be exposed to the user as `Squared : Double -> Double` rather than as an operation `Square : Double => Double`. This allows for the subroutine to be called in more places (for example: inside of other functions), and provides useful optimization information to the compiler that can affect performance and optimizations.
- `ForEach<'TInput, 'TOutput>('TInput => 'TOutput, 'TInput[]) => 'TOutput[]` and `Mapped<'TInput, 'TOutput>('TInput -> 'TOutput, 'TInput[]) -> 'TOutput[]` differ in the guarantees made with respect to determinism; both are useful in different circumstances.
- API routines that transform the application of quantum operations can often be carried out in a deterministic fashion and hence can be made available as functions such as `CControlled<'T>(op : 'T => Unit) => ((Bool, 'T) => Unit)`.
- DO generalize the input type as much as reasonable for each function and operation, using type parameters as needed.

*Examples:*

- `ApplyToEach` has type `<'T>('T => Unit), 'T[] => Unit` rather than the specific type of its most common application, `((Qubit => Unit), Qubit[]) => Unit`.

#### TIP

It is important to anticipate future needs, but it is also important to solve concrete problems for your users. Acting on this key principle thus always requires careful consideration and balancing to avoid developing APIs "just in case."

**Key principle:** choose input and output types for functions and operations that are predictable, and that communicate the purpose of a callable.

- **DO** use tuple types to logically group inputs and outputs that are only significant when considered together. Consider using a user-defined type in these cases.

*Examples:*

- A function to output the local minima of another function may need to take bounds of a search interval as input, such that  
`LocalMinima(fn : (Double -> Double), (left : Double, right : Double)) : Double` may be an appropriate signature.
- An operation to estimate a derivative of a machine learning classifier using the parameter shift technique may need to take both the shifted and unshifted parameter vectors as inputs. An input similar to `(unshifted : Double[], shifted : Double[])` may be appropriate in this case.
- **DO** order items in input and output tuples consistently across different functions and operations.

*Examples:*

- If considering two or functions or operations that each take a rotation angle and a target qubit as inputs, ensure that they are ordered the same in each input tuple. That is, prefer  
`ApplyRotation(angle : Double, target : Qubit) : Unit` is `Adj + Ctl` and  
`DelayedRotation(angle : Double, target : Qubit) : (Unit => Unit)` is `Adj + Ctl` to  
`ApplyRotation(target : Qubit, angle : Double) : Unit` is `Adj + Ctl` and  
`DelayedRotation(angle : Double, target : Qubit) : (Unit => Unit)`.

**Key principle:** design functions and operations to work well with Q# language features such as partial application.

- **DO** order items in input tuples such that the most commonly applied inputs occur first (for example, so that partial application acts similarly to currying).

*Examples:*

- An operation `ApplyRotation` that takes a floating-point number and a qubit as inputs may often be partially applied with the floating-point input first for use with operations that expect an input of type `Qubit => Unit`. Thus, a signature of  
`operation ApplyRotation(angle : Double, target : Qubit) : Unit` is `Adj + Ctl` would work most consistently with partial application.
- Typically, this guidance means placing all classical data before all qubits in input tuples, but use good judgment and examine how your API is called in practice.

## User-Defined Types

**Key principle:** use user-defined types to help make APIs more expressive and convenient to use.

- **DO** introduce new user-defined types to provide helpful shorthand for long and/or complicated

types.

*Examples:*

- In cases where an operation type with three qubit array inputs is commonly taken as an input or returned as an output, providing a UDT such as  
`newtype TimeDependentBlockEncoding = ((Qubit[], Qubit[], Qubit[]) => Unit is Adj + Ctl)` can help provide a useful shorthand.
- DO introduce new user-defined types to indicate that a given base type should only be used in a very particular sense.

*Examples:*

- An operation that should be interpreted specifically as an operation that encodes classical data into a quantum register may be appropriate to label with a user-defined type  
`newtype InputEncoder = (Apply : (Qubit[] => Unit))`.
- DO introduce new user-defined types with named items that allow for future extensibility (for example: a results structure that may contain additional named items in the future).

*Examples:*

- When an operation `TrainModel` exposes a large number of configuration options, exposing these options as a new `TrainingOptions` UDT and providing a new function `DefaultTrainingOptions : Unit -> TrainingOptions` allows users to override specific named items in `TrainingOptions` UDT values while still allowing library developers to add new UDT items as appropriate.
- DO declare named items for new user-defined types in preference to requiring users to know the correct tuple deconstruction.

*Examples:*

- When representing a complex number in its polar decomposition, prefer  
`newtype ComplexPolar = (Magnitude: Double, Argument: Double)` to  
`newtype ComplexPolar = (Double, Double)`.

**Key principle:** use user-defined types in ways reduce cognitive load and that don't require the user to learn additional concepts and nomenclature.

- DON'T introduce user-defined types that require the user to make frequent use of the unwrap operator (`!`), or that commonly require multiple levels of unwrapping. Possible mitigation strategies include:
  - When exposing a user-defined type with a single item, consider defining a name for that item. For instance, consider `newtype Encoder = (Apply : (Qubit[] => Unit is Adj + Ctl))` in preference to `newtype Encoder = (Qubit[] => Unit is Adj + Ctl)`.
  - Ensuring that other functions and operations can accept "wrapped" UDT instances directly.
- DON'T introduce new user-defined types that duplicate built-in types without providing additional expressiveness.

*Examples:*

- A UDT `newtype QubitRegister = Qubit[]` provides no additional expressiveness over `Qubit[]`, and is thus harder to use with no discernable benefit.
- A UDT `newtype LittleEndian = Qubit[]` documents how the underlying register is to be used and interpreted, and thus provides additional expressiveness over its base type.

- DON'T introduce accessor functions unless strictly required; strongly prefer named items in this case.

*Examples:*

- When introducing a UDT `newtype Complex = (Double, Double)`, prefer modifying the definition to `newtype Complex = (Real : Double, Imag : Double)` to introducing functions `GetReal : Complex -> Double` and `GetImag : Complex -> Double`.

## Namespaces and Organization

**Key principle:** choose namespace names that are predictable and that clearly communicate the purpose of functions, operations, and user-defined types in each namespace.

- DO name namespaces as `Publisher.Product.DomainArea`.

*Examples:*

- Functions, operations, and UDTs published by Microsoft as a part of the quantum simulation feature of the Quantum Development Kit are placed in the `Microsoft.Quantum.Simulation` namespace.
- `Microsoft.Quantum.Math` represents a namespace published by Microsoft as part of the Quantum Development Kit pertaining to the mathematics domain area.
- DO place operations, functions, and user-defined types used for specific functionality into a namespace that describes that functionality, even when that functionality is used across different problem domains.

*Examples:*

- State preparation APIs published by Microsoft as a part of the Quantum Development Kit would be placed into `Microsoft.Quantum.Preparation`.
- Quantum simulation APIs published by Microsoft as a part of the Quantum Development Kit would be placed into `Microsoft.Quantum.Simulation`.
- DO place operations, functions, and user-defined types used only within specific domains into namespaces indicating their domain of utility. If needed, use subnamespaces to indicate focused tasks within each domain-specific namespace.

*Examples:*

- The quantum machine learning library published by Microsoft is largely placed into the `Microsoft.Quantum.MachineLearning` namespace, but example datasets are provided by the `Microsoft.Quantum.MachineLearning.Datasets` namespace.
- Quantum chemistry APIs published by Microsoft as a part of the Quantum Development Kit should be placed into the `Microsoft.Quantum.Chemistry` namespace. Functionality specific to implementing the Jordan--Wigner decomposition may be placed in the `Microsoft.Quantum.Chemistry.JordanWigner` namespace, so that the primary interface for the quantum chemistry domain area is not concerned with implementations.

**Key principle:** Use namespaces and access modifiers together to be intentional about the API surface exposed to users, and to hide internal details related to implementation and testing of your APIs.

- Whenever reasonable, DO place all functions and operations needed to implement an API into the same namespace as the API being implemented, but marked with the "private" or "internal" keywords to indicate that they are not part of the public API surface for a library. Use a name beginning with an underscore (`_`) to visually distinguish private and internal operations and functions from public callables.

*Examples:*

- The operation name `_Features` indicates a function that is private to a given namespace and assembly, and should be accompanied by either the `internal` keyword.
- In the rare case that an extensive set of private functions or operations are needed to implement the API for a given namespace, DO place them in a new namespace matching the namespace being implemented and ending in `.Private`.
- DO place all unit tests into namespaces matching the namespace under test and ending in `.Tests`.

## Naming Conventions and Vocabulary

**Key principle:** Choose names and terminology that are clear, accessible, and readable across a diverse range of audiences, including both quantum novices and experts.

- DON'T use discriminatory or exclusionary identifier names, nor terminology in API documentation comments.
- DO use API documentation comments to provide relevant context, examples, and references, especially for more difficult concepts.
- DON'T use identifier names that are unnecessarily esoteric, or that require significant quantum algorithms knowledge to read.

*Examples:*

- Prefer "amplitude amplification iteration" to "Grover iteration."
- DO choose operations and function names that clearly communicate the intended effect of a callable, and not its implementation. Note that the implementation can and should be documented in [API documentation comments](#).

*Examples:*

- Prefer "estimate overlap" to "Hadamard test," as the latter communicates how the former is implemented.
- DO use words in a consistent fashion across all Q# APIs:

○ **Verbs:**

- **Assert:** Check that an assumption about the state of a target machine and its qubits holds, possibly by using unphysical resources. Operations using this verb should always be safely removable without affecting the functionality of libraries and executable programs. Note that unlike facts, assertions may, in general, depend on external state, such as the state of a qubit register, the run environment or so forth. As dependency on external state is a kind of side effect, assertions must be exposed as operations rather than functions.
- **Estimate:** Using one or more possibly repeated measurements, estimate a classical quantity from measurement results.

*Examples:*

- [EstimateFrequency](#) operation
- [EstimateOverlapBetweenStates](#) operation
- **Prepare:** Apply a quantum operation or sequence of operations to one or more qubits assumed to start in a particular initial state (typically  $\ket{00\cdots 0}$ ), causing the state of those qubits to evolve to a desired end state. In general, acting on states other than the given starting state **MAY** result in an undefined unitary transformation, but **SHOULD** still preserve that an operation and its adjoint "cancel out" and apply a no-op.

*Examples:*

- [PrepareArbitraryState](#) operation
- [PrepareUniformSuperposition](#) operation
- **Measure**: Apply a quantum operation or sequence of operations to one or more qubits, reading classical data back out.

*Examples:*

- [Measure](#) operation
- [MeasureFxP](#) operation
- [MeasureInteger](#) operation
- **Apply**: Apply a quantum operation or sequence of operations to one or more qubits, causing the state of those qubits to change in a coherent fashion. This verb is the most general verb in Q# nomenclature, and **SHOULD NOT BE** used when a more specific verb is more directly relevant.

- **Nouns:**

- **Fact**: A Boolean condition which depends only on its inputs and not on the state of a target machine, its environment, or the state of the machine's qubits. By contrast with an assertion, a fact is only sensitive to the *values* provided to that fact. For example:

*Examples:*

- [EqualityFactl](#) function: represents an equality fact about two integer inputs; either the integers provided as input are equal to each other, or they are not, independent of any other program state.
- **Options**: A UDT containing several named items that can act as "optional arguments" to a function or operation. For example:

*Examples:*

- The [TrainingOptions](#) user defined type UDT includes named items for learning rate, minibatch size, and other configurable parameters for ML training.

- **Adjectives:**

- **⊖ New**: This adjective **SHOULD NOT** be used, as to avoid confusion with its usage as a verb in many programming languages (e.g.: C++, C#, Java, TypeScript, PowerShell).
- **Prepositions**: In some cases, prepositions can be used to further disambiguate or clarify the roles of nouns and verbs in function and operation names. Care should be taken to do so sparingly and consistently, however.
  - **As**: Represents that a function's input and output represent the same information, but that the output represents that information **as** an *X* instead of its original representation. This is especially common for type conversion functions.

*Examples:*

- `IntAsDouble(2)` indicates that both the input (`2`) and the output (`2.0`) represent qualitatively the same information, but using different Q# data types to do so.
- **From**: To ensure consistency, this preposition **SHOULD NOT** be used to indicate type conversion functions or any other case where **As** is appropriate.
- **⊖ To**: This preposition **SHOULD NOT** be used, as to avoid confusion with its usage as a verb in many programming languages.

# The Q# User Guide

5/27/2021 • 2 minutes to read • [Edit Online](#)

Welcome to the Q# User Guide!

In the different topics of this guide, we introduce some of the basics for developing quantum programs using Q#.

We refer to the [Q# language guide](#) for a full specification and documentation of the Q# quantum programming language.

## User Guide Contents

- [Q# programs](#): An quick introduction to quantum programs in Q#.
- [Ways to run a Q# program](#): describes how a Q# program is run, and provides an overview of the various ways you can call the program: from the command line, in Q# Jupyter Notebooks, or from a classical host program written in Python or a .NET language.
- [Testing and debugging](#): Details some techniques for making sure your code is doing what it is supposed to do. Due to the general opacity of quantum information, debugging a quantum program can require specialized techniques. Fortunately, Q# supports many of the classical debugging techniques programmers are familiar with, as well as those that are quantum-specific. These include creating and running unit tests in Q#, embedding *assertions* on values and probabilities in your code, and the `Dump` functions which output the states of target machines. The latter can be used alongside our full-state simulator to debug certain parts of computations by skirting some quantum limitations, for example, the [no-cloning theorem](#).

### Quantum Simulators and Resource Estimators

- [Quantum simulators and host applications](#): An overview of the different simulators available, as well of how host programs and target machines work together to run Q# programs.
- [Full state simulator](#): The target machine which simulates the full quantum state. Useful for fully running or debugging smaller-scale programs (less than a few dozen qubits)
- [Resources estimator](#): Estimates the resources required to run a given instance of a Q# operation on a quantum computer.
- [Trace simulator](#): Runs a quantum program without actually simulating the state of a quantum computer, and therefore can run quantum programs that use thousands of qubits. Useful for debugging classical code within a quantum program, as well as estimating resources required.
- [Toffoli simulator](#): A special-purpose quantum simulator that can be used with millions of qubits, but only for programs with a restricted set of quantum operations - X, CNOT, and multi-controlled X.

# Q# quantum programming language

4/5/2021 • 2 minutes to read • [Edit Online](#)

Everything you need to know about the Q# programming language is detailed in the [Q# language guide](#). Like anything else, our [language design process](#) is open source and we welcome contributions. For more background about the foundations and motivation behind Q#, see [Why do we need Q#?](#).

Q# is shipped as part of the Quantum Development Kit (QDK) For a quick overview, check out [What is the QDK?](#).

## What is a quantum program?

A quantum program can be seen as a particular set of classical subroutines which, when called, perform a computation by interacting with a quantum system; a program written in Q# does not directly model the quantum state, but rather describes how a classical control computer interacts with qubits. This allows us to be entirely agnostic about what a quantum state even *is* on each target machine, which might have different interpretations depending on the machine.

An important consequence of that is that Q# has no ability to introspect into the state of a qubit or other properties of quantum mechanics directly, which guarantees that a Q# program can be physically executed on a quantum computer. Instead, a program can call operations such as [Measure](#) to extract classical information from a qubit.

Once allocated, a qubit can be passed to operations and functions, also referred to as *callables*. In some sense, this is all that a Q# program can do with a qubit; Any direct actions on state of a qubit are all defined by *intrinsic* callables such as [X](#) and [H](#) - that is, callables whose implementations are not defined within Q# but are instead defined by the target machine. What these operations actually *do* is only made concrete by the target machine we use to run the particular Q# program.

For example, if running the program on our [full-state simulator](#), the simulator performs the corresponding mathematical operations to the simulated quantum system. But looking toward the future, when the target machine is a real quantum computer, calling such operations in Q# will direct the quantum computer to perform the corresponding *real* operations on the *real* quantum system (for example, precisely timed laser pulses).

A Q# program recombines these operations as defined by a target machine to create new, higher-level operations to express quantum computation. In this way, Q# makes it easy to express the logic underlying quantum and hybrid quantum–classical algorithms, while also being general with respect to the structure of a target machine or simulator.

A simple example is the following program, which allocates one qubit in the  $\lvert \text{ket}0 \rangle$  state, then applies a Hadamard operation [H](#) to it and measures the result in the [Pauliz](#) basis.

```
@EntryPoint()
operation MeasureOneQubit() : Result {
    // The following using block creates a fresh qubit and initializes it
    // in the |0⟩ state.
    use qubit = Qubit();
    // We apply a Hadamard operation H to the state, thereby preparing the
    // state 1 / sqrt(2) (|0⟩ + |1⟩).
    H(qubit);
    // Now we measure the qubit in Z-basis.
    let result = M(qubit);
    // As the qubit is now in an eigenstate of the measurement operator,
    // we reset the qubit before releasing it.
    if result == One { X(qubit); }
    // Finally, we return the result of the measurement.
    return result;
}
```

Our [Quantum Katas](#) give a good introduction on [Quantum Computing Concepts](#) such as common quantum operations and how to manipulate qubits. More examples can also be found in [Intrinsic Operations and Functions](#).

# Ways to run a Q# program

6/8/2021 • 21 minutes to read • [Edit Online](#)

One of the Quantum Development Kit's greatest strengths is its flexibility across platforms and development environments. However, this also means that new Q# users may find themselves confused or overwhelmed by the numerous options found in the [install guide](#). On this page, we explain what happens when a Q# program is run, and compare the different ways in which users can do so.

A primary distinction is that Q# can be run:

- as a standalone application, where Q# is the only language involved and the program is invoked directly. Two methods actually fall in this category:
  - the command-line interface
  - Q# Jupyter Notebooks
- with an additional *host program*, written in Python or a .NET language (for example, C# or F#), which then invokes the program and can further process returned results.

To best understand these processes and their differences, we consider a simple Q# program and compare the ways it can be run.

## Basic Q# program

A basic quantum program might consist of preparing a qubit in an equal superposition of states  $\lvert\text{ket}\{0\}\rangle$  and  $\lvert\text{ket}\{1\}\rangle$ , measuring it, and returning the result, which will be randomly either one of these two states with equal probability. Indeed, this process is at the core of the [quantum random number generator](#) quickstart.

In Q#, this would be performed by the following code:

```
use q = Qubit();    // allocates qubit for use (automatically in |0>)
H(q);                // puts qubit in superposition of |0> and |1>
return MResetZ(q);  // measures qubit, returns result (and resets it to |0> before deallocation)
```

However, this code alone can't be run by Q#. For that, it needs to make up the body of an [operation](#), which is then run when called---either directly or by another operation. Hence, you can write an operation of the following form:

```
operation MeasureSuperposition() : Result {
    use q = Qubit();
    H(q);
    return MResetZ(q);
}
```

You have defined an operation, `MeasureSuperposition`, which takes no inputs and returns a value of type `Result`.

In addition to operations, Q# also allows to encapsulate deterministic computations into functions. Aside from the determinism guarantee that implies that computations that act on qubits need to be encapsulated into operations rather than functions, there is little difference between operations and function. We refer to them collectively as *callables*.

### Callable defined in a Q# file

The callable is precisely what's called and run by Q#. However, it requires a few more additions to comprise a full \*.qs Q# file.

All Q# types and callables (both those you define and those intrinsic to the language) are defined within *namespaces*, which provide each a full name that can then be referenced.

For example, the `H` and `MResetZ` operations are found in the `Microsoft.Quantum.Intrinsic` and `Microsoft.Quantum.Measurement` namespaces (part of the [Q# Standard Libraries](#)). As such, they can always be called via their *full names*, `Microsoft.Quantum.Intrinsic.H(<qubit>)` and `Microsoft.Quantum.Measurement.MResetZ(<qubit>)`, but always doing this would lead to very cluttered code.

Instead, `open` statements allow callables to be referenced with more concise shorthand, as we've done in the operation body above. The full Q# file containing our operation would therefore consist of defining our own namespace, opening the namespaces for those callables our operation uses, and then our operation:

```
namespace NamespaceName {
    open Microsoft.Quantum.Intrinsic;      // for the H operation
    open Microsoft.Quantum.Measurement;    // for MResetZ

    operation MeasureSuperposition() : Result {
        use q = Qubit();
        H(q);
        return MResetZ(q);

    }
}
```

#### NOTE

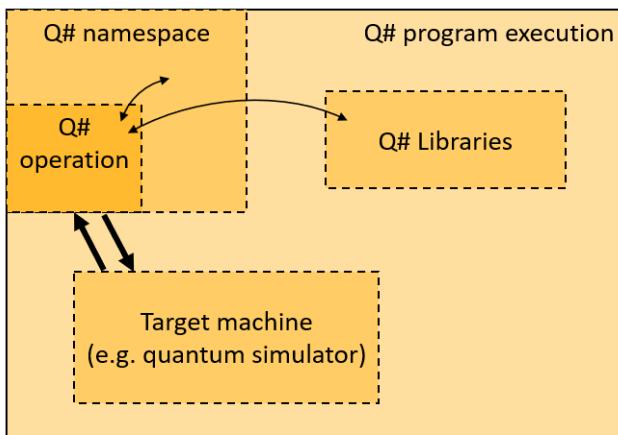
Namespaces can also be *aliased* when opened, which can be helpful if callable/type names in two namespaces conflict. For example, we could instead use `open Microsoft.Quantum.Intrinsic as NamespaceWithH;` above, and then call `H` via `NamespaceWithH.H(<qubit>)`.

#### NOTE

One exception to all of this is the `Microsoft.Quantum.Core` namespace, which is always automatically opened. Therefore, callables like `Length` can always be used directly.

## Running on target machines

Now the general run model of a Q# program becomes clear.



Firstly, the specific callable to be run has access to any other callables and types defined in the same namespace. It also access those from any of the [Q# libraries](#), but those must be referenced either via their full name, or through the use of `open` statements described above.

The callable itself is then run on a [target machine](#). Such target machines can be actual quantum hardware or the multiple simulators available as part of the QDK. For our purposes here, the most useful target machine is an instance of the [full-state simulator](#), `QuantumSimulator`, which calculates the program's behavior as if it were being run on a noise-free quantum computer.

So far, we've described what happens when a specific Q# callable is being run. Regardless of whether Q# is used in a standalone application or with a host program, this general process is more or less the same---hence the QDK's flexibility. The differences between the ways of calling into the Quantum Development Kit therefore reveal themselves in *how* that Q# callable is called to be run, and in what manner any results are returned. More specifically, the differences revolve around:

- Indicating which Q# callable is to be run
- How potential callable arguments are provided
- Specifying the target machine on which to run it
- How any results are returned

First, we discuss how this is done with the Q# standalone application from the command prompt, and then proceed to using Python and C# host programs. We reserve the standalone application of Q# Jupyter Notebooks for last, because unlike the first three, its primary functionality does not center around a local Q# file.

#### NOTE

Although we don't illustrate it in these examples, one commonality between the run methods is that any messages printed from inside the Q# program (by way of `Message` or `DumpMachine`, for example) will typically always be printed to the respective console.

## Q# from the command prompt

One of the easiest ways to get started writing Q# programs is to avoid worrying about separate files and a second language altogether. Using Visual Studio Code or Visual Studio with the QDK extension allows for a seamless work flow in which we run Q# callables from only a single Q# file.

For this, we will ultimately run the program by entering

```
dotnet run
```

at the command prompt. The simplest workflow is when the terminal's directory location is the same as the Q# file, which can be easily handled alongside Q# file editing by using the integrated terminal in VS Code, for example. However, the `dotnet run` command accepts numerous options, and the program can also be run from a different location by simply providing `--project <PATH>` with the location of the Q# file.

### Add entry point to Q# file

Most Q# files will contain more than one callable, so naturally we need to let the compiler know *which* callable to run when we provide the `dotnet run` command. This is done with a simple change to the Q# file itself: - add a line with `@EntryPoint()` directly preceding the callable.

Our file from above would therefore become

```

namespace NamespaceName {
    open Microsoft.Quantum.Intrinsic;      // for the H operation
    open Microsoft.Quantum.Measurement;    // for MResetZ

    @EntryPoint()
    operation MeasureSuperposition() : Result {
        use q = Qubit();
        H(q);
        return MResetZ(q);

    }
}

```

Now, a call of `dotnet run` from the command prompt leads to `MeasureSuperposition` being run, and the returned value is then printed directly to the terminal. So, you will see either `One` or `Zero` printed.

Note that it doesn't matter if you have more callables defined below it, only `MeasureSuperposition` will be run. Additionally, it's no problem if your callable includes [documentation comments](#) before its declaration, the `@EntryPoint()` attribute can be simply placed above them.

## Callable arguments

So far, we've only considered an operation that takes no inputs. Suppose we wanted to perform a similar operation, but on multiple qubits---the number of which is provided as an argument. Such an operation could be written as

```

operation MeasureSuperpositionArray(n : Int) : Result[] {
    use qubits = Qubit[n];           // allocate a register of n qubits
    ApplyToEach(H, qubits);          // apply H to each qubit in the register
    return ForEach(MResetZ, qubits); // perform MResetZ on each qubit, returns the resulting array
}

```

where the returned value is an array of the measurement results. Note that `ApplyToEach` and `ForEach` are in the `Microsoft.Quantum.Canon` and `Microsoft.Quantum.Arrays` namespaces, requiring additional `open` statements for each.

If we move the `@EntryPoint()` attribute to precede this new operation (note there can only be one such line in a file), attempting to run it with simply `dotnet run` results in an error message which indicates what additional command-line options are required, and how to express them.

The general format for the command line is actually `dotnet run [options]`, and callable arguments are provided there. In this case, the argument `n` is missing, and it shows that we need to provide the option `-n <n>`. To run `MeasureSuperpositionArray` for `n=4` qubits, we therefore use

```
dotnet run -n 4
```

yielding an output similar to

```
[Zero,One,One,One]
```

This of course extends to multiple arguments.

## NOTE

Argument names defined in `camelCase` are slightly altered by the compiler to be accepted as Q# inputs. For example, if instead of `n`, we used the name `numQubits` above, then this input would be provided in the command line via `--num-qubits 4` instead of `-n 4`.

The error message also provides other options which can be used, including how to change the target machine.

## Different target machines

As the outputs from our operations thus far have been the expected results of their action on real qubits, it's clear that the default target machine from the command line is the full-state quantum simulator, `QuantumSimulator`. However, we can instruct callables to be run on a specific target machine with the option `--simulator` (or the shorthand `-s`).

For example, we could run it on `ResourcesEstimator`:

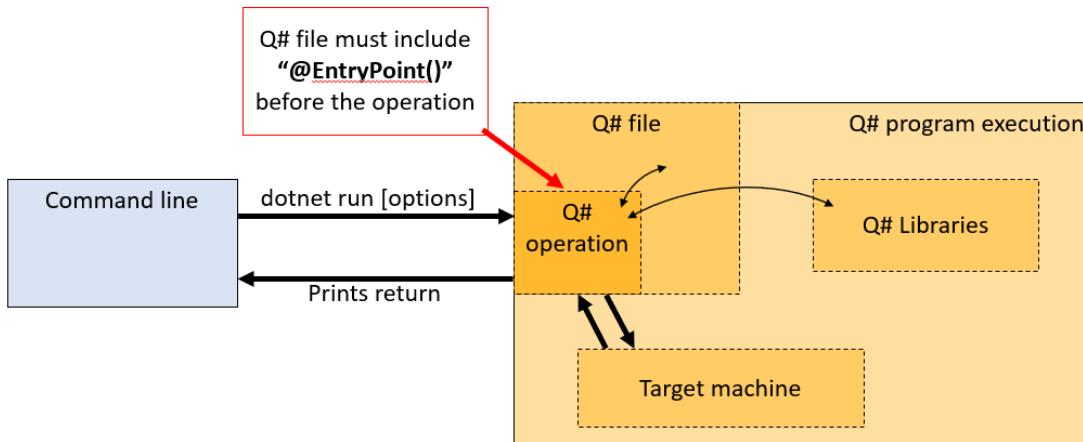
```
dotnet run -n 4 -s ResourcesEstimator
```

The printed output is then

| Metric        | Sum |
|---------------|-----|
| CNOT          | 0   |
| QubitClifford | 4   |
| R             | 0   |
| Measure       | 4   |
| T             | 0   |
| Depth         | 0   |
| Width         | 4   |
| BorrowedWidth | 0   |

For details on what these metrics indicate, see [Resource estimator: metrics reported](#).

## Command line run summary



## Non-Q# `dotnet run` options

As we briefly mentioned above with the `--project` option, the `dotnet run` command also accepts options unrelated to the Q# callable arguments. If providing both kinds of options, the `dotnet` -specific options must be provided first, followed by a delimiter `--`, and then the Q#-specific options. For example, specifying a path along with a number qubits for the operation above would be run via `dotnet run --project <PATH> -- -n <n>`.

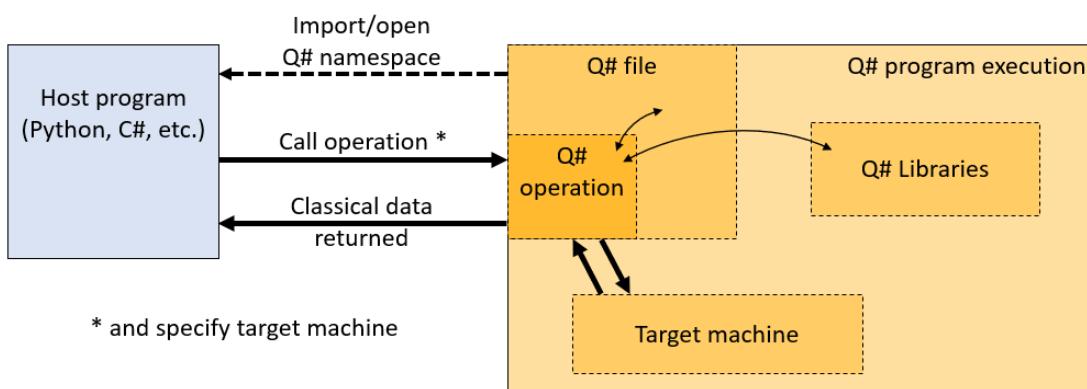
## Q# with host programs

With our Q# file in hand, an alternative to calling an operation or function directly from the command prompt is to use a *host program* in another classical language. Specifically, this can be done with either Python or a .NET language such as C# or F# (for the sake of brevity we will only detail C# here). A little more setup is required to enable the interoperability, but those details can be found in the [install guides](#).

In a nutshell, the situation now includes a host program file (for example, `*.py` or `*.cs`) in the same location as our Q# file. It's now the *host* program that gets run, and while it is running, it can call specific Q# operations and functions from the Q# file. The core of the interoperability is based on the Q# compiler making the contents of the Q# file accessible to the host program so that they can be called.

One of the main benefits of using a host program is that the classical data returned by the Q# program can then be further processed in the host language. This could consist of some advanced data processing (for example, something that can't be performed internally in Q#), and then calling further Q# actions based on those results, or something as simple as plotting the Q# results.

The general scheme is shown here, and we discuss the specific implementations for Python and C# below. A sample using an F# host program can be found at the [.NET interoperability samples](#).



### NOTE

The `@EntryPoint()` attribute used for Q# applications cannot be used with host programs. An error will be raised if it is present in the Q# file being called by a host.

To work with different host programs, there are no changes required to a `*.qs` Q# file. The following host program implementations all work with the same Q# file:

```

namespace NamespaceName {
    open Microsoft.Quantum.Intrinsic;      // contains H
    open Microsoft.Quantum.Measurement;    // MResetZ
    open Microsoft.Quantum.Canon;          // ApplyToEach
    open Microsoft.Quantum.Arrays;         // ForEach

    operation MeasureSuperposition() : Result {
        use q = Qubit();
        H(q);
        return MResetZ(q);
    }

    operation MeasureSuperpositionArray(n : Int) : Result[] {
        use qubits = Qubit[n];
        ApplyToEach(H, qubits);
        return ForEach(MResetZ, qubits);
    }
}

```

Select the tab corresponding to your host language of interest.

- [Python](#)
- [C#](#)

A Python host program is constructed as follows:

1. Import the `qsharp` module, which registers the module loader for Q# interoperability. This allows Q# namespaces to appear as Python modules, from which we can "import" Q# callables. Note that it is technically not the Q# callables themselves which are imported, but rather Python stubs which allow calling into them. These behave as objects of Python classes. We use methods on these objects to specify the target machines that we send the operation to when running the program.
2. Import those Q# callables which we will directly invoke---in this case, `MeasureSuperposition` and `MeasureSuperpositionArray`.

```

import qsharp
from NamespaceName import MeasureSuperposition, MeasureSuperpositionArray

```

With the `qsharp` module imported, you can also import callables directly from the Q# library namespaces.

3. Alongside regular Python code, you can now run those callables on [specific target machines](#), and assign their return values to variables for further use:

```

random_bit = MeasureSuperposition.simulate()
print(random_bit)

```

### Specifying target machines

Running Q# operations on a specific target machine is done by invoking Python methods directly on the imported operation object. Thus, there is no need to create an object for the run target (such as a simulator). Instead, invoke one of the following methods to run the imported Q# operation:

- `.simulate(<args>)` uses the [full state simulator](#) to simulate the operation for an ideal quantum computer ([api reference for .simulate\(\)](#))
- `.estimate_resources(<args>)` uses the [resources estimator](#) to compute various quantum resources required

by the program ([api reference for .estimate\\_resources\(\)](#))

- `.toffoli_simulate(<args>)` uses the [Toffoli simulator](#) to provide a more efficient simulation method for a restricted class of quantum programs ([api reference for .toffoli\\_simulate\(\)](#))

For more information about local target machines, see [Quantum simulators](#).

#### Passing inputs to Q#

Arguments for the Q# callable should be provided in the form of a keyword argument, where the keyword is the argument name in the Q# callable definition. That is, `MeasureSuperpositionArray.simulate(n=4)` is valid, whereas `MeasureSuperpositionArray.simulate(4)` would throw an error.

Therefore, the Python host program

```
import qsharp
from NamespaceName import MeasureSuperposition, MeasureSuperpositionArray

single_qubit_result = MeasureSuperposition.simulate()
single_qubit_resources = MeasureSuperposition.estimate_resources()

multi_qubit_result = MeasureSuperpositionArray.simulate(n=4)
multi_qubit_resources = MeasureSuperpositionArray.estimate_resources(n=4)

print('Single qubit:\n' + str(single_qubit_result))
print(single_qubit_resources)

print('\nMultiple qubits:\n' + str(multi_qubit_result))
print(multi_qubit_resources)
```

results in an output like the following:

```
Single qubit:
1
{'CNOT': 0, 'QubitClifford': 1, 'R': 0, 'Measure': 1, 'T': 0, 'Depth': 0, 'Width': 1, 'BorrowedWidth': 0}

Multiple qubits:
[0, 1, 1, 1]
{'CNOT': 0, 'QubitClifford': 4, 'R': 0, 'Measure': 4, 'T': 0, 'Depth': 0, 'Width': 4, 'BorrowedWidth': 0}
```

#### Using Q# code from other projects or packages

By default, the `import qsharp` command loads all of the `.qs` files in the current folder and makes their Q# operations and functions available for use from inside the Python script.

To load Q# code from another folder, the [qsharp.projects API](#) can be used to add a reference to a `.csproj` file for a Q# project (that is, a project that references `Microsoft.Quantum.Sdk`). This command will compile any `.qs` files in the folder containing the `.csproj` and its subfolders. It will also recursively load any packages referenced via `PackageReference` or Q# projects referenced via `ProjectReference` in that `.csproj` file.

As an example, the following Python code imports an external project, referencing its path relative to the current folder, and invokes one of its Q# operations:

```
import qsharp
qsharp.projects.add("../qrng/QRNG.csproj")
from QRNG import SampleQuantumRandomNumberGenerator
print(f"QRNG result: {SampleQuantumRandomNumberGenerator.simulate()}")
```

This results in output like the following:

```
Adding reference to project: ../qrng/QRNG.csproj
QRNG result: 0
```

To load external packages containing Q# code, use the [qsharp.packages API](#).

If the Q# code in the current folder depends on external projects or packages, you may see errors when running `import qsharp`, since the dependencies have not yet been loaded. To load required external packages or Q# projects during the `import qsharp` command, ensure that the folder with the Python script contains a `.csproj` file which references `Microsoft.Quantum.Sdk`. In that `.csproj`, add the property `<IQSharpLoadAutomatically>true</IQSharpLoadAutomatically>` to the `<PropertyGroup>`. This will instruct IQ# to recursively load any `ProjectReference` or `PackageReference` items found in that `.csproj` during the `import qsharp` command.

For example, here is a simple `.csproj` file that causes IQ# to automatically load the `Microsoft.Quantum.Chemistry` package:

```
<Project Sdk="Microsoft.Quantum.Sdk/0.17.2105143879">
  <PropertyGroup>
    <OutputType>Library</OutputType>
    <TargetFramework>netstandard2.1</TargetFramework>
    <IQSharpLoadAutomatically>true</IQSharpLoadAutomatically>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Quantum.Chemistry" Version="0.17.2105143879" />
  </ItemGroup>
</Project>
```

#### NOTE

Currently this custom `<IQSharpLoadAutomatically>` property is required by Python hosts, but in the future, this may become the default behavior for a `.csproj` file located in the same folder as the Python script.

#### NOTE

Currently the `<QsharpCompile>` setting in the `.csproj` is ignored by Python hosts, and all `.qs` files in the folder of the `.csproj` (including subfolders) are loaded and compiled. Support for `.csproj` settings will be improved in the future (see [iqsharp#277](#) for more details).

## Q# Jupyter Notebooks

Q# Jupyter Notebooks make use of the IQ# kernel, which allows you to define, compile, and run Q# callables in a single notebook---all alongside instructions, notes, and other content. This means that while it is possible to import and use the contents of `*.qs` Q# files, they are not necessary in the run model.

Here, we will detail how to run the Q# operations defined above, but a more broad introduction to using Q# Jupyter Notebooks is provided at [Intro to Q# and Jupyter Notebooks](#).

### Defining operations

In a Q# Jupyter Notebook, you enter Q# code just as we would from inside the namespace of a Q# file.

So, we can enable access to callables from the [Q# standard libraries](#) with `open` statements for their respective namespaces. Upon running a cell with such a statement, the definitions from those namespaces are available throughout the workspace.

## NOTE

Callable from `Microsoft.Quantum.Intrinsic` and `Microsoft.Quantum.Canon` (for example, `H` and `ApplyToEach`) are automatically available to operations defined within cells in Q# Jupyter Notebooks. However, this is not true for code brought in from external Q# source files (a process shown at [Intro to Q# and Jupyter Notebooks](#)).

Similarly, defining operations requires only writing the Q# code and running the cell.

```
In [8]: ┌ open Microsoft.Quantum.Measurement; // for MultiM

operation MeasureSuperposition() : Result {
    use q = Qubit() {
        H(q);
        return M(q);
    }
}

operation MeasureSuperpositionArray(n : Int) : Result[] {
    use qubits = Qubit[n] {
        ApplyToEach(H, qubits);
        return MultiM(qubits);
    }
}
```

Out[8]: • MeasureSuperposition  
• MeasureSuperpositionArray

The output then lists those operations, which can then be called from future cells.

## Target machines

The functionality to run operations on specific target machines is provided via [IQ# Magic Commands](#). For example, `%simulate` makes use of the `QuantumSimulator`, and `%estimate` uses the `ResourcesEstimator`:

```
In [2]: ┌ %simulate MeasureSuperposition
```

Out[2]: One

```
In [3]: ┌ %estimate MeasureSuperposition
```

Metric	Sum	Max
CNOT	0	0
QubitClifford	1	1
R	0	0
Measure	1	1
T	0	0
Depth	0	0
Width	1	1
BorrowedWidth	0	0

## Passing inputs to functions and operations

To pass inputs to the Q# operations, the arguments can be passed as `key=value` pairs to the run magic command. So, to run `MeasureSuperpositionArray` with four qubits, we can run

```
%simulate MeasureSuperpositionArray n=4 :
```

```
In [4]: ┌ %simulate MeasureSuperpositionArray n=4
```

```
Out[4]:
```

- One
- One
- Zero
- One

This pattern can be used similarly with `%estimate` and other run commands.

#### NOTE

Passing callables in a similar manner with run commands such as `%simulate` or `%estimate` is not possible.

### Using Q# code from other projects or packages

By default, a Q# Jupyter Notebook loads all of the `.qs` files in the current folder and makes their Q# operations and functions available for use from inside the notebook. The `%who` magic command lists all currently-available Q# operations and functions.

To load Q# code from another folder, the `%project` magic command can be used to add a reference to a `.csproj` file for a Q# project (that is, a project that references `Microsoft.Quantum.Sdk`). This command will compile any `.qs` files in the folder containing the `.csproj` (and subfolders). It will also recursively load any packages referenced via `PackageReference` or Q# projects referenced via `ProjectReference` in that `.csproj` file.

As an example, the following cells simulate a Q# operation from an external project, where the project path is referenced relative to the current folder:

```
In [5]: ┌ %project "../qrng/Qrng.csproj"
```

```
Adding reference to project: ../qrng/Qrng.csproj
```

```
Reloading workspace: done!
```

```
Out[5]:
```

- C:\qdk\qrng\Qrng.csproj

```
In [6]: ┌ %simulate Qrng.SampleRandomNumber
```

```
Sampling a random number between 0 and 50:
```

```
Out[6]: 44
```

To load external packages containing Q# code, use the `%package` magic command. Loading a package will also make available any custom magic commands or display encoders that are contained in any assemblies that are part of the package.

To load external packages or Q# projects at notebook initialization time, ensure that the notebook folder contains a `.csproj` file which references `Microsoft.Quantum.Sdk`. In that `.csproj`, add the property `<IQSharpLoadAutomatically>true</IQSharpLoadAutomatically>` to the `<PropertyGroup>`. This will instruct IQ# to recursively load any `ProjectReference` or `PackageReference` items found in that `.csproj` at notebook load time.

For example, here is a simple `.csproj` file that causes IQ# to automatically load the `Microsoft.Quantum.Chemistry` package:

```
<Project Sdk="Microsoft.Quantum.Sdk/0.17.2105143879">
  <PropertyGroup>
    <OutputType>Library</OutputType>
    <TargetFramework>netstandard2.1</TargetFramework>
    <IQSharpLoadAutomatically>true</IQSharpLoadAutomatically>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Quantum.Chemistry" Version="0.17.2105143879" />
  </ItemGroup>
</Project>
```

#### NOTE

Currently this custom `<IQSharpLoadAutomatically>` property is required by Q# Jupyter Notebook hosts, but in the future, this may become the default behavior for a `.csproj` file located in the same folder as the notebook file.

#### NOTE

Currently the `<QsharpCompile>` setting in the `.csproj` is ignored by Q# Jupyter Notebook hosts, and all `.qs` files in the folder of the `.csproj` (including subfolders) are loaded and compiled. Support for `.csproj` settings will be improved in the future (see [iqsharp#277](#) for more details).

# Testing and debugging

6/30/2021 • 16 minutes to read • [Edit Online](#)

As with classical programming, it is essential to be able to check that quantum programs act as intended, and to be able to diagnose incorrect behavior. Unlike classical programming, though, observing the state of a quantum system and tracking the behavior of a quantum program is not always easy. In this section, we cover the tools offered by the Quantum Development Kit for testing and debugging quantum programs.

## Unit Tests

One common approach to testing classical programs is to write small programs called *unit tests*, which run code in a library and compare its output to some expected output. For example, you can ensure that `square(2)` returns `4` since you know *a priori* that  $2^2 = 4$ .

Q# supports creating unit tests for quantum programs, and which can run as tests within the [xUnit](#) unit testing framework.

### Creating a Test Project

- [Visual Studio 2019](#)
- [Command Line / Visual Studio Code](#)

Open Visual Studio 2019. Go to the **File** menu and select **New > Project....**. In the upper right corner, search for `Q#`, and select the **Q# Test Project** template.

Your new project will have two files in it, a code file and a project file. `Tests.qs` provides a convenient place to define new Q# unit tests, while the `.csproj` file contains configuration parameters needed to build the project.

Initially, the code file contains one sample unit test `AllocateQubit` which checks that a newly allocated qubit is in the `|0⟩` state and prints a message:

```
namespace TestProject {
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Diagnostics;
    open Microsoft.Quantum.Intrinsic;

    @Test("QuantumSimulator")
    operation AllocateQubit () : Unit {
        use qubit = Qubit();
        AssertMeasurement([PauliZ], [qubit], Zero, "Newly allocated qubit must be in the |0⟩ state.");
        Message("Test passed");
    }
}
```

Any Q# operation or function that takes an argument of type `Unit` and returns `Unit` can be marked as a unit test via the `@Test("...")` attribute. In the previous example, the argument to that attribute, `"QuantumSimulator"`, specifies the target on which the test runs. A single test can run on multiple targets. For example, add an attribute `@Test("ResourcesEstimator")` before `AllocateQubit`.

```

namespace TestProject {
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Diagnostics;
    open Microsoft.Quantum.Intrinsic;

    @Test("QuantumSimulator")
    @Test("ResourcesEstimator")
    operation AllocateQubit () : Unit {
        ...
    }
}

```

After saving the file you will see two unit tests when running the tests: one where `AllocateQubit` runs on the `QuantumSimulator`, and one where it runs in the `ResourcesEstimator`.

The Q# compiler recognizes the built-in targets `"QuantumSimulator"`, `"ToffoliSimulator"`, and `"ResourcesEstimator"` as valid run targets for unit tests. It is also possible to specify any fully qualified simulator name to define a custom run target.

Besides the code file, the test project template includes the `.csproj` file with the following contents:

```

<Project Sdk="Microsoft.Quantum.Sdk/0.16.2105140472">

<PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <IsPackable>false</IsPackable>
</PropertyGroup>

<ItemGroup>
    <PackageReference Include="Microsoft.Quantum.Xunit" Version="0.16.2105140472" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.4.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
    <DotNetCliToolReference Include="dotnet-xunit" Version="2.3.1" />
</ItemGroup>

</Project>

```

The first line specifies the version number of the Microsoft Quantum Development Kit used to build the application. Note that the exact version numbers you see in this file will depend on your installation.

The `TargetFramework` generally contains either of two values for Q# applications depending on the project type: `netcoreapp3.1` for executable and test projects, and `netstandard2.1` for libraries. Next, the `IsPackable` parameter is set to false (true when omitted). It determines whether a NuGet package is generated from this project when the `dotnet pack` command is run.

Finally, the file lists NuGet package dependencies inside the `<ItemGroup>` tag. `xUnit` is a popular testing framework for the .NET framework, which Q# test projects make use of (third reference in the list). The `Microsoft.Quantum.Xunit` and `Microsoft.NET.Test.Sdk` packages are used to expose Q# constructs to xUnit and build .NET test projects. In order to run any tests, xUnit also requires a unit test runner. Both the `xunit.runner.visualstudio` and the `dotnet-xunit` runners are required to run tests from the command line, while the former is sufficient when running tests from within Visual Studio. Make sure to include the appropriate package references if you are creating a test project manually or are converting from a regular project.

## Running Q# Unit Tests

- [Visual Studio 2019](#)
- [Command Line / Visual Studio Code](#)

As a one-time per-solution setup, go to the **Test** menu and select **Test Settings > Default Processor Architecture > X64**.

#### TIP

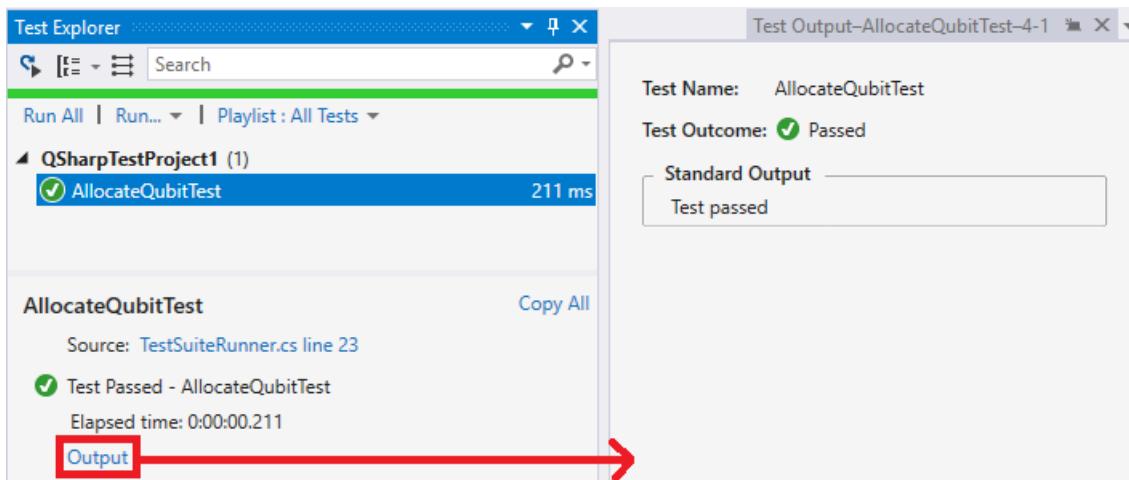
The default processor architecture setting for Visual Studio is stored in the solution options (.suo) file for each solution. If you delete this file, then you need to select **X64** as your processor architecture again.

Build the project, open the **Test** menu, and select **Windows > Test Explorer**. **AllocateQubit** displays in the list of tests in the **Not Run Tests** group. Select **Run All** or run this individual test.

The intrinsic function **Message function** has type `(String -> Unit)` and enables the creation of diagnostic messages.

- [Visual Studio 2019](#)
- [Command Line / Visual Studio Code](#)

After you run a test in Test Explorer and click the test name, a panel displays with information about test run: Pass/fail status, elapsed time, and a link to the output. Click **Output** to open the test output in a new window.



## Facts and Assertions

Because functions in Q# have no *logical* side effects, you can never observe, from within a Q# program, any other kinds of effects from running a function whose output type is the empty tuple `()`. That is, a target machine can choose not to run any function which returns `()` with the guarantee that this omission will not modify the behavior of any following Q# code. This behavior makes functions returning `()` (such as `Unit`) a useful tool for embedding assertions and debugging logic into Q# programs.

Let's consider a simple example:

```
namespace DebuggingFactsTest {

    @EntryPoint()
    function PositivityFact(value : Int) : Unit {

        if value <= 0 {

            fail "Expected a positive number.";
        }
    }
}
```

Here, the keyword `fail` indicates that the computation should not proceed, and raises an exception in the target machine running the Q# program. By definition, a failure of this kind cannot be observed from within Q#, as the target machine no longer runs the Q# code after reaching a `fail` statement. Thus, if we proceed past a call to `PositivityFact`, we can be assured that its input was positive.

Note that we can implement the same behavior as `PositivityFact` using the `Fact` function from the [Microsoft.Quantum.Diagnostics namespace](#):

```
Fact(value > 0, "Expected a positive number.");
```

*Assertions*, on the other hand, are used similarly to facts but may depend on the state of the target machine. Correspondingly, they are defined as operations, whereas facts are defined as functions (as in the previous example).

The [prelude](#), building on these ideas, offers two especially useful assertions, `AssertMeasurement` operation and `AssertMeasurementProbability` operation both modeled as operations onto `()`. These assertions each take a Pauli operator describing a particular measurement of interest, a quantum register on which a measurement is performed, and a hypothetical outcome. Target machines which work by simulation are not bound by [the no-cloning theorem](#), and can perform such measurements without disturbing the register that passes to such assertions. A simulator can then, similar to the `PositivityFact` function previous, stop computation if the hypothetical outcome is not observed in practice:

```
namespace AssertionsTest {

    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Diagnostics;

    @EntryPoint()
    operation Assert() : Unit {
        use register = Qubit();
        H(register);
        AssertMeasurement([PauliX], [register], Zero, "The state of the quantum register is not |+>");
        ResetAll([register]);

        // Even though we do not have access to states in Q#,
        // we know by the anthropic principle that the state
        // of register at this point is |+>.
    }
}
```

On physical quantum hardware, where the no-cloning theorem prevents examination of a quantum state, the `AssertMeasurement` and `AssertMeasurementProbability` operations simply return `()` with no other effect.

The [Microsoft.Quantum.Diagnostics namespace](#) provides several more functions of the `Assert` family, with which you can check more advanced conditions.

## Dump Functions

Just like a real quantum computation, Q# does not allow us to directly access qubit states. However, the [Microsoft.Quantum.Diagnostics namespace](#) offers three functions that can dump into a file the current status of the target machine and can provide valuable insight for debugging and learning when used in conjunction with the full state simulator: `DumpOperation` operation, `DumpMachine` function and `DumpRegister` function. The generated output of each depends on the target machine.

## DumpOperation

Suppose you are implementing a quantum gate described by a matrix. You've written a Q# operation and want to check that it implements exactly the unitary matrix you're looking for. The [DumpOperation operation](#) takes an operation that acts on an array of qubits as a parameter (if your operation acts on a single qubit, like most intrinsic gates, or on a mix of individual qubits and qubit arrays, you'll need to write a wrapper for it to use `DumpOperation` on it), and prints a matrix implemented by this operation. Let's take the CNOT gate as an example.

```
namespace DumpOperationTest {

    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Diagnostics;

    @EntryPoint()
    operation DumpCnot() : Unit {
        DumpOperation(2, ApplyToFirstTwoQubitsCA(CNOT,_));
    }
}
```

Calling `DumpOperation` will print the following matrix,

```
Real:
[[1, 0, 0, 0],
 [0, 0, 0, 1],
 [0, 0, 1, 0],
 [0, 1, 0, 0]]
Imag:
[[0, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]]
```

### NOTE

`DumpOperation` and the rest of Dump functions use the little-endian encoding for converting basis states to the indices of matrix elements. Thus, the second column of the CNOT matrix corresponds to the input state  $|10_{LE}\rangle = |10\rangle$ , which the CNOT gate converts to  $|11\rangle = |3_{LE}\rangle$ . Similarly, the input state  $|20_{LE}\rangle = |01\rangle$ .

## DumpMachine

The full-state quantum simulator distributed as part of the Quantum Development Kit writes into the file the [wave function](#) of the entire quantum system, as a one-dimensional array of complex numbers, in which each element represents the amplitude of the probability of measuring the computational basis state  $\lvert \text{ket}\{n\} \rangle$ , where  $\lvert \text{ket}\{n\} \rangle = \lvert \text{ket}\{b_{n-1}...b_1b_0\} \rangle$  for bits  $\{b_i\}$ . For example, consider the following operation that allocates two qubits and prepares an uneven superposition state on them:

```

namespace MultiDumpMachineTest {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Diagnostics;

    @EntryPoint()
    operation MultiQubitDumpMachineDemo() : Unit {
        use qubits = Qubit[2];
        X(qubits[1]);
        H(qubits[1]);
        R1Frac(1, 2, qubits[1]);

        DumpMachine("dump.txt");

        ResetAll(qubits);
    }
}

```

The resulting quantum state of `MultiQubitDumpMachineDemo` operation is

$$\$ \$ \begin{aligned} \lvert \psi \rangle = & \frac{1}{\sqrt{2}} (\lvert 00 \rangle - \frac{1+i}{2} \lvert 10 \rangle) \end{aligned} \$ \$$$

Calling [DumpMachine function](#) on the previous quantum state generates the following output:

```

# wave function for qubits with ids (least to most significant): 0;1
| 0:  0.707107 +  0.000000 i == *****
| 1:  0.000000 +  0.000000 i ==
| 2: -0.500000 + -0.500000 i == *****
| 3:  0.000000 +  0.000000 i ==

```

The first row provides a comment with the ids of the corresponding qubits in their significant order. The rest of the rows describe the probability amplitude of measuring the basis state vector  $\lvert \psi \rangle$  in both Cartesian and polar formats. In detail for the first row:

- `| 0:` this row corresponds to the `0` computational basis state
- `0.707107 + 0.000000 i`: the probability amplitude in Cartesian format.
- `==`: the `equal` sign separates both equivalent representations.
- `*****`: A graphical representation of the magnitude, the number of `*` is proportionate to the probability of measuring this state vector.
- `[ 0.500000 ]`: the numeric value of the magnitude
- `---`: A graphical representation of the amplitude's phase (see the following output).
- `[ 0.000 rad ]`: the numeric value of the phase (in radians).

Both the magnitude and the phase are displayed with a graphical representation. The magnitude representation is straight-forward: it shows a bar of `*`, the bigger the probability the bigger the bar will be. For the phase, we show the following symbols to represent the angle based on ranges:

```

[ -π/16, π/16)    ---
[ π/16, 3π/16)    /-
[ 3π/16, 5π/16)   /
[ 5π/16, 7π/16)   +/
[ 7π/16, 9π/16)   ↑
[ 8π/16, 11π/16)  \-
[ 7π/16, 13π/16)  \
[ 7π/16, 15π/16)  +\
[15π/16, 19π/16)  ---
[17π/16, 19π/16)  -/
[19π/16, 21π/16)  /
[21π/16, 23π/16)  /+
[23π/16, 25π/16)  ↓
[25π/16, 27π/16)  -\
[27π/16, 29π/16)  \
[29π/16, 31π/16)  \+
[31π/16, π/16)    ---
```

The following examples show `DumpMachine` for some common states:

| 0

```
# wave function for qubits with ids (least to most significant): 0
| 0: 1.000000 + 0.000000 i == **** [ 1.000000 ]      --- [ 0.00000 rad ]
| 1: 0.000000 + 0.000000 i == [ 0.000000 ]
```

| 1

```
# wave function for qubits with ids (least to most significant): 0
| 0: 0.000000 + 0.000000 i == [ 0.000000 ]
| 1: 1.000000 + 0.000000 i == **** [ 1.000000 ]      --- [ 0.00000 rad ]
```

| +

```
# wave function for qubits with ids (least to most significant): 0
| 0: 0.707107 + 0.000000 i == ***** [ 0.500000 ]      --- [ 0.00000 rad ]
| 1: 0.707107 + 0.000000 i == ***** [ 0.500000 ]      --- [ 0.00000 rad ]
```

| -

```
# wave function for qubits with ids (least to most significant): 0
| 0: 0.707107 + 0.000000 i == ***** [ 0.500000 ]      --- [ 0.00000 rad ]
| 1: -0.707107 + 0.000000 i == ***** [ 0.500000 ]      --- [ 3.14159 rad ]
```

## NOTE

The id of a qubit is assigned at runtime and is not necessarily aligned with the order in which the qubit was allocated or its position within a qubit register.

- [Visual Studio 2019](#)
- [Command Line / Visual Studio Code](#)

### TIP

You can locate a qubit id in Visual Studio by putting a breakpoint in your code and inspecting the value of a qubit variable, for example:

Locals	
Name	Value
↳ this	{Operation}
↳ _in	{0}
↳ register1	Count = 2
↳ q	Count = 2
↳ register2	Count = 4
↳ [0]	{q:3}
↳ Id	3
↳ Probability	0
↳ [1]	{q:2}
↳ Id	2
↳ Probability	0
↳ [2]	{q:4}
↳ Id	4
↳ Probability	0
↳ [3]	{q:5}
↳ Id	5
↳ Probability	0

the qubit with index 0 on register2 has id= 3, the qubit with index 1 has id= 2 .

### DumpMachine with Jupyter Notebook

For the sake of simplicity, in the previous testing and debugging tools we displayed examples of code using a Q# standalone application at a command prompt and any IDE. However, you can use any of the development options offered by Quantum Development Kit to develop quantum computing applications in Q#. For more information, see [Set up the QDK](#).

In this example for [DumpMachine function](#), we explicitly show the development using a Q# Jupyter Notebook, as it offers more visualization tools for testing and debugging quantum programs.

1. To run `DumpMachine` on a Jupyter Notebook, open a [new Jupyter Notebook with a Q# kernel](#) and copy the following code to the first notebook cell:

```
open Microsoft.Quantum.Diagnostics;

operation MultiQubitDumpMachineDemo() : Unit {
    use qubits = Qubit[2];
    X(qubits[1]);
    H(qubits[1]);
    R1Frac(1, 2, qubits[1]);

    DumpMachine();

    ResetAll(qubits);
}
```

1. In a new cell, run the `MultiQubitDumpMachineDemo` operation on a full state quantum simulator by using the `%simulate` magic command. The `DumpMachine` call prints the information about the quantum state of the program after the Controlled Ry gate as a set of lines, one per basis state, showing their complex amplitudes, phases, and measurement probabilities.

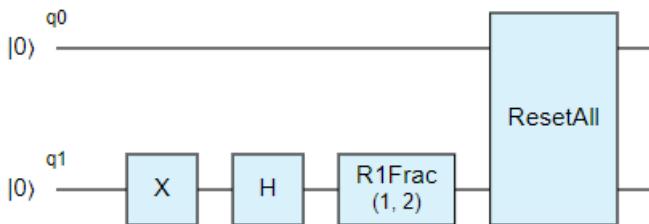
Qubit IDs	0, 1			
Basis state (little endian)	Amplitude	Meas. Pr.	Phase	
0>	0.7071 + 0.0000i	<div style="width: 70.71%; background-color: #0070C0;"></div> 50.0000%		↑
1>	0.0000 + 0.0000i	<div style="width: 0%; background-color: #C0C0C0;"></div> 0.0000%		↑
2>	-0.5000 - 0.5000i	<div style="width: 50%; background-color: #0070C0;"></div> 50.0000%		✓
3>	0.0000 + 0.0000i	<div style="width: 0%; background-color: #C0C0C0;"></div> 0.0000%		↑

### NOTE

You can use `%config (magic command)` (available only in Q# Jupyter Notebooks) to tweak the format of the `DumpMachine` output. It offers many settings that you can use in different scenarios. For example, by default `DumpMachine` uses little-endian integers to denote the basis states (the first column of the output); if you find raw bit strings easier to read, you can use `%config dump.basisStateLabelingConvention="Bitstring"` to switch.

1. Jupyter Notebooks offers the option to visualize the run of the quantum program as a quantum circuit by using `%trace (magic command)` (available only in Q# Jupyter Notebooks). This command traces one run of the Q# programs and build a circuit based on that run. This is the circuit resulting from the running of

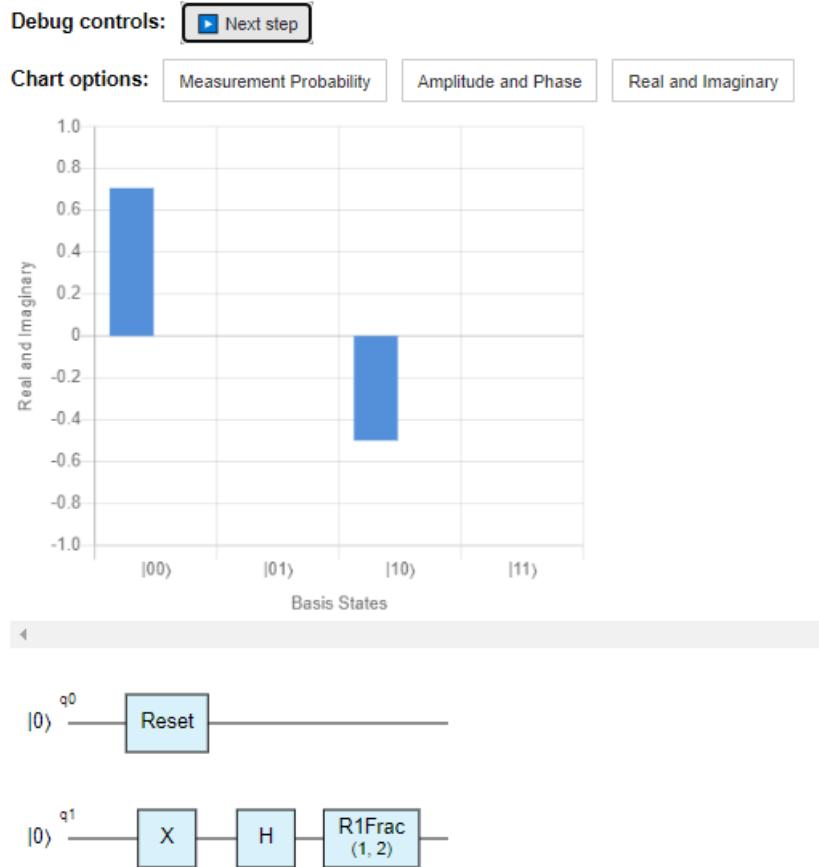
```
%trace MultiQubitDumpMachineDemo ,
```



The visualization is interactive, allowing you to click on each block to drill down to the intrinsic gates.

1. Finally, `%debug (magic command)` (available only in Q# Jupyter Notebooks) allows you to combine tracing the program execution (as a circuit) and observing the program state as it evolves at the same time. The visualization is also interactive; you can click through each of the steps until the program run is complete, and switch to observe real and imaginary components of the amplitudes, instead of measurement probabilities in the beginning of the program.

Starting debug session for MultiQubitDumpMachineDemo...



## DumpRegister

[DumpRegister function](#) works like [DumpMachine function](#), except that it also takes an array of qubits to limit the amount of information to only that relevant to the corresponding qubits.

As with [DumpMachine function](#), the information generated by [DumpRegister function](#) depends on the target machine. For the full-state quantum simulator it writes into the file the wave function up to a global phase of the quantum sub-system generated by the provided qubits in the same format as [DumpMachine function](#). For example, take again a machine with only two qubits allocated and in the quantum state `\$`begin{align} \ket{\psi} = \frac{1}{\sqrt{2}} (\ket{00} - \frac{1+i}{\sqrt{2}} \ket{10}) = -e^{-i\pi/4} (\frac{1}{\sqrt{2}} \ket{00} - \frac{1+i}{\sqrt{2}} \ket{10}) `otimes \frac{-i}{\sqrt{2}} \ket{10}`end{align}` calling [DumpRegister function](#) for `qubit[0]` generates this output:

```
# wave function for qubits with ids (least to most significant): 0
|0: -0.707107 + -0.707107 i == **** [ 1.000000 ] /      [ -2.35619 rad ]
|1: 0.000000 + 0.000000 i ==      [ 0.000000 ]
```

and calling [DumpRegister function](#) for `qubit[1]` generates this output:

```
# wave function for qubits with ids (least to most significant): 1
|0: 0.707107 + 0.000000 i == **** [ 0.500000 ] --- [ 0.00000 rad ]
|1: -0.500000 + -0.500000 i == **** [ 0.500000 ] /      [ -2.35619 rad ]
```

In general, the state of a register that is entangled with another register is a mixed state rather than a pure state. In this case, [DumpRegister function](#) outputs the following message:

```
Qubits provided (0;) are entangled with some other qubit.
```

The following example shows you how you can use both [DumpRegister function](#) and [DumpMachine function](#) in

your Q# code:

```
namespace DumpRegisterTest {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Diagnostics;

    @EntryPoint()
    operation DumpRegisterTestOp () : Unit {
        use qubits = Qubit[2];
        X(qubits[1]);
        H(qubits[1]);
        R1Frac(1, 2, qubits[1]);

        DumpMachine("dump.txt");
        DumpRegister("q0.txt", qubits[0..0]);
        DumpRegister("q1.txt", qubits[1..1]);
        ResetAll(qubits);
    }
}
```

## Debugging

On top of `Assert` and `Dump` functions and operations, Q# supports a subset of standard Visual Studio debugging capabilities: [setting line breakpoints](#), [stepping through code using F10](#), and [inspecting values of classic variables](#) are all possible when running your code on the simulator.

Debugging in Visual Studio Code leverages the debugging capabilities provided by the C# for Visual Studio Code extension powered by OmniSharp and requires installing the [latest version](#).

# Q# Language

3/29/2021 • 2 minutes to read • [Edit Online](#)

Q# is part of Microsoft's [Quantum Development Kit](#) and provides rich IDE support and tools for program visualization and analysis. Our goal is to support the development of future large-scale applications while supporting user's first efforts in that direction on current quantum hardware.

The type system permits Q# programs to safely interleave and naturally represent the composition of classical and quantum computations. A Q# program may express arbitrary classical computations based on quantum measurements that execute while qubits remain live, meaning they are not released and maintain their state. Even though the full complexity of such computations requires further hardware development, Q# programs can be targeted to run on various quantum hardware backends in [Azure Quantum](#).

Q# is a stand-alone language offering a high level of abstraction. There is no notion of a quantum state or a circuit; instead, Q# implements programs in terms of statements and expressions, much like classical programming languages. Distinct quantum capabilities (such as support for functors and control-flow constructs) facilitate expressing, for example, phase estimation and quantum chemistry algorithms.

# Program execution

3/29/2021 • 2 minutes to read • [Edit Online](#)

The following program gives a first glimpse at how a Q# command-line application is implemented:

```
namespace Microsoft.Quantum.Samples {

    open Microsoft.Quantum.Arithmetic;
    open Microsoft.Quantum.Arrays as Array;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Diagnostics as Diagnostics;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Preparation;

    operation ApplyQFT (reg : LittleEndian) : Unit
    is Adj + Ctl {

        let qs = reg!;
        SwapReverseRegister(qs);

        for (i in Array.IndexRange(qs)) {
            for (j in 0 .. i-1) {
                Controlled R1Frac([qs[i]], (1, i - j, qs[j]));
            }
            H(qs[i]);
        }
    }

    @EntryPoint()
    operation RunProgram(vector : Double[]) : Unit {

        let n = Floor(Log(IntAsDouble(Length(vector))) / LogOf2());
        if (1 <<< n != Length(vector)) {
            fail "Length(vector) needs to be a power of two.";
        }

        let amps = Array.Mapped(ComplexPolar(_,0.), vector);
        using (qs = Qubit[n]) {
            let reg = LittleEndian(qs);

            PrepareArbitraryState(amps, reg);
            Message("Before QFT:");
            Diagnostics.DumpRegister(), qs);

            ApplyQFT(reg);
            Message("After QFT:");
            Diagnostics.DumpRegister(), qs);

            ResetAll(qs);
        }
    }
}
```

The operation `PrepareArbitraryState` initializes a quantum state where the amplitudes for each basis state correspond to the normalized entries of the specified vector. A quantum Fourier transformation (QFT) is then applied to that state.

The corresponding project file to build the application is the following:

```
<Project Sdk="Microsoft.Quantum.Sdk/0.12.20070124">

<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>netcoreapp3.1</TargetFramework>
</PropertyGroup>

</Project>
```

The first line specifies the version number of the software development kit used to build the application, and line 4 indicates that the project is executable opposed to e.g. a library that cannot be invoked from the command line.

To run the application, you will need to install [.NET Core](#). Then put both files in the same folder and run `dotnet build <projectFile>`, where `<projectFile>` is to be replaced with the path to the project file.

To run the program after having built it, run the command

```
dotnet run --no-build --vector 1. 0. 0. 0.
```

The output from this invocation shows that the amplitudes of the quantum state after application of the QFT are evenly distributed and real. Note that the reason that we can so readily output the amplitudes of the state vector is that the previous program is, by default, run on a full state simulator, which supports outputting the tracked quantum state via `DumpRegister` for debugging purposes. The same would not be possible if we were to run it on quantum hardware instead, in which case the two calls to `DumpRegister` wouldn't do anything. You can see this by targeting the application to a particular hardware platform by adding the project property

```
<ExecutionTarget>honeywell.qpu</ExecutionTarget> after <PropertyGroup>.
```

# Namespaces

3/29/2021 • 2 minutes to read • [Edit Online](#)

At its top-level, a Q# program consists of a set of namespaces. Aside from [comments](#), namespaces are the only top-level elements in a Q# program, and any other elements must reside within a namespace. Each file may contain zero or more namespaces, and each namespace may span multiple files. Q# does not support nested namespaces.

A namespace block consists of the keyword `namespace`, followed by the namespace name, and the content of the block inside braces `{ }`. Namespace names consist of a sequence of one or more [legal symbols](#) separated by a dot (`.`). Double underscores (`__`), double dots (`...`), or an underscore followed by a dot (`_.`) are not permitted since these character sequences are reserved. More precisely, a fully qualified name may not contain such a sequence, and namespace names correspondingly cannot end with an underscore. While namespace names may contain dots for better readability, Q# does not support relative references to namespaces. For example, two namespaces `Foo` and `Foo.Bar` are unrelated, and there is no notion of a hierarchy. In particular, for a function `Baz` defined in `Foo.Bar`, it is *not* possible to open `Foo` and then access that function via `Bar.Baz`.

Within a namespace block, [open directives](#) precede any other namespace elements. Aside from `open` directives, namespace blocks may contain [operation](#), [function](#), and [type](#) declarations. These may occur in any order and are recursive by default, meaning they can be declared and used in any order and can call themselves; there is no need for the declaration of a type or callable to precede its use.

## Open Directives

By default, everything declared within the same namespace can be accessed without further qualification. However, declarations in a different namespace can only be used by qualifying their name with the name of the namespace they belong to or by opening that namespace before it is used, as shown in the following example.

```
namespace Microsoft.Quantum.Samples {  
  
    open Microsoft.Quantum.Arithmetic;  
    open Microsoft.Quantum.Arrays as Array;  
  
    // ...  
}
```

The example uses an `open` directive to import all types and callables declared in the `Microsoft.Quantum.Arithmetic` namespace. They can then be referred to by their unqualified name unless that name conflicts with a declaration in the namespace block or another opened namespace.

To avoid typing out the full name while still distinguishing where certain elements come from, you can define an alternative name, or *alias*, which is usually shorter, for a particular namespace. In this case, all types and callables declared in that namespace can be qualified by the defined short name instead. In the previous example, this is the case for the `Microsoft.Quantum.Arrays` namespace. A function `IndexRange` declared in `Microsoft.Quantum.Arrays`, for example, can then be used via `Array.IndexRange` within that namespace block.

Defining namespace aliases is particularly helpful when combined with the code completion functionality provided by the Q# extensions available for Visual Studio Code and Visual Studio. With the extension installed, typing the namespace alias followed by a dot will show a list of all the available elements in that namespace that are valid at the current location.

Whether you are opening a namespace or defining an alias, `open` directives need to precede any other namespace elements and are valid throughout the namespace piece in that file only.

# Type Declarations

3/29/2021 • 2 minutes to read • [Edit Online](#)

Q# supports user-defined types. User-defined types are similar to record types in F#; they are immutable but support a [copy-and-update](#) construct.

## User-defined types

User-defined types may contain both named and anonymous items. The following declaration within a namespace, for example, defines a type `Complex` which has two named items `Real` and `Imaginary`, both of type `Double`:

```
newtype Complex = (Real: Double, Imaginary : Double);
```

Any combination of named and unnamed items is supported, and inner items may also be named. For example, the type `Nested`, defined as

```
newtype Nested = (Double, (ItemName : Int, String));
```

contains two anonymous items of type `Double` and `String` respectively, and a named item `ItemName` of type `Int`.

You can access the contained items via their name or by *deconstruction* (for more information, see [item access](#)). You can also access a tuple of all items where the shape matches the one defined in the declaration via the [unwrap operator](#).

User-defined types are useful for two reasons. First, as long as the libraries and programs that use the defined types access items via their name rather than by deconstruction, the type can be extended to contain additional items later on without breaking any library code. Because of this, accessing items via deconstruction is generally discouraged.

Second, Q# allows you to convey the intent and expectations for a specific data type since there is no automatic conversion between values of two user-defined types, even if their item types are identical.

For example, the arithmetic library includes quantum arithmetic operations for both big-endian and little-endian quantum integers. It hence defines two types, `BigEndian` and `LittleEndian`, both of which contain a single anonymous item of type `Qubit[]`:

```
newtype BigEndian = Qubit[];
newtype LittleEndian = Qubit[];
```

These types allow operations to specify whether they are written for big-endian or little-endian representations and leverages the type system to ensure at compile-time that mismatched operands aren't allowed.

Type names must be unique within a namespace and may not conflict with operation and function names. Types may not have circular dependencies in Q#; that is, defining something like a directly or indirectly recursive type is not allowed. For example, the following construct will give a compilation error:

```
newtype Foo = (Foo, Int); // gives an error
newtype Bar = Baz;       // gives an error
newtype Baz = Bar;       // gives an error
```

## User-defined constructors

Constructors for user-defined types are automatically generated by the compiler. Currently, it is not possible to define a custom constructor, though this may be an addition to the language in the future.

# Callable declarations

3/29/2021 • 3 minutes to read • [Edit Online](#)

Callable declarations, or *callables*, are declared at a global scope and publicly visible by default; that is, they can be used anywhere in the same project and in a project that references the assembly in which they are declared. [Access modifiers](#) allow you to restrict their visibility to the current assembly only, such that implementation details can be changed later on without breaking code that relies on a specific library.

Q# supports two kinds of callables: operations and functions. The topic [Operations and Functions](#) elaborates on the distinction between the two. Q# also supports defining *templates*; for example, type-parameterized implementations for a certain callable. For more information, see [Type parameterizations](#).

## NOTE

Such type-parametrized implementations may not use any language constructs that rely on particular properties of the type arguments; there is currently no way to express type constraints in Q#, or to define specialized implementations for particular type arguments. However, it is conceivable to introduce a suitable mechanism, similar to type classes in Haskell, for example, to allow for more expressiveness in the future.

## Callables and functors

Q# allows specialized implementations for specific purposes; for example, operations in Q# can implicitly or explicitly define support for certain *functors*, and along with it the specialized implementations to invoke when a specific functor is applied to that callable.

A functor, in a sense, is a factory that defines a new callable implementation that has a specific relation to the callable it was applied to. Functors are more than traditional higher-level functions in that they require access to the implementation details of the callable they have been applied to. In that sense, they are similar to other factories, such as templates. Correspondingly, they can be applied not just to callables, but templates as well.

The example program shown in [Program implementation](#), for example, defines the operation `ApplyQFT` and the operation `RunProgram`, which is used as an entry point. `ApplyQFT` takes a tuple-valued argument containing an integer and a value of type `LittleEndian` and returns a value of type `Unit`. The annotation `is Adj + Ctl` in the declaration of `ApplyQFT` indicates that the operation supports both the `Adjoint` and the `Controlled` functor. (For more information, see [Operation characteristics](#)). If `Unitary` is an operation that has an adjoint and a controlled specialization, the expression `Adjoint Unitary` accesses the specialization that implements the adjoint of `Unitary`, and `Controlled Unitary` accesses the specialization that implements the controlled version of `Unitary`. In addition to the original operation's argument, the controlled version of an operation takes an array of control qubits and then applies the original operation conditional on all of these control qubits being in a  $|1\rangle$  state.

In theory, an operation for which an adjoint version can be defined should also have a controlled version and vice versa. In practice, however, it may be hard to develop an implementation for one or the other, especially for probabilistic implementations following a repeat-until-success pattern. For that reason, Q# allows you to declare support for each functor individually. However, since the two functors commute, an operation that defines support for both also has to have an implementation (usually implicitly defined, meaning compiler-generated) for when both functors are applied to the operation.

There are no functors that can be applied to functions, such that functions currently have exactly one body implementation and no further specializations. For example, the declaration

```
function Hello (name : String) : String {
    return $"Hello, {name}!";
}
```

is equivalent to

```
function Hello (name : String) : String {
    body (...) {
        return $"Hello, {name}!";
    }
}
```

Here, `body` specifies that the given implementation applies to the default body of the function `Hello`, meaning the implementation is invoked when no functors or other factory mechanisms have been applied prior to invocation. The three dots in `body (...)` correspond to a compiler directive indicating that the argument items in the function declaration should be copy and pasted into this spot.

The reasons behind explicitly indicating where the arguments of the parent callable declaration are to be copied and pasted are twofold: one, it is unnecessary to repeat the argument declaration, and two, and more importantly, it ensures that functors that require additional arguments, like the `Controlled` functor, can be introduced in a consistent manner.

The same applies to operations; when there is exactly one specialization defining the implementation of the default body, the additional wrapping of the form `body (...){ <implementation> }` may be omitted.

## Recursion

Q# callables can be directly or indirectly recursive and can be declared in any order; an operation or function may call itself, or it may call another callable that directly or indirectly calls the caller.

When running on quantum hardware, stack space may be limited, and recursions that exceed that stack space limit result in a runtime error.

# Specialization declarations

3/29/2021 • 4 minutes to read • [Edit Online](#)

As explained in the section about [callable declarations](#), there is currently no reason to explicitly declare specializations for functions. This topic applies to operations and elaborates on how to declare the necessary specializations to support certain functors.

It is quite a common problem in quantum computing to require the adjoint of a given transformation. Many quantum algorithms require both an operation and its adjoint to perform a computation. Q# employs symbolic computation that can automatically generate the corresponding adjoint implementation for a particular body implementation. This generation is possible even for implementations that freely mix classical and quantum computations. There are, however, some restrictions that apply in this case. For example, auto-generation is not supported for performance reasons if the implementation makes use of mutable variables. Moreover, each operation called within the body generates the corresponding adjoint needs to support the `Adjoint` functor itself.

Even though one cannot easily undo measurements in the multi-qubit case, it is possible to combine measurements so that the applied transformation is unitary. In this case, it means that, even though the body implementation contains measurements that on their own don't support the `Adjoint` functor, the body in its entirety is adjointable. Nonetheless, auto-generating the adjoint implementation will fail in this case. For this reason, it is possible to manually specify the implementation. The compiler automatically generates optimized implementations for common patterns such as [conjugations](#). Nonetheless, an explicit specialization may be desirable to define a more optimized implementation by hand. It is possible to specify any one implementation and any number of implementations explicitly.

## NOTE

The correctness of such a manually specified implementation is not verified by the compiler.

In the following example, the declaration for an operation `SWAP`, which exchanges the state of two qubits `q1` and `q2`, declares an explicit specialization for its adjoint version and its controlled version. While the implementations for `Adjoint SWAP` and `Controlled SWAP` are thus user-defined, the compiler still needs to generate the implementation for the combination of both functors (`Controlled Adjoint SWAP`, which is the same as `Adjoint Controlled SWAP`).

```

operation SWAP (q1 : Qubit, q2 : Qubit) : Unit
is Adj + Ctl {

    body (...) {
        CNOT(q1, q2);
        CNOT(q2, q1);
        CNOT(q1, q2);
    }

    adjoint (...) {
        SWAP(q1, q2);
    }

    controlled (cs, ...) {
        CNOT(q1, q2);
        Controlled CNOT(cs, (q2, q1));
        CNOT(q1, q2);
    }
}

```

## Auto-generation directives

When determining how to generate a particular specialization, the compiler prioritizes user-defined implementations. This means that if an adjoint specialization is user-defined and a controlled specialization is auto-generated, then the controlled adjoint specialization is generated based on the user-defined adjoint and vice versa. In this case, both specializations are user-defined. As the auto-generation of an adjoint implementation is subject to more limitation, the controlled adjoint specialization defaults to generating the controlled specialization of the explicitly defined implementation of the adjoint specialization.

In the case of the `SWAP` implementation, the better option is to adjoint the controlled specialization to avoid unnecessarily conditioning the execution of the first and the last `CNOT` on the state of the control qubits. Adding an explicit declaration for the controlled adjoint version that specifies a suitable *generation directive* forces the compiler to generate the controlled adjoint specialization based on the manually specified implementation of the controlled version instead. Such an explicit declaration of a specialization that is to be generated by the compiler takes the form

```
controlled adjoint invert;
```

and is inserted inside the declaration of `SWAP`. On the other hand, inserting the line

```
controlled adjoint distribute;
```

forces the compiler to generate the specialization based on the defined (or generated) adjoint specialization. See this [partial specialization inference](#) proposal for more details.

For the operation `SWAP`, there is a better option. `SWAP` is *self-adjoint*, that is, it is its own inverse; the -defined implementation of the adjoint merely calls the body of `SWAP`. You express this with the directive

```
adjoint self;
```

Declaring the adjoint specialization in this manner ensures that the controlled adjoint specialization that is automatically inserted by the compiler merely invokes the controlled specialization.

The following generation directives exist and are valid:

SPECIALIZATION	DIRECTIVE(S)
<code>body</code> specialization:	-
<code>adjoint</code> specialization:	<code>self</code> , <code>invert</code>
<code>controlled</code> specialization:	<code>distribute</code>
<code>controlled adjoint</code> specialization:	<code>self</code> , <code>invert</code> , <code>distribute</code>

That all generation directives are valid for a controlled adjoint specialization is not a coincidence; as long as functors commute, the set of valid generation directives for implementing the specialization for a combination of functors is always the union of the set of valid generators for each one.

In addition to the previously listed directives, the directive `auto` is always valid; it indicates that the compiler should automatically pick a suitable generation directive. The declaration

```
operation DoNothing() : Unit {
    body (...) { }
    adjoint auto;
    controlled auto;
    controlled adjoint auto;
}
```

is equivalent to

```
operation DoNothing() : Unit
is Adj + Ctl { }
```

The annotation `is Adj + Ctl` in this example specifies the *operation characteristics*, which contain the information about what functors a particular operation supports.

While for readability's sake, it is recommended that you annotate each operation with a complete description of its characteristics, the compiler automatically inserts or completes the annotation based on explicitly declared specializations. Conversely, the compiler also generates specializations that haven't been declared explicitly but need to exist based on the annotated characteristics. We say the given annotation has *implicitly declared* these specializations. The compiler automatically generates the necessary specializations if it can, picking a suitable directive. Q# thus supports inference of both operation characteristics and existing specializations based on (partial) annotations and explicitly defined specializations.

In a sense, specializations are similar to individual overloads for the same callable, with the caveat that certain restrictions apply to which overloads you can declare.

# Comments

3/29/2021 • 2 minutes to read • [Edit Online](#)

Comments begin with two forward slashes (//) and continue until the end of line. Such end-of-line comments may appear anywhere in the source code. Q# does not currently support block comments.

## Documentation Comments

Comments that begin with three forward slashes, ///, are treated specially by the compiler when they appear before a type or callable declaration. In that case, their contents are taken as documentation for the defined type or callable, as for other .NET languages.

Within /// comments, text to appear as a part of API documentation is formatted as [Markdown](#), with different parts of the documentation indicated by specially-named headers. As an extension to Markdown, cross-references to operations, functions, and user-defined types in Q# can be included using @<ref target>, where <ref target> is replaced by the fully qualified name of the code object being referenced. Optionally, a documentation engine may also support additional Markdown extensions.

For example:

```
/// # Summary
/// Given an operation and a target for that operation,
/// applies the given operation twice.
///
/// # Input
/// ## op
/// The operation to be applied.
/// ## target
/// The target to which the operation is to be applied.
///
/// # Type Parameters
/// ## 'T
/// The type expected by the given operation as its input.
///
/// # Example
/// ``Q#
/// // Should be equivalent to the identity.
/// ApplyTwice(H, qubit);
/// ``
///
/// # See Also
/// - Microsoft.Quantum.Intrinsic.H
operation ApplyTwice<'T>(op : ('T => Unit), target : 'T) : Unit {
    op(target);
    op(target);
}
```

Q# recognizes the following names as documentation comment headers.

- **Summary:** A short summary of a function or operation's behavior or the purpose of a type. The first paragraph of the summary is used for hover information. It should be plain text.
- **Description:** A description of a function or operation's behavior or the purpose of a type. The summary and description are concatenated to form the generated documentation file for the function, operation, or type. The description may contain in-line LaTeX-formatted symbols and equations.
- **Input:** A description of the input tuple for an operation or function. May contain additional Markdown

subsections indicating each element of the input tuple.

- **Output:** A description of the tuple returned by an operation or function.
- **Type Parameters:** An empty section that contains one additional subsection for each generic type parameter.
- **Named Items:** A description of the named items in a user-defined type. May contain additional Markdown subsections with the description for each named item.
- **Example:** A short example of the operation, function, or type in use.
- **Remarks:** Miscellaneous prose describing some aspect of the operation, function, or type.
- **See Also:** A list of fully qualified names indicating related functions, operations, or user-defined types.
- **References:** A list of references and citations for the documented item.

# Statements

3/29/2021 • 2 minutes to read • [Edit Online](#)

Q# distinguishes between statements and expressions. Q# programs consist of a mixture of classical and quantum computations and the implementation looks much like in any other classical programming language. Some statements, such as the `let` and `mutable` bindings, are well-known from classical languages, while others such as conjugations or qubit allocations are unique to the quantum domain.

The following statements are currently available in Q#:

- **Call Statement**

A call statement consists of an operation or function call returning `Unit`. The invoked callable needs to satisfy the requirements imposed by the current context. See [this section](#) for more details.

- **Return Statement**

A return statement terminates the execution within the current callable context and returns control to the caller. Any finalizing tasks are executed after the return value is evaluated but before control is returned. See [this section](#) for more details.

- **Fail Statement**

A fail statement aborts the execution of the entire program, collecting information about the current program state before terminating in an error. It aggregates the collected information and presents it to the user along with the message specified as part of the statement. See [this section](#) for more details.

- **Variable Declaration**

Defines one or more local variables that will be valid for the remainder of the current scope, and binds them to the specified values. Variables can be permanently bound or declared to be reassignable later on. See [this section](#) for more details.

- **Variable Reassignment**

Variables that have been declared as being reassignable can be rebound to contain different values. See [this section](#) for more details.

- **Iteration**

An iteration is a loop-like statement that during each iteration assigns the declared loop variables to the next item in a sequence (a value of array or `Range` type) and executes a specified block of statements. See [this section](#) for more details.

- **While Statement**

If a specified condition evaluates to `true`, a block of statements is executed. The execution is repeated indefinitely until the condition evaluates to `false`. See [this section](#) for more details.

- **Repeat Statement**

Quantum-specific loop that breaks based on a condition. The statement consists of an initial block of statements that is executed before a specified condition is evaluated. If the condition evaluates to `false`, an optional subsequent `fixup`-block is executed before entering the next iteration of the loop. The loop terminates only once the condition evaluates to `true`. See [this section](#) for more details.

- **If Statement**

The statement consists of one or more blocks of statements, each preceded by a boolean expression. The first block for which the boolean expression evaluates to `true` is executed. Optionally, a block of statements can be specified that is executed if none of the conditions evaluates to `true`. See [this section](#)

for more details.

- **Conjugation**

A conjugations is a special quantum-specific statement, where a block of statements that applies a unitary transformation to the quantum state is executed, followed by another statement block, before the transformation applied by the first block is reverted again. In mathematical notation, conjugations describe transformations of the form  $U^\dagger V U$  to the quantum state. See [this section](#) for more details.

- **Qubit Allocation**

Instantiates and initializes qubits and/or arrays of qubits, and binds them to the declared variables. Executes a block of statements. The instantiated qubits are available for the duration of the block, and will be automatically released when the statement terminates. See [this section](#) for more details.

# Visibility of Local Variables

3/29/2021 • 2 minutes to read • [Edit Online](#)

In general, symbol bindings become inoperative at the end of the statement block they occur in. The exceptions to this rule are:

- Bindings of the loop variables in a `for` loop are defined for the body of the loop, but not after the end of the loop.
- Bindings of allocated qubits in `use`- and `borrow`-statements are defined for the body of the allocation, but not after the statement terminates. This only applies to `use` and `borrow`-statements that have an associated statement block.
- For `repeat`-statements, both blocks as well as the condition are treated as a single scope; i.e. symbols that are bound in the body are accessible in the condition and in the `fixup`-block.

For loops, each iteration executes in its own scope, and all defined variables are bound anew for each iteration.

Bindings in outer blocks are visible and defined in inner blocks. A symbol may only be bound once per block; it is illegal to define a symbol with the same name as another symbol that is accessible (no "shadowing").

The following sequences would be legal:

```
if (a == b) {  
    ...  
    let n = 5;  
    ...           // n is 5  
}  
let n = 8;          // n is 8  
...
```

and

```
if (a == b) {  
    ...  
    let n = 5;  
    ...           // n is 5  
} else {  
    ...  
    let n = 8;  
    ...           // n is 8  
}  
...           // n is not bound to a value
```

The following sequences would be illegal:

```
let n = 5;  
...           // n is 5  
let n = 8;          // Error  
...
```

and

```
let n = 8;
if (a == b) {
    ...
    // n is 8
    let n = 5;      // Error
    ...
}
...
...
```

[This section](#) gives further details on variable declarations and reassignments.

# Call Statements

3/29/2021 • 2 minutes to read • [Edit Online](#)

Call statements are a very important part of any programming language. While operation and function calls, much like [partial applications](#), can be used as an expression anywhere as long as the returned value is of a suitable type, they can also be used as statements if they return `Unit`. The usefulness of calling functions in this form primarily lies in debugging, whereas such operation calls are one of the most common constructs in any Q# program. At the same time, operations can only be called from within other operations and not from within functions (for context, see also [this section](#)).

With callables being first-class values, call statements in a sense are the generic way of supporting patterns that aren't common enough to merit their own dedicated language construct or dedicated syntax has not (yet) been introduced for other reasons. Examples for library methods that serve exactly that purpose are `ApplyIf`, that invokes an operation conditional on a classical bit being set, `ApplyToEach`, that applies a given operation to each element in an array, and `ApplyWithInputTransformation` shown below to give to just a few.

```
operation ApplyWithInputTransformation<'TArg, 'TIn>(
    fn : 'TIn -> 'TArg,
    op : 'TArg => Unit,
    input : 'TIn
) : Unit {
    op(fn(input));
}
```

`ApplyWithInputTransformation` takes a function `fn`, an operation `op`, and an `input` value as argument, applies the give function to the input, before invoking the given operation with the value returned form the function.

For the compiler to be able to auto-generate the specializations to support particular [functors](#) usually requires that the called operations support those functors as well. The exception are calls in outer blocks of [conjugations](#), which always need to support the `Adjoint` functor but never need to support the `Controlled` functor, and self-adjoint operations, which support the `Adjoint` functor without imposing any additional requirements on the individual calls.

## Discussion

More sophisticated [generation directives](#) may allow to further relax that requirement in the future.

# Returns and Termination

3/29/2021 • 2 minutes to read • [Edit Online](#)

There are two statements available that conclude the execution of the current subroutine or the program; the `return`- and the `fail`-statement. For callables that return any other type than `Unit` each execution path needs to terminate either in a `return`- or a `fail`-statement

## Return-Statement

The `return`-statement exits from the current callable and returns control to the callee. It changes the context of the execution by popping a stack frame.

The statement always returns a value back to the context of the callee; it consists of the keyword `return`, followed by an expression of the appropriate type, and a terminating semicolon. The return value is evaluated before any terminating actions are performed and control is returned. Such terminating actions include, e.g., cleaning up and releasing qubits that have been allocated within the context of the callable. When executing on a simulator or validator, terminating actions often also include checks related to the state of those qubits, like, e.g., whether they are properly disentangled from all qubits that remain live.

The `return`-statement at the end of a callable that returns a `Unit` value may be omitted. In that case, control is returned automatically when all statements have been executed and all terminating actions were performed. Callables may contain multiple `return`-statements, one for each possible execution path, albeit the adjoint implementation for operations containing multiple `return`-statements cannot be automatically generated.

For example,

```
return 1;
```

or

```
return ();
```

## Fail-Statement

The `fail`-statement on the other hand aborts the computation entirely. It corresponds to a fatal error that was not expected to happen as part of normal execution.

It consists of the keyword `fail`, followed by an expression of type `String` and a terminating semicolon. The `String` value should be used to give information about the encountered failure.

For example,

```
fail "Impossible state reached";
```

or, using an [interpolated string](#),

```
fail $"Syndrome {syn} is incorrect";
```

In addition to the given `String`, ideally a `fail`-statement collects and permits to retrieve information about the program state that facilitate diagnosing and remedying the source of the error. This requires support from the executing runtime and firmware which may vary across different targets.

# Variable Declarations and Reassignments

3/29/2021 • 4 minutes to read • [Edit Online](#)

Values can be bound to symbols via `let` - and `mutable` -statements. Such bindings provide a convenient way to access a value via the defined handle. Despite the somewhat misleading terminology, borrowed from other languages, we will call handles that are declared on a local scope and contain values *variables*. The reason that this may be somewhat misleading is that `let` -statements define "single-assignment handles", i.e. handles that for the duration of their validity will always be bound to the same value. Variables that can be re-bound to different values at different points in the code need to be explicitly declared as such, as specific by the `mutable` -statement.

```
let var1 = 3;
mutable var2 = 3;
set var2 = var2 + 1;
```

Line 1 declares a variable named `var1` that cannot be reassigned and will always contain the value `3`. Line 2 on the other hand defines a variable `var2` that is temporarily bound to the value `3`, but can be reassigned to a different value later on. Such a reassignment can be done via a `set` -statement, as shown in Line 3. The same could have been expressed with the shorter version `set var2 += 1;` explained further [below](#), as it is common in other languages as well.

To summarize:

- `let` is used to create an immutable binding.
- `mutable` is used to create a mutable binding.
- `set` is used to change the value of a mutable binding.

For all three statements, the left hand side consists of a symbol or a symbol tuple; i.e. if the right-hand side of the binding is a tuple, then that tuple may be fully or partially deconstructed upon assignment. The only requirement for deconstruction is that the shape of the tuple on the right hand side matches the shape of the symbol tuple. The symbol tuple may contain nested tuples and/or omitted symbols, indicated by an underscore. For example:

```
let (a, (_, b)) = (1, (2, 3)); // a is bound to 1, b is bound to 3
mutable (x, y) = ((1, 2), [3, 4]); // x is bound to (1, 2), y is bound to [3, 4]
set (x, _, y) = ((5, 6), 7, [8]); // x is re-bound to (5,6), y is re-bound to [8]
```

The same deconstruction rules are obeyed by all assignments in Q#, including, e.g., qubit allocations and loop-variable assignments.

For both kinds of binding, the types of the variables are inferred from the right-hand side of the binding. The type of a variable always remains the same and a `set` -statement cannot change it. Local variable can be declared as either being mutable or immutable, with some exceptions like loop-variables in `for` -loops for which the behavior is predefined and cannot be specified. Function and operation arguments are always immutably bound; in combination with the lack of reference types, as discussed in the section on [immutability](#), that means that a called function or operation can never change any values on the caller side. Since the states of `Qubit` values are not defined or observable from within Q#, this does not preclude the accumulation of quantum side effects, that are observable (only) via measurements (see also [this section](#)).

Independent on how a value is bound, the values themselves are immutable. This in particular also holds for

arrays and array items. In contrast to popular classical languages where arrays often are reference types, arrays - like all type - are value types in Q# and always immutable; they cannot be modified after initialization. Changing the values accessed by variables of array type thus requires explicitly constructing a new array and reassigning it to the same symbol, see also the section on [immutability](#) and [copy-and-update expressions](#) for more details.

## Evaluate-and-Reassign Statements

Statements of the form `set intValue += 1;` are common in many other languages. Here, `intValue` needs to be a mutably bound variable of type `Int`. Such statements provide a convenient way of concatenation if the right hand side consists of the application of a binary operator and the result is to be rebound to the left argument to the operator. For example,

```
mutable counter = 0;
for i in 1 .. 2 .. 10 {
    set counter += 1;
    // ...
}
```

increments the value of the counter `counter` in each iteration of the `for` loop. The code above is equivalent to

```
mutable counter = 0;
for i in 1 .. 2 .. 10 {
    set counter = counter + 1;
    // ...
}
```

Similar statements exist for a wide range of [operators](#). The `set` keyword in this case needs to be followed by a single mutable variable, which is inserted as the left-most sub-expression by the compiler. Such evaluate-and-reassign statements exist for all operators where the type of the left-most sub-expression matches the expression type. More precisely, they are available for binary logical and bitwise operators including right and left shift, for arithmetic expressions including exponentiation and modulus, for concatenations, as well as for [copy-and-update expressions](#).

The following function for example computes the sum of an array of `Complex` numbers:

```
function ComplexSum(values : Complex[]) : Complex {
    mutable res = Complex(0., 0.);
    for complex in values {
        set res w/= Re <- res::Re + complex::Re;
        set res w/= Im <- res::Im + complex::Im;
    }
    return res;
}
```

Similarly, the following function multiplies each item in an array with the given factor:

```
function Multiplied(factor : Double, array : Double[]) : Double[] {
    mutable res = new Double[Length(array)];
    for i in IndexRange(res) {
        set res w/= i <- factor * array[i];
    }
    return res;
}
```

The section on [contextual expressions](#) contains other examples where expressions can be omitted in a certain

context when a suitable expression can be inferred by the compiler.

# Iterations

3/29/2021 • 2 minutes to read • [Edit Online](#)

Loops that iterate over a sequence of values are expressed as `for`-loops in Q#. A `for`-loop in Q# does not break based on a condition, but instead corresponds to what is often expressed as `foreach` or `iter` in other languages. There are currently two data types in Q# that support iteration: arrays and ranges.

The statement consists of the keyword `for`, followed by a symbol or symbol tuple, the keyword `in`, an expression of array or `Range` type, and a statement block.

The statement block (the body of the loop) is executed repeatedly, with the defined symbol(s) (the loop variable(s)) bound to each value in the range or array. The same deconstruction rules apply to the defined loop variable(s) as to any other variable assignment, such as bindings in `let`-, `mutable`-, `set`-, `use`- and `borrow`-statements. The loop variables themselves are immutably bound, cannot be reassigned within the body of the loop, and go out of scope when the loop terminates. The expression over which the loop iterates is fully evaluated before entering the loop, and will not change while the loop is executing.

Supposing `qubits` is a value of type `Qubit[]`. The following examples illustrate what is described above:

```
for qubit in qubits {
    H(qubit);
}

mutable results = new (Int, Result)[0];
for index in 0 .. Length(qubits) - 1 {
    set results += [(index, M(qubits[index]))];
}

mutable accumulated = 0;
for (index, measured) in results {
    if measured == One {
        set accumulated += 1 <<< index;
    }
}
```

## Target-Specific Restrictions

There are no `break`- or `continue`-primitives in Q#, such that the length of the loop is perfectly predictable as soon as the value to iterate over is known. Such `for`-loops can hence be executed on all quantum hardware.

# Conditional Loops

3/29/2021 • 2 minutes to read • [Edit Online](#)

Much like most classical programming languages, Q# supports loops that break based on a condition, i.e. loops for which the number of iterations is unknown and may vary from run to run. Since the instruction sequence is unknown at compile time, these kinds of loops need to be handled with particular care in a quantum runtime.

As long as the condition does not depend on quantum measurements, such loops can be handled without issues by doing a just-in-time compilation before sending off the instruction sequence to the quantum processor. In particular, their use within functions is unproblematic, since code within functions can always be executed on conventional (non-quantum) hardware. Q# therefore supports the use of traditional `while`-loops within functions.

Additionally, Q# allows to express control flow that depends on the results of quantum measurements. This capability enables probabilistic implementations that can significantly reduce the computational costs. A common example are *repeat-until-success* patterns, which repeat a computation until a certain condition - which usually depends on a measurement - is satisfied. Such `repeat`-loops are widely used in particular classes of quantum algorithms, and Q# hence has a dedicated language construct to express them, despite that they still pose a challenge for execution on quantum hardware.

## Repeat-Statement

The `repeat`-statement takes the following form:

```
repeat {  
    // ...  
}  
until condition  
fixup {  
    // ...  
}
```

or alternatively

```
repeat {  
    // ...  
}  
until condition;
```

where `condition` is an arbitrary expression of type `Bool`.

The `repeat`-statement executes a block of statements before evaluating a condition. If the condition evaluates to true, the loop exists. If the condition evaluates to false, an additional block of statements defined as part of an optional `fixup`-block, if present, is executed prior to entering the next loop iteration.

All parts of the `repeat`-statement (both blocks and the condition) are treated as a single scope for each repetition; i.e. symbols that are defined within the `repeat`-block are visible both to the condition and within the `fixup`-block. As for other loops, symbols go out of scope after each iteration, such that symbols defined in the `fixup`-block are not visible in the `repeat`-block.

### Target-Specific Restrictions

Loops that break based on a condition are a challenge to process on quantum hardware if the condition

depends on measurement outcomes, since the length of the instruction sequence to execute is not known ahead of time.

Despite their common presence in particular classes of quantum algorithms, current hardware does not yet provide native support for these kind of control flow constructs. Execution on quantum hardware can potentially be supported in the future by imposing a maximum number of iterations.

## While-Loop

A more familiar looking statement for classical computations is the `while`-loop. It is supported only within functions.

A `while` statement consists of the keyword `while`, an expression of type `Bool`, and a statement block. For example, if `arr` is an array of positive integers,

```
mutable (item, index) = (-1, 0);
while index < Length(arr) && item < 0 {
    set item = arr[index];
    set index += 1;
}
```

The statement block is executed as long as the condition evaluates to `true`.

### ***Discussion***

Due to the challenge they pose for execution, we would like to discourage the use of loops that break based on a condition and hence do not support while-loops within operations. We may consider allowing the use of `while`-loops within operations in the future, imposing that the condition cannot depend on the outcomes of quantum measurements.

# Conditional Branching

3/29/2021 • 2 minutes to read • [Edit Online](#)

Conditional branching is expressed in the form of `if`-statements. An `if`-statement consists of an `if`-clause, followed by zero or more `elif`-clauses and optionally an `else`-block. Each clause follows the pattern:

```
keyword condition {  
    <statements>  
}
```

where `keyword` is to be replaced with `if` or `elif` respectively, `condition` is an expression of type `Bool`, and `<statements>` is to be replaced with zero or more statements. The optional `else`-block consists of the keyword `else` followed by zero or more statements enclosed in `{` and `}`.

The first block for which the `condition` evaluates to `true` will be executed. The `else`-block, if present, is executed if none of the conditions evaluate to `true`. The block is executed in its own scope, meaning any bindings made as part of the statement block are not visible after its end.

For example, suppose `qubits` is value of type `Qubit[]` and `r1` and `r2` are of type `Result`,

```
if r1 == One {  
    let q = qubits[0];  
    H(q);  
}  
elif r2 == One {  
    let q = qubits[1];  
    H(q);  
}  
else {  
    H(qubits[2]);  
}
```

Additionally, Q# also allows to express simple branching in the form of a [conditional expression](#).

## Target-Specific Restrictions

A tight integration between control-flow constructs and quantum computations poses a challenge for current quantum hardware. Certain quantum processors do not support branching based on measurement outcomes. Comparison for values of type `Result` will hence always result in a compilation error for Q# programs that are targeted to execute on such hardware.

Other quantum processors support specific kinds of branching based on measurement outcomes. The more general `if`-statements supported in Q# are compiled into suitable instructions that can be executed on such processors. The imposed restrictions are that values of type `Result` may only be compared as part of the condition within `if`-statements in operations. The conditionally executed blocks furthermore cannot contain any return statements or update mutable variables that are declared outside that block.

# Conjugations

3/29/2021 • 2 minutes to read • [Edit Online](#)

Conjugations are fairly omnipresent in quantum computations. Expressed in mathematical terms, they are patterns of the form  $U^\dagger V U$  for unitary transformations  $U$  and  $V$ . That pattern is especially relevant due to the particularities of quantum memory: To leverage the unique assets of quantum, computations build up quantum correlations, i.e. entanglement. However, that also means that once qubits are no longer needed for a particular subroutine, they cannot easily be reset and released, since observing their state would impact the rest of the system. For that reason, the effects of a previous computation commonly need to be reversed prior to being able to release and reuse quantum memory.

Q# hence has a dedicated statement for expressing computation that require such a subsequent clean-up. The statement consists of two code blocks, one containing the implementation of  $U$  and one containing the implementation of  $V$ . The uncomputation (i.e. the application of  $U^\dagger$ ) is done automatically as part of the statement.

The statement takes the form

```
within {
    <statements>
}
apply {
    <statements>
}
```

where `<statements>` is to be replaced with any number of statements defining the implementation of  $U$  and  $V$  respectively. Both blocks may contain arbitrary classical computations, aside from the usual restrictions for automatically generating adjoint versions that apply to the `within`-block. Mutably bound variables that are used as part of the `within`-block may not be reassigned as part of the `apply`-block.

The example of the `ApplyXOrIfGreater` operation defined in the arithmetic library illustrates the usage of such a conjugation: The operation maps  $|lhs\rangle|rhs\rangle|res\rangle \rightarrow |lhs\rangle|rhs\rangle|res \oplus (lhs > rhs)\rangle$ , i.e. it coherently applies an XOR to a given qubit `res` if the quantum integer represented by `lhs` is greater than the one in `rhs`. The two integers are expected to be represented in little endian encoding, as indicated by the usage of the corresponding data type.

```

operation ApplyXOrIfGreater(
    lhs : LittleEndian,
    rhs : LittleEndian,
    res : Qubit
) : Unit is Adj + Ctl {

    let (x, y) = (lhs!, rhs!);
    let shuffled = Zip3(Most(x), Rest(y), Rest(x));

    use anc = Qubit();
    within {
        ApplyToEachCA(X, x + [anc]);
        ApplyMajorityInPlace(x[0], [y[0], anc]);
        ApplyToEachCA(MAJ, shuffled);
    }
    apply {
        X(res);
        CNOT(Tail(x), res);
    }
}

```

A temporarily used storage qubit `anc` is automatically cleaned up before it is released at the end of the operation; the statements in the `within`-block are applied first, followed by the statements in the `apply`-block, and finally the automatically generated adjoint of the `within`-block is applied to clean up the temporarily used helper qubit `anc`.

### **Discussion**

Returning control from within the `apply`-block is not yet supported. It should be possible to support this in the future. The expected behavior in this case is to evaluate the returned value before the adjoint of the `within`-block is executed, any qubits going out of scope are released (`anc` in this case), and the control is returned to the caller. In short, the statement should behave similarly to a `try-finally` pattern in C#. However, the necessary functionality is not yet implemented.

# Quantum Memory Management

3/29/2021 • 4 minutes to read • [Edit Online](#)

A program always starts with no qubits, meaning values of type `Qubit` cannot be passed as entry point arguments. This restriction is intentional, since a purpose of Q# is to express and reason about a program in its entirety. Instead, a program allocates and releases quantum memory as it goes. In this regard, Q# models the quantum computer as a qubit heap.

Rather than supporting separate allocate and release statements for quantum memory, Q# supports quantum memory allocation in the form of block statements, where the memory is accessible only within the scope of that statement. An attempt to access that memory after the statement terminates will result in a runtime exception.

## Discussion

Forcing that qubits cannot escape their scope greatly facilitates reasoning about quantum dependencies and how the quantum parts of the computation can impact the continuation of the program. An additional benefit of this setup is that qubits cannot get allocated and never freed, which avoids a class of common bugs in manual memory management languages without the overhead of qubit garbage collection.

Q# has two statements to instantiate qubit values, arrays of qubits, or any combination thereof. Both of these statements can only be used within operations. They gather the instantiated qubit values, bind them to the variable(s) specified in the statement, and then execute a block of statements. At the end of the block, the bound variables go out of scope and are no longer defined.

Q# distinguishes between the allocation of "clean" qubits, meaning qubits that are unentangled and are not used by another part of the computation, and what is often referred to as "dirty" qubits, meaning qubits whose state is unknown and can even be entangled with other parts of the quantum processor's memory.

## Use-Statement

Clean qubits are allocated by the `use`-statement. The statement consists of the keyword `use` followed by a binding and an optional statement block. If a statement block is present, the qubits are only available within that block. Otherwise, the qubits are available until the end of the current scope. The binding follows the same pattern as `let` statements: either a single symbol or a tuple of symbols, followed by an equals sign `=`, and either a single tuple or a matching tuple of *initializers*.

Initializers are available either for a single qubit, indicated as `Qubit()`, or an array of qubits, `Qubit[n]`, where `n` is an `Int` expression. For example,

```

use qubit = Qubit();
// ...

use (aux, register) = (Qubit(), Qubit[5]);
// ...

use qubit = Qubit() {
    // ...
}

use (aux, register) = (Qubit(), Qubit[5]) {
    // ...
}

```

The qubits are guaranteed to be in a  $|0\rangle$  state upon allocation. They are released at the end of the scope and are required to either be in a  $|0\rangle$  state upon release, or to have been measured right beforehand. This requirement is not compiler-enforced, since this would require a symbolic evaluation that quickly gets prohibitively expensive. When executing on simulators, the requirement can be runtime enforced. On quantum processors, the requirement cannot be runtime enforced; an unmeasured qubit may be reset to  $|0\rangle$  via unitary transformation. Failing to do so will result in incorrect behavior.

The `use`-statement allocates the qubits from the quantum processor's free qubit heap, and returns them to the heap no later than the end of the scope in which the qubits are bound.

## Borrow-Statement

The `borrow`-statement is used to make qubits available for temporary use, which do not need to be in a specific state: Some quantum algorithms are capable of using qubits without relying on their exact state - or even that they are unentangled with the rest of the system. That is, they require extra qubits temporarily, but they can ensure that those qubits are returned exactly to their original state independent of which state that was.

If there are qubits that are in use but not touched during the execution of a subroutine, those qubits can be borrowed for use by such an algorithm instead of having to allocate additional quantum memory. Borrowing instead of allocating can significantly reduce the overall quantum memory requirements of an algorithm, and is a quantum example of a typical space-time tradeoff.

A `borrow`-statement follows the same pattern as described [above](#) for a `use`-statement, with the same initializers being available. For example,

```

borrow qubit = Qubit();
// ...

borrow (aux, register) = (Qubit(), Qubit[5]);
// ...

borrow qubit = Qubit() {
    // ...
}

borrow (aux, register) = (Qubit(), Qubit[5]) {
    // ...
}

```

The borrowed qubits are in an unknown state and go out of scope at the end of the statement block. The borrower commits to leaving the qubits in the same state they were in when they were borrowed, i.e. their state at the beginning and at the end of the statement block is expected to be the same.

The `borrow`-statement retrieves in-use qubits that are guaranteed not to be used from the time the qubit is

bound until the last usage of that qubit. If there aren't enough qubits available to borrow, then qubits will be allocated from and returned to the heap like a `use` statement.

### ***Discussion***

Among the known use cases of dirty qubits are implementations of multi-controlled CNOT gates that require only very few qubits and implementations of incrementers. This [paper](#) gives an example of an algorithm that utilizes borrowed qubits.

# Expressions

3/29/2021 • 2 minutes to read • [Edit Online](#)

At the core, Q# expressions are either [value literals](#) or [identifiers](#), where identifiers can refer to either locally declared variables or to globally declared callables (there are currently no global constants in Q#). Operators, combinators, and modifiers can be used to combine these into a wider variety of expressions.

- *Operators* in a sense are nothing but dedicated syntax for particular callables.

Even though Q# is not yet expressive enough to formally capture the capabilities of each operator in the form of a backing callable declaration, that should be remedied in the future.

- *Modifiers* can only be applied to certain expressions. One or more modifiers can be applied to expressions that are either identifiers, array item access expressions, named item access expressions, or an expression within parenthesis which is the same as a single item tuple (see [this section](#)). They can either precede (prefix) the expression or follow (postfix) the expression. They are thus special unary operators that bind tighter than function or operation calls, but less tight than any kind of item access. Concretely, [functors](#) are prefix modifiers, whereas the [unwrap operator](#) is a postfix modifier.
- Like modifiers, function and operation calls as well as item access can be seen as a special kind of operator subject to the same restrictions regarding where they can be applied; we refer to them as *combinators*.

The section on [precedence and associativity](#) contains a complete [list of all operators](#) as well as a complete [list of all modifiers and combinators](#).

# Precedence and Associativity

3/29/2021 • 2 minutes to read • [Edit Online](#)

Precedence and associativity define the order in which operators are applied. Operators with higher precedence will be bound to their arguments (operands) first, while operators with the same precedence will bind be bound in the direction of their associativity. For example, the expression `1+2*3` according to the precedence for addition and multiplication is equivalent to `1+(2*3)`, and `2^3^4` equals `2^(3^4)` since exponentiation is right-associative.

## Operators

The following table lists the available operators, as well as their precedence and associativity. Additional modifiers and combinators are listed further below and bind tighter than any of these operators.

DESCRIPTION	SYNTAX	OPERATOR	ASSOCIATIVITY	PRECEDENCE
copy-and-update operator	<code>w/</code> <code>&lt;-</code>	ternary	left	1
range operator	<code>..</code>	infix	left	2
conditional operator	<code>?</code> <code>\ </code>	ternary	right	5
logical OR	<code>or</code>	infix	left	10
logical AND	<code>and</code>	infix	left	11
bitwise OR	<code>\ \ \ </code>	infix	left	12
bitwise XOR	<code>^^^</code>	infix	left	13
bitwise AND	<code>&amp;&amp;</code>	infix	left	14
equality	<code>==</code>	infix	left	20
inequality	<code>!=</code>	infix	left	20
less-than-or-equal	<code>&lt;=</code>	infix	left	25
less-than	<code>&lt;</code>	infix	left	25
greater-than-or-equal	<code>&gt;=</code>	infix	left	25
greater-than	<code>&gt;</code>	infix	left	25
right shift	<code>&gt;&gt;&gt;</code>	infix	left	28

DESCRIPTION	SYNTAX	OPERATOR	ASSOCIATIVITY	PRECEDENCE
left shift	<code>&lt;&lt;&lt;</code>	infix	left	28
addition or concatenation	<code>+</code>	infix	left	30
subtraction	<code>-</code>	infix	left	30
multiplication	<code>*</code>	infix	left	35
division	<code>/</code>	infix	left	35
modulus	<code>%</code>	infix	left	35
exponentiation	<code>^</code>	infix	right	40
bitwise NOT	<code>~~~</code>	prefix	right	45
logical NOT	<code>not</code>	prefix	right	45
negative	<code>-</code>	prefix	right	45

Copy-and-update expressions necessarily need to have the lowest precedence to ensure a consistent behavior of the corresponding [evaluate-and-reassign statement](#). Similar considerations hold for the range operator to ensure a consistent behavior of the corresponding [contextual expression](#).

## Modifiers and Combinators

Modifiers can be seen as special operators that can be applied to certain expressions only (see [this section](#) for more detail). We can assign them an artificial precedence to capture their behavior.

This artificial precedence is listed in the table below, which also shows how the precedence of operators and modifiers relates to how tight item access combinators (`[ , ]` and `::` respectively) and call combinators (`( , )`) bind.

DESCRIPTION	SYNTAX	OPERATOR	ASSOCIATIVITY	PRECEDENCE
Call combinator	<code>( )</code>	n/a	left	900
Adjoint functor	<code>Adjoint</code>	prefix	right	950
Controlled functor	<code>Controlled</code>	prefix	right	950
Unwrap application	<code>!</code>	postfix	left	1000
Named item access	<code>::</code>	n/a	left	1100
Array item access	<code>[ ]</code>	n/a	left	1100

To illustrate the implications of the assigned precedences, suppose we have a unitary operation `DoNothing` as defined in [this section](#), a callable `GetStatePrep` that returns a unitary operation, and an array `algorithms`

containing items of type `Algorithm` defined as follows

```
newtype Algorithm = (
    Register : LittleEndian,
    Initialize : Transformation,
    Apply : Transformation
);

newtype Transformation =
    LittleEndian => Unit is Adj + Ctl;
```

where `LittleEndian` is defined in [this section](#). Then the following expressions are all valid:

```
GetStatePrep()(arg)
(Transformation(GetStatePrep()))!(arg)
Adjoint DoNothing()
Controlled Adjoint DoNothing(cs, ())
Controlled algorithms[0]::Apply!(cs, _)
algorithms[0]::Register![i]
```

Looking at the precedences defined in the table above, we see that the parentheses around `(Transformation(GetStatePrep()))` are necessary for the subsequent unwrap operator to be applied to the `Transformation` value rather than the returned operation. However, parentheses are not required in `GetStatePrep()(arg)`; functions are applied left-to-right, so this expression is equivalent to `(GetStatePrep()(arg))(arg)`. Functor applications also don't require parentheses around them in order to invoke the corresponding specialization. Neither do array or named item access expressions, such that an expression `arr2D[i][j]` is perfectly valid, just like `algorithms[0]::Register![i]` is.

# Copy-and-Update Expressions

3/29/2021 • 3 minutes to read • [Edit Online](#)

To reduce the need for mutable bindings, Q# supports copy-and-update expressions for value types with item access. User defined types and arrays both are immutable and fall into this category. User defined types allow to access items via name, whereas arrays allow to access items via an index or range of indices.

Copy-and-update expressions instantiate a new value of with all items set to the corresponding value in the original expression, except certain specified items(s), which are set to the one(s) defined on the right hand side of the expression. They are constructed using a ternary operator `w/ <-`; the syntax `w/` should be read as the commonly used short notation for "with":

```
original w/ itemAccess <- modification
```

where `original` is either an expression of user defined type or an array expression. The corresponding requirements for `itemAccess` and `modification` are specified in the corresponding subsection below.

In terms of precedence, the copy-and-update operator is left-associative and has lowest precedence, and in particular lower precedence than the range operator (`...`) or the ternary conditional operator (`? |`). The chosen left associativity allows easy chaining of copy-and-update expressions:

```
let model = Default<SequentialModel>()
    w/ Structure <- ClassifierStructure()
    w/ Parameters <- parameters
    w/ Bias <- bias;
```

Like for any operator that constructs an expression that is of the same type as the left-most expression involved, the corresponding [evaluate-and-reassign statement](#) is available. The two statements below for example achieve the following: The first statement declares a mutable variable `arr` and binds it to the default value of an integer array. The second statement then builds a new array with the first item (with index 0) set to 10, and reassigns it to `arr`.

```
mutable arr = new Int[3]; // arr contains [0,0,0]
set arr w/= 0 <- 10;      // arr contains [10,0,0]
```

The second statement is nothing but a short-hand for the more verbose syntax `set arr = arr w/ 0 <- 10;`.

## Copy-and-Update of User Defined Types

If the value `original` is of user defined type, then `itemAccess` denotes the name of the item that diverges from the original value. The reason that this is not simply another expression, like `original` and `modification`, is that the ability to simply use the item name without any further qualification is limited to this context; it is one of two [contextual expressions](#) in Q#.

The type of the `modification` expression needs to match the type of the named item that diverges. For instance, if `complex` contains the value `Complex(0., 0.)`, where the type `Complex` is defined [here](#), then

```
complex w/ Re <- 1.
```

is an expression of type `Complex` that evaluates to `Complex(1.,0.)`.

## Copy-and-Update of Arrays

For arrays, `itemAccess` can be an arbitrary expression of a suitable type; the same types that are valid for array slicing are valid in this context. Concretely, the `itemAccess` expression can be of type `Int` or `Range`. If `itemAccess` is a value of type `Int`, then the type of `modification` has to match the item type of the array. If `itemAccess` is a value of type `Range` then the type of `modification` has to be the same as the array type.

For example, if `arr` contains an array `[0,1,2,3]`, then

- `arr w/ 0 <- 10` is the array `[10,1,2,3]`.
- `arr w/ 2 <- 10` is the array `[0,1,10,3]`.
- `arr w/ 0..2..3 <- [10,12]` is the array `[10,1,12,3]`.

Copy-and-update expressions allow efficient creation of new arrays based on existing ones. The implementation for copy-and-update expressions avoids copying the entire array but merely duplicates the necessary parts to achieve the desired behavior, and performs an in-place modification if possible. Array initializations hence do not incur additional overhead due to immutability.

The `Microsoft.Quantum.Arrays` namespace provides an arsenal of convenient tools for array creation and manipulation. For instance, the function `ConstantArray` creates an array of the specified length and initializes each item to a given value.

Copy-and-update expressions are a convenient way to construct new arrays on the fly; the following expression, e.g., evaluates to an array with all items set to `PauliI`, except the item at index `i`, which is set to `PauliZ`:

```
ConstantArray(n, PauliI) w/ i <- PauliZ
```

# Conditional Expressions

3/29/2021 • 2 minutes to read • [Edit Online](#)

Conditional expressions consist of three sub-expressions, where the left-most one is of type `Bool` and determines which one of the two other sub-expressions is evaluated. They are of the form

```
cond ? ifTrue | ifFalse
```

Specifically, if `cond` evaluates to `true` then the conditional expression evaluates to the `ifTrue` expression, and otherwise it evaluates to the `ifFalse` expression. The other expression (the `ifFalse` and `ifTrue` expression respectively) is never evaluated, much like for the branches in an `if`-statement. For instance, in an expression `a == b ? C(qs) | D(qs)`, if `a` equals `b` then the callable `C` will be invoked, and otherwise `D` will be invoked.

The types of the `ifTrue` and the `ifFalse` expression have to have a [common base type](#). Independent of which one of the two ultimately yields the value to which the expression evaluates, its type will always match the determined base type.

For example, if

- `Op1` is of type `Qubit[] => Unit is Adj`
- `Op2` is of type `Qubit[] => Unit is Ctl`
- `Op3` is of type `Qubit[] => Unit is Adj + Ctl`

then

- `cond ? Op1 | Op2` is of type `Qubit[] => Unit`
- `cond ? Op1 | Op3` is of type `Qubit[] => Unit is Adj`
- `cond ? Op2 | Op3` is of type `Qubit[] => Unit is Ctl`

See the section on [subtyping](#) for more detail.

# Comparative Expressions

3/29/2021 • 2 minutes to read • [Edit Online](#)

## Equality Comparison

Equality comparison (`==`) and inequality comparison (`!=`) is currently limited to the following data types: `Int`, `BigInt`, `Double`, `String`, `Bool`, `Result`, `Pauli`, and `Qubit`. The comparison for equality of arrays, tuples, ranges, user defined types, or callables is currently not supported.

Equality comparison for values of type `Qubit` evaluates whether the two expressions identify the same qubit. There is no notion of a quantum state in Q#; equality comparison in particular does *not* access, measure, or modify the quantum state of the qubits.

Equality comparison for `Double` values may be misleading due to rounding effects. For instance, the following comparison evaluates to `false` due to rounding errors: `49.0 * (1.0/49.0) == 1.0`.

### Discussion

In the future, we may support the comparisons of ranges, as well as arrays, tuples, and user defined types, provided their items support comparison. As for all types, the comparison will be by value, meaning two values are considered equal if all of their items are. For values of user defined type, additionally their type also needs to match. Future support for the comparison of values of type `Range` follows the same logic; they should be equal as long as they produce the same sequence of integers, meaning the two ranges

```
let r1 = 0..2..5; // generates the sequence 0,2,4
let r2 = 0..2..4; // generates the sequence 0,2,4
```

should be considered equal.

Conversely, there is a good reason not to allow the comparison of callables as the behavior would be ill-defined. Suppose we will introduce the capability to define functions locally via a possible syntax

```
let f1 = (x -> Bar(x)); // not yet supported
let f2 = Bar;
```

for some globally declared function `Bar`. The first line defines a new anonymous function that takes an argument `x` and invokes a function `Bar` with it and assigns it to the variable `f1`. The second line assigns the function `Bar` to `f2`. Since invoking `f1` and invoking `f2` will do the same thing, it should be possible to replace those with each other without changing the behavior of the program. This wouldn't be the case if the equality comparison for functions was supported and `f1 == f2` evaluates to `false`. If conversely `f1 == f2` were to evaluate to `true`, then this leads to the question of determining whether two callable will have the same side effects and evaluate to the same value for all inputs, which is not possible to reliably determine. Therefore, if we would like to be able to replace `f1` with `f2`, we can't allow equality comparisons for callables.

## Quantitative Comparison

The operators less-than (`<`), less-than-or-equal (`<=`), greater-than (`>`), and greater-than-or-equal (`>=`) define quantitative comparisons. They can only be applied to data types that support such comparisons; these are the same data types that can also support [arithmetic expressions](#).

# Logical Expressions

3/29/2021 • 2 minutes to read • [Edit Online](#)

Logical operators are expressed as keywords. Q# supports the standard logical operators *AND*( `and` ), *OR*( `or` ), and *NOT*( `not` ). There is currently no operator for a logical *XOR*. All of these operators act on operands of type `Bool`, and result in an expression of type `Bool` as well. As it is common in most languages, the evaluation of *AND* and *OR* short-circuits, meaning if the first expression of *OR* evaluates to `true`, the second expression is not evaluated, and the same holds if the first expression of *AND* evaluates to `false`. The behavior of conditional expressions in a sense is similar, in that only ever the condition and one of the two expressions is evaluated.

# Bitwise Expressions

3/29/2021 • 2 minutes to read • [Edit Online](#)

Bitwise operators are expressed as three non-letter characters. In addition to bitwise versions for *AND* (`&&`), *OR* (`|||`), and *NOT* (`~~~`), a bitwise *XOR* (`^^^`) exists as well. They expect operands of type `Int` or `BigInt`, and for binary operators, the type of both operands has to match. The type of the entire expression equals the type of the operand(s).

Additionally, left- and right-shift operators (`<<` and `>>` respectively) exist, multiplying or dividing the given left-hand-side (lhs) expression by powers of two. The expression `lhs << 3` shifts the bit representation of `lhs` by three, meaning `lhs` is multiplied by `2^3`, provided that is still within the valid range for the data type of `lhs`. The lhs may be of type `Int` or `BigInt`. The right-hand-side expression always has to be of type `Int`. The resulting expression will be of the same type as the lhs operand.

For left- and right-shift, the shift amount (i.e. the right-hand-side operand) must be greater than or equal to zero; the behavior for negative shift amounts is undefined. If the left-hand-side operand is of type `Int`, then the shift amount additionally needs to be smaller than 64; the behavior for larger shifts is undefined.

# Arithmetic Expressions

3/29/2021 • 2 minutes to read • [Edit Online](#)

Arithmetic operators are addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), negation (`-`), exponentiation (`^`). They can be applied to operands of type `Int`, `BigInt`, or `Double`. Additionally, for integral types (`Int` and `BigInt`) an operator computing the modulus (`%`) is available.

For binary operators, the type of both operands has to match, except for exponentiation; an exponent for a value of type `BigInt` always has to be of type `Int`. The type of the entire expression matches the type of the left operand. For exponentiation of `Int` and `BitInt`, the behavior is undefined if the exponent is negative or if it requires more than 32 bits to represent (i.e. it is larger than 2147483647).

Division and modulus for values of type `Int` and `BigInt` follow the following behavior for negative numbers:

A	B	A / B	A % B
5	2	2	1
5	-2	-2	1
-5	2	-2	-1
-5	-2	2	-1

That is, `a % b` will always have the same sign as `a`, and `b * (a / b) + a % b` will always equal `a`.

Q# does not support any automatic conversions between arithmetic data types - or any other data types for that matter. This is of importance especially for the `Result` data type, and facilitates to restrict how runtime information can propagate. It has the benefit of avoiding accidental errors e.g. related to precision loss.

# Concatenation

3/29/2021 • 2 minutes to read • [Edit Online](#)

Concatenations are supported for values of type `String` and arrays. In both cases they are expressed via the operator `+`. For instance, `"Hello " + "world!"` evaluates to `"Hello world!"`, and `[1,2,3] + [4,5,6]` evaluates to `[1,2,3,4,5,6]`.

Concatenating two arrays requires that both arrays are of the same type, in contrast to constructing an [array literal](#) where a common base type for all array items is determined. This is due to the fact that arrays are treated as [invariant](#). The type of the entire expression matches the type of the operands.

# Partial Application

3/29/2021 • 2 minutes to read • [Edit Online](#)

Callables are declared at a global scope and by default can be used anywhere in the same project and in a project that references the assembly in which they are declared. However, often there is a need to construct a callable for use in a local context only. Q# currently provides one rather powerful mechanism to construct new callables on the fly: partial applications.

Partial application refers to that some of the argument items to a callable are provided while others are still missing as indicated by an underscore. The result is a new callable value that takes the remaining argument items, combines them with the already given ones, and invokes the original callable. Naturally, partial application preserves the characteristics of a callable, i.e. a callable constructed by partial application supports the same functors as the original callable.

Q# allows any subset of the parameters to be left unspecified, not just a final sequence, which ties in more naturally with the design to have each callable take and return exactly one value. For a function `Foo` whose argument type is `(Int, (Double, Bool), Int)` for instance, `Foo(_, (1.0, _), 1)` is a function that takes an argument of type `(Int, (Bool))`, which is the same as an argument of type `(Int, Bool)`, see [this section](#).

Because partial application of an operation does not actually evaluate the operation, it has no impact on the quantum state. This means that building a new operation from existing operations and computed data may be done in a function; this is useful in many adaptive quantum algorithms and in defining new control flow constructs.

# Functor Application

3/29/2021 • 2 minutes to read • [Edit Online](#)

Functors are factories that allow to access particular specialization implementations of a callable. Q# currently supports two functors; the `Adjoint` and the `controlled`, both of which can be applied to operations that provide the necessary specialization(s).

The `controlled` and `Adjoint` functors commute; if `ApplyUnitary` is an operation that supports both functors, then there is no difference between `Controlled Adjoint ApplyUnitary` and `Adjoint Controlled ApplyUnitary`. Both have the same type and upon invocation execute the implementation defined for the `controlled adjoint specialization`.

## Adjoint Functor

If the operation `ApplyUnitary` defines a unitary transformation  $U$  of the quantum state, `Adjoint ApplyUnitary` accesses the implementation of  $U^\dagger$ . The `Adjoint` functor is its own inverse, since  $(U^\dagger)^\dagger = U$  by definition; i.e. `Adjoint Adjoint ApplyUnitary` is the same as `ApplyUnitary`.

The expression `Adjoint ApplyUnitary` is an operation of the same type as `ApplyUnitary`; it has the same argument and return type and supports the same functors. Like any operation, it can be invoked with an argument of suitable type. The following expression will apply the `adjoint specialization` of `ApplyUnitary` to an argument `arg`:

```
Adjoint ApplyUnitary(arg)
```

## Controlled Functor

For an operation `ApplyUnitary` that defines a unitary transformation  $U$  of the quantum state, `Controlled ApplyUnitary` accesses the implementation that applies  $U$  conditional on all qubits in an array of control qubits being in the  $|1\rangle$  state.

The expression `Controlled ApplyUnitary` is an operation with the same return type and [operation characteristics](#) as `ApplyUnitary`, meaning it supports the same functors. It takes an argument of type `(Qubit[], <TIn>)`, where `<TIn>` should be replaced with the argument type of `ApplyUnitary`, taking [singleton tuple equivalence](#) into account.

OPERATION	ARGUMENT TYPE	CONTROLLED ARGUMENT TYPE
X	<code>Qubit</code>	<code>(Qubit[], Qubit)</code>
SWAP	<code>(Qubit, Qubit)</code>	<code>(Qubit[], (Qubit, Qubit))</code>
ApplyQFT	<code>LittleEndian</code>	<code>(Qubit[], LittleEndian)</code>

Concretely, if `cs` contains an array of qubits, `q1` and `q2` are two qubits, and the operation `SWAP` is as defined [here](#), then the following expression exchanges the state of `q1` and `q2` if all qubits in `cs` are in the  $|1\rangle$  state:

```
Controlled SWAP(cs, (q1, q2))
```

## Discussion

Conditionally applying an operation based on the control qubits being in another state than a zero-state may be achieved by applying the appropriate adjointable transformation to the control qubits before invocation, and applying its inverse after. Conditioning the transformation on all control qubits being in the  $|0\rangle$  state, for example, can be achieved by applying the  $\boxed{x}$  operation before and after. This can be conveniently expressed using a [conjugation](#). Nonetheless, the verbosity of such a construct may merit additional support for a more compact syntax in the future.

# Item Access

3/29/2021 • 2 minutes to read • [Edit Online](#)

Q# supports item access for array items and for items in user defined types. In both cases, the access is read-only, i.e. the value cannot be changed without creating a new instance using a [copy-and-update expression](#).

## Array Item Access and Array Slicing

Given an array expression and an expression of type `Int` or `Range`, a new expression may be formed using the array item access operator consisting of `[` and `]`.

If the expression inside the brackets is of type `Int` then the new expression will contain the array item at that index.

For instance, if `arr` is of type `Double[]` and contains five or more items, then `arr[4]` is an expression of type `Double`.

If the expression inside the brackets is of type `Range` then the new expression will contain an array of all items indexed by the specified `Range`. If the `Range` is empty, then the resulting array will be empty.

For instance,

```
let arr = [10, 11, 36, 49];
let ten = arr[0]; // contains the value 10
let odds = arr[1..2..4]; // contains the value [11, 49]
let reverse = arr[...-1...]; // contains the value [49, 36, 11, 10]
```

In the last line, the start and end value of the range have been omitted for convenience; see [this section](#) for more detail.

If the array expression is not a simple identifier, it must be enclosed in parentheses in order to extract an item or a slice, see also the section on [precedence](#). For instance, if `arr1` and `arr2` are both arrays of integers, an item from the concatenation would be expressed as `(arr1 + arr2)[13]`.

All arrays in Q# are zero-based. That is, the first element of an array `arr` is always `arr[0]`. An exception will be thrown at runtime if the index or one of the indices used for slicing is outside the bounds of the array, i.e. if it is less than zero or larger or equal to the length of the array.

## Item Access for User Defined Types

[This section](#) describes how to define custom types, containing one or more named or anonymous items.

The contained items can be accessed via their name or by deconstruction, illustrated by the following statements that may be used as part of a operation or function implementation:

```
let complex = Complex(1.,0.); // create a value of type Complex
let (re, _) = complex!;      // item access via deconstruction
let im = complex::Imaginary; // item access via name
```

The item access operator (`::`) retrieves named items. While named items can be accessed by their name or via deconstruction, anonymous items can only be accessed by the latter. Since deconstruction relies on all of the contained items, the usage of anonymous items is discouraged when these items need to be accessed outside the compilation unit in which the type is defined.

Access via deconstruction makes use of the unwrap operator (`!`). That operator will return a tuple of all contained items, where the shape of the tuple matches the one defined in the declaration, and a single item tuple is equivalent to the item itself (see [this section](#)).

For example, for a value `nested` of type `Nested` defined as follows

```
newtype Nested = (Double, (ItemName : Int, String));
```

the expression `nested!` return a value of type `(Double, (Int, String))`.

The `!` operator has lower [precedence](#) than both item access operators, but higher precedence than any other operator. A complete list of precedences can be found [here](#).

# Contextual and Omitted Expressions

3/29/2021 • 2 minutes to read • [Edit Online](#)

The usage of item names in [copy-and-update expressions](#) without having to qualify them is an example for an expression that is only valid in a certain context.

Furthermore, expressions can be omitted when they can be inferred and automatically inserted by the compiler, as it is the case in [evaluate-and-reassign statements](#).

There is one more example for both; open-ended ranges are valid only within a certain context, and the compiler will translate them into normal `Range` expressions during compilation by inferring suitable boundaries.

A value of type `Range` generates a sequence of integers, specified by a start, optionally a step, and an end value. For example, the `Range` literal expressions `1..3` generates the sequence 1,2,3, and the expression `3..-1..1` generates the sequence 3,2,1. Ranges can be used for example to create a new array from an existing one by slicing:

```
let arr = [1,2,3,4];
let slice1 = arr[1..2..4]; // contains [2,4]
let slice2 = arr[2..-1..0]; // contains [3,2,1]
```

No infinite ranges exist in Q#, such that start and end value always need to be specified, except when a `Range` is used to slice an array. In that case, the start and/or end value of the range can reasonably be inferred.

Looking at the array slicing expressions above, it is reasonable for the compiler to assume that the intended range end should be the index of the last item in the array if the step size is positive. If the step size on the other hand is negative, then the range end likely should be the index of the first item in the array, i.e. `0`. The converse holds for the start of the range.

To summarize, if the range start value is omitted, then the inferred start value

- is zero if no step is specified or the specified step is positive,
- is the length of the array minus one if the specified step is negative.

If the range end value is omitted, then the inferred end value

- is the length of array minus one if no step is specified or the specified step is positive, and
- is zero if the specified step is negative.

Q# hence allows to use open-ended ranges within array slicing expressions:

```
let arr = [1,2,3,4,5,6];
let slice1 = arr[3...];      // slice1 is [4,5,6];
let slice2 = arr[0..2...];  // slice2 is [1,3,5];
let slice3 = arr[...2];    // slice3 is [1,2,3];
let slice4 = arr[...2..3];  // slice4 is [1,3];
let slice5 = arr[...2...];  // slice5 is [1,3,5];
let slice7 = arr[4...-2...]; // slice7 is [5,3,1];
let slice8 = arr[...-1..3]; // slice8 is [6,5,4];
let slice9 = arr[...-1...]; // slice9 is [6,5,4,3,2,1];
let slice10 = arr[...];    // slice10 is [1,2,3,4,5,6];
```

Since the information whether the range step is positive or negative is runtime information, the compiler inserts a suitable expression that will be evaluated at runtime. For omitted end values, the inserted expression is

`step < 0 ? 0 | Length(arr)-1`, and for omitted start values it is `step < 0 ? Length(arr)-1 | 0`, where `step` is the expression given for the range step, or `1` if no step is specified.

# Literals

3/29/2021 • 6 minutes to read • [Edit Online](#)

## Unit Literal

The only existing literal for the `Unit` type is the value `()`.

The `unit` value is commonly used as an argument to callables, e.g. either because no other arguments need to be passed or to delay execution. It is also used as return value when no other value needs to be returned, which is in particular the case for unitary operations, i.e. operations that support the `Adjoint` and/or the `Controlled` functor.

## Int Literals

Value literals for the `Int` type can be expressed in binary, octal, decimal, or hexadecimal representation. Literals expressed in binary are prefixed with `0b`, with `0o` for octal, and with `0x` for hexadecimal. There is no prefix for the commonly used decimal representation.

REPRESENTATION	VALUE LITERAL
Binary	<code>0b101010</code>
Octal	<code>0o52</code>
Decimal	<code>42</code>
Hexadecimal	<code>0x2a</code>

## BigInt Literals

Value literals for the `BigInt` type are always postfixed with `L` and can be expressed in binary, octal, decimal, or hexadecimal representation. Literals expressed in binary are prefixed with `0b`, with `0o` for octal, and with `0x` for hexadecimal. There is no prefix for the commonly used decimal representation.

REPRESENTATION	VALUE LITERAL
Binary	<code>0b101010L</code>
Octal	<code>0o52L</code>
Decimal	<code>42L</code>
Hexadecimal	<code>0x2aL</code>

## Double Literals

Value literals for the `Double` type can be expressed in standard or scientific notation.

REPRESENTATION	VALUE LITERAL
Standard	0.1973269804
Scientific	1.973269804e-1

If nothing follows after the decimal point, then the digit after dot may be omitted, e.g. `1.` is a valid `Double` literal and the same as `1.0`. Similarly, if the digits before the decimal point are all zero, then they may be omitted, e.g. `.1` is a valid `Double` literal and the same as `0.1`.

## Bool Literals

Existing literals for the `Bool` type are `true` and `false`.

## String Literals

A value literal for the `String` type is an arbitrary length sequence of Unicode characters enclosed in double quotes. Inside of a string, the back-slash character `\` may be used to escape a double quote character, and to insert a new-line as `\n`, a carriage return as `\r`, and a tab as `\t`.

The following are examples for valid string literals:

```
"This is a simple string."
\"This is a more complex string.\", she said.\n"
```

Q# also supports *interpolated strings*. An interpolated string is a string literal that may contain any number of interpolation expressions. These expressions can be of arbitrary types. Upon construction, the expressions are evaluated and their `String` representation is inserted at the corresponding location within the defined literal. Interpolation is enabled by prepending the special character `$` directly before the initial quote, i.e. without any white space between them.

For instance, if `res` is an expression that evaluates to `1`, then the second sentence in the following `String` literal will say "The result was 1.":

```
$"This is an interpolated string. The result was {res}."
```

## Qubit Literals

No literals for the `Qubit` type exist, since quantum memory is managed by the runtime. Values of type `Qubit` can hence only be obtained via [allocation](#).

Values of type `qubit` represent an opaque identifier by which a quantum bit, a.k.a. a qubit, can be addressed. The only operator they support is [equality comparison](#). See [this section](#) for more details on the `qubit` data type.

## Result Literals

Existing literals for the `Result` type are `Zero` and `One`.

Values of type `Result` represent the result of a binary quantum measurement. `Zero` indicates a projection onto the +1 eigenspace, `One` indicates a projection onto the -1 eigenspace.

## Pauli Literals

Existing literals for the `Pauli` type are `PauliI`, `PauliX`, `PauliY`, and `PauliZ`.

Values of type `Pauli` represent one of the four single-qubit **Pauli matrices**, with `PauliI` representing the identity. Values of type `Pauli` are commonly used to denote the axis for rotations and to specify with respect to which basis to measure.

## Range Literals

Value literals for the `Range` type are expressions of the form `start..step..stop`, where `start`, `step`, and `end` are expressions of type `Int`. If the step size is one, it may be omitted, i.e. `start..stop` is a valid `Range` literal and the same as `start..1..stop`.

Values of type `Range` represent a sequence of integers, where the first element in the sequence is `start`, and subsequent elements are obtained by adding `step` to the previous one, until `stop` is passed. `Range` values are inclusive at both ends; i.e. the last element of the range will be `stop` if the difference between `start` and `stop` is a multiple of `step`. A range may be empty if, for instance, `step` is positive and `stop < start`.

The following are examples for valid `Range` literals:

- `1..3` is the range 1, 2, 3.
- `2..2..5` is the range 2, 4.
- `2..2..6` is the range 2, 4, 6.
- `6..-2..2` is the range 6, 4, 2.
- `2..-2..1` is the range 2.
- `2..1` is the empty range.

See also the section on [contextual expressions](#).

## Array Literals

An `array` literal is a sequence of one or more expressions, separated by commas, enclosed in `[` and `]`, e.g. `[1,2,3]`. All expressions must have a [common base type](#), which will be the item type of the array.

Arrays of arbitrary length, and in particular empty arrays, may be created using a new array expression. Such an expression is of the form `new <ItemType>[expr]`, where `expr` can be any expression of type `Int` and `<ItemType>` is to be replace by the type of the array items.

For instance, `new Int[10]` creates an array of integers with containing ten items. The length of an array can be queries with the function `Length`. It is defined in the automatically opened namespace `Microsoft.Quantum.Core` and returns an `Int` value.

All items in the create array are set to the [default value](#) of the item type. Arrays containing qubits or callables must be properly initialized with non-default values before their elements may be safely used. Suitable initialization routines can be found in the `Microsoft.Quantum.Arrays` namespace.

## Tuple Literals

A `tuple` literal is a sequence of one or more expressions of any type, separated by commas, enclosed in `(` and `)`. The type of the tuple includes the information about each item type.

VALUE LITERAL	TYPE
<code>("Id", 0, 1.)</code>	<code>(String, Int, Double)</code>

VALUE LITERAL	TYPE
(PauliX,(3,1))	(Pauli, (Int, Int))

Tuples containing a single item are treated as identical to the item itself, both in type and value. We refer to this a [singleton tuple equivalence](#).

Tuples are used to bundle values together into a single value, making it easier to pass them around. This makes it possible that every callable takes exactly one input and returns exactly one output.

## Literals for User Defined Types

Values of a [user defined type](#) are constructed by invoking their constructor. A default constructor is automatically generated when [declaring the type](#). It is currently not possible to define custom constructors.

For instance, if `IntPair` has two items of type `Int`, then `IntPair(2, 3)` creates a new instance by invoking the default constructor.

## Operation Literals

No literals exist for values of [operation type](#); operations have to be [declared](#) on a global scope and new operations can be constructed locally using [partial application](#).

## Function Literals

No literals exist for values of [function type](#); functions have to be [declared](#) on a global scope and new functions can be constructed locally using [partial application](#).

## Default Values

TYPE	DEFAULT
<code>Unit</code>	<code>()</code>
<code>Int</code>	<code>0</code>
<code>BigInt</code>	<code>0L</code>
<code>Double</code>	<code>0.0</code>
<code>Bool</code>	<code>false</code>
<code>String</code>	<code>""</code>
<code>Qubit</code>	<i>invalid qubit</i>
<code>Result</code>	<code>Zero</code>
<code>Pauli</code>	<code>PauliI</code>
<code>Range</code>	empty range

TYPE	DEFAULT
Array	empty array
Tuple	all items are set to default values
User defined type	all items are set to default values
Operation	<i>invalid operation</i>
Function	<i>invalid function</i>

For qubits and callables, the default is an invalid reference that cannot be used without causing a runtime error.

# Type System

3/29/2021 • 2 minutes to read • [Edit Online](#)

With the focus for quantum algorithm being more towards what should be achieved rather than on a problem representation in terms of data structures, taking a more functional perspective on language design is a natural choice. At the same time, the type system is a powerful mechanism that can be leveraged for program analysis and other compile-time checks that facilitate formulating robust code.

All in all, the Q# type system is fairly minimalist, in the sense that there isn't an explicit notion of classes or interfaces as one might be used to from classical languages like C# or Java. We also take a somewhat pragmatic approach making incremental progress, such that certain constructs are not yet fully integrated into the type system. An example are functors, which can be used within expressions but don't yet have a representation in the type system. Correspondingly, they cannot currently be assigned or passed as arguments, similar as it is the case for type parametrized callables. We expect to make incremental progress in extending the type system to be more complete, and balance immediate needs with longer-term plans.

## Available Types

All types in Q# are [immutable](#).

TYPE	DESCRIPTION
<code>Unit</code>	Represents a singleton type whose only value is <code>()</code> .
<code>Int</code>	Represents a 64-bit signed integer. <a href="#">Values</a> range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
<code>BigInt</code>	Represents signed integer <a href="#">values</a> of any size.
<code>Double</code>	Represents a double-precision 64-bit floating-point number. <a href="#">Values</a> range from -1.79769313486232e308 to 1.79769313486232e308 as well as NaN (not a number).
<code>Bool</code>	Represents Boolean <a href="#">values</a> . Possible values are <code>true</code> or <code>false</code> .
<code>String</code>	Represents text as <a href="#">values</a> that consist of a sequence of UTF-16 code units.
<code>Qubit</code>	Represents an opaque identifier by which virtual quantum memory can be addressed. <a href="#">Values</a> of type <code>Qubit</code> are instantiated via allocation.
<code>Result</code>	Represents the result of a projective measurement onto the eigenspaces of a quantum operator with eigenvalues $\pm 1$ . Possible <a href="#">values</a> are <code>Zero</code> or <code>One</code> .
<code>Pauli</code>	Represents a single-qubit Pauli matrix. Possible <a href="#">values</a> are <code>PauliI</code> , <code>PauliX</code> , <code>PauliY</code> , or <code>PauliZ</code> .

TYPE	DESCRIPTION
Range	Represents an ordered sequence of equally spaced <code>Int</code> values. <code>Values</code> may represent sequences in ascending or descending order.
Array	Represents <code>values</code> that each contain a sequence of values of the same type.
Tuple	Represents <code>values</code> that each contain a fixed number of items of different types. Tuples containing a single element are equivalent to the element they contain.
User defined type	Represents a <a href="#">user defined type</a> consisting of named and anonymous items of different types. <code>Values</code> are instantiated by invoking the constructor.
Operation	Represents a non-deterministic <code>callable</code> that takes one (possibly tuple-valued) input argument returns one (possibly tuple-valued) output. Calls to operation <code>values</code> may have side effects and the output may vary for each call even when invoked with the same argument.
Function	Represents a deterministic <code>callable</code> that takes one (possibly tuple-valued) input argument returns one (possibly tuple-valued) output. Calls to function <code>values</code> do not have side effects and the output is will always be the same given the same input.

# Quantum-Specific Data Types

3/29/2021 • 2 minutes to read • [Edit Online](#)

In addition to the `Qubit` type explained in detail below, there are two other types that are somewhat specific to the quantum domain: `Pauli` and `Result`. Values of type `Pauli` specify a single-qubit Pauli operator; the possibilities are `PauliI`, `PauliX`, `PauliY`, and `PauliZ`. `Pauli` values are used primarily to specify the basis for a measurement. The `Result` type specifies the result of a quantum measurement. Q# mirrors most quantum hardware by providing measurements in products of single-qubit Pauli operators; a `Result` of `zero` indicates that the +1 eigenvalue was measured, and a `Result` of `one` indicates that the -1 eigenvalue was measured. That is, Q# represents eigenvalues by the power to which -1 is raised. This convention is more common in the quantum algorithms community, as it maps more closely to classical bits.

## Qubits

Q# treats qubits as opaque items that can be passed to both functions and operations, but that can only be interacted with by passing them to instructions that are native to the targeted quantum processor. Such instructions are always defined in the form of operations, since their intent is indeed to modify the quantum state. That functions cannot modify the quantum state despite that qubits can be passed as input arguments is enforced by the restriction that functions can only call other functions, and cannot call operations.

The Q# libraries are compiled against a standard set of intrinsic operations, meaning operations which have no definition for their implementation within the language. Upon targeting, the implementations that express them in terms of the instructions that are native to the execution target are linked in by the compiler. A Q# program thus combines these operations as defined by a target machine to create new, higher-level operations to express quantum computation. In this way, Q# makes it very easy to express the logic underlying quantum and hybrid quantum-classical algorithms, while also being very general with respect to the structure of a target machine and its realization of quantum state.

Within Q# itself, there is no type or construct in Q# that represents the quantum state. Instead, a qubit represents the smallest addressable physical unit in a quantum computer. As such, a qubit is a long-lived item, so Q# has no need for linear types. Importantly, we hence do not explicitly refer to the state within Q#, but rather describe how the state is transformed by the program, e.g., via application of operations such as `X` and `H`. Similar to how a graphics shader program accumulates a description of transformations to each vertex, a quantum program in Q# accumulates transformations to quantum states, represented as entirely opaque reference to the internal structure of a target machine.

A Q# program has no ability to introspect into the state of a qubit, and thus is entirely agnostic about what a quantum state is or on how it is realized. Rather, a program can call operations such as `Measure` to learn information about the quantum state of the computation.

# Immutability

3/29/2021 • 2 minutes to read • [Edit Online](#)

All types in Q# are *value types*. Q# does not have a concept of a reference or pointer. Instead, it allows to reassign a new value to a previously declared variable via a `set`-statement. There is no distinction in behavior between reassessments for, e.g., variables of type `Int` or variables of type `Int[]`. To give an explicit illustration, consider the following sequence of statements:

```
mutable arr1 = new Int[3];
let arr2 = arr1;
set arr1 w/= 0 <- 3;
```

The first statements instantiates a new arrays of integers `[0,0,0]` and assigns it to `arr1`. Line 2 assigns that value to a variable with name `arr2`. Line 3 then creates a new array instance based on `arr1` with the same values except for the value at index 0 which is set to 3. The newly created array is then assigned to the variable `arr1`. The last line makes use of the abbreviated syntax for [evaluate-and-reassign statements](#), and could equivalently have been written as `set arr1 = arr1 w/ 0 <- 1;`.

After executing the three statements, `arr1` will contain the value `[3,0,0]` while `arr2` remains unchanged and contains `[0,0,0]`.

Q# clearly thus distinguishes the mutability of a handle and the behavior of a type. Mutability within Q# is a concept that applies to a *symbol* rather than a type or value; it applies to the handle that allows one to access a value rather than to the value itself. It is *not* represented in the type system, implicitly or explicitly.

Of course, this is merely a description of the formally defined behavior; under the hood, the actual implementation uses a reference counting scheme to avoid copying memory as much as possible. The modification is specifically done in-place as long as there is only one currently valid handle that accesses a certain value.

# Operations and Functions

3/29/2021 • 3 minutes to read • [Edit Online](#)

As elaborated in more detail in the description of the [qubits](#), quantum computations are executed in the form of side effects of operations that are natively supported on the targeted quantum processor. These are in fact the only side effects in Q#; since all types are [immutable](#), there are no side effect that impact a value that is explicitly represented in Q#. Hence, as long as an implementation of a certain callable does not directly or indirectly call any of these natively implemented operations, its execution will always produce the same output given the same input.

Q# allows to explicitly split out such purely deterministic computations into *functions*. Since the set of natively supported instructions is not fixed and built into the language itself, but rather fully configurable and expressed as a Q# library, determinism is guaranteed by requiring that functions can only call other functions, but cannot call any operations. Additionally, native instructions that are not deterministic, e.g., because they impact the quantum state are represented as operations. With these two restrictions, function can be evaluated as soon as their input value is known, and in principle never need to be evaluated more than once for the same input.

Q# therefore distinguishes between two types of callables: operations and functions. All callables take a single (potentially tuple-valued) argument as input and produce a single value (tuple) as output. Syntactically, the operation type is expressed as `<TIn> => <TOut>` is `<Char>`, where `<TIn>` is to be replaced by the argument type, `<TOut>` is to be replaced by the return type, and `<Char>` is to be replaced by the [operation characteristics](#). If no characteristics need to be specified, the syntax simplifies to `<TIn> => <TOut>`. Similarly, function types are expressed as `<TIn> -> <TOut>`.

There is little difference between operations and functions beside this determinism guarantee. Both are first-class values that can be passed around freely; they can be used as return values or arguments to other callables, as illustrated by the example below.

```
function Pow<'T>(op : 'T => Unit, pow : Int) : 'T => Unit {
    return PowImpl(op, pow, _);
}
```

They can be instantiated based on a type parametrized definition such as, e.g., the [type parametrized](#) function `Pow` above, and they can be [partially applied](#) as done in Line 2 in the example.

## Operation Characteristics

In addition to the information about in- and output type, the operation type contains information about the characteristics of an operation. This information for example describes what functors are supported by the operation. Additionally, the internal representation also contains optimization relevant information that is inferred by the compiler.

The characteristics of an operation are a set of predefined and built-in labels. They are expressed in the form of a special expression that is part of the type signature. The expression consists either of one of the predefined sets of labels, or of a combination of characteristics expressions via a supported binary operator.

There are two predefined sets, `Adj` and `Ctl`.

- `Adj` is the set that contains a single label indicating that an operation is adjointable - meaning it supports the [Adjoint functor](#) and the applied quantum transformation can be "undone" (i.e. it can be inverted).
- `Ctl` is the set that contains a single label indicating that an operation is controllable - meaning it supports

the `Controlled` functor and its execution can be conditioned on the state of other qubits.

The two operators that are supported as part of characteristics expressions are the set union `+` and the set intersection `*`. In EBNF,

```
predefined = "Adj" | "Ctl";
characteristics = predefined
| "(" , characteristics, ")"
| characteristics ("+"|"*") characteristics;
```

As one would expect, `*` has higher precedence than `+` and both are left-associative. The type of a unitary operation for example is expressed as `<TIn> => <TOut>` is `Adj + Ctl` where `<TIn>` should be replaced with the type of the operation argument, and `<TOut>` with the type of the returned value.

### **Discussion**

Indicating the characteristics of an operation in this form has two major advantages; for one, new labels can be introduced without having exponentially many language keywords for all combinations of labels. Perhaps more importantly, using expressions to indicate the characteristics of an operation also permits to support parameterizations over operation characteristics in the future.

# Singleton Tuple Equivalence

3/29/2021 • 2 minutes to read • [Edit Online](#)

To avoid any ambiguity between tuples and parentheses that group sub-expressions, a tuple with a single element is considered to be equivalent to the contained item. This includes its type; for instance, the types `Int`, `(Int)`, and `((Int))` are treated as identical. The same holds for the values `5`, `(5)` and `((5))`, or for `(5, (6))` and `(5, 6)`. This equivalence applies for all purposes, including assignment. Since there is no dynamic dispatch or reflection in Q# and all types in Q# are resolvable at compile-time, singleton tuple equivalence can be readily implemented during compilation.

# Subtyping and Variance

3/29/2021 • 4 minutes to read • [Edit Online](#)

Q# supports only very few conversion mechanisms. Implicit conversions can happen only when applying binary operators, when evaluating conditional expressions, and when constructing an array literal. In these cases, a common supertype is determined and the necessary conversions are performed automatically. Aside from such implicit conversions, explicit conversation via function calls are possible and often necessary.

At present time, the only subtyping relation that exists applies to operations. Intuitively it makes sense that one should be allowed to substitute an operation that supports more than the required set of functors. Concretely, for any two concrete types  $T_{In}$  and  $T_{Out}$ , the subtyping relation is

```
(TIn => TOut) :>  
(TIn => TOut is Adj), (TIn => TOut is Ctl) :>  
(TIn => TOut is Adj + Ctl)
```

where  $A :> B$  indicates that  $B$  is a subtype of  $A$ . Phrased differently,  $B$  is more restrictive than  $A$  such that a value of type  $B$  can be used wherever a value of type  $A$  is required. If a callable relies on an argument (item) of being of type  $A$ , then an argument of type  $B$  can safely be substituted since it provides all the necessary capabilities.

This kind of polymorphism extends to tuples in that a tuple type  $B$  is a subtype of a tuple type  $A$  if it contains the same number of items and the type of each item is a subtype of the corresponding item type in  $A$ . This is known as *depth subtyping*. There is currently no support for *width subtyping*; there is no subtype relation between any two user defined types or a user defined type and any built-in type. The existence of the `unwrap` operator that allows to extract a tuple containing all named and anonymous items prevents this.

## Discussion

Looking at callables, if a callable processes an argument of type  $A$ , then it is also capable of processing an argument of type  $B$ . If a callable is passed as an argument to another callable, then it has to be capable of processing anything that the type signature requires. This means that if the callable needs to be able to process an argument of type  $B$ , any callable that is capable of processing a more general argument of type  $A$  can safely be passed. Conversely, we expect that if we require that the passed callable returns an a value of type  $A$ , then the promise to return a value of type  $B$  is sufficient since that value will provide all necessary capabilities.

The operation or function type is *contravariant* in its argument type and *covariant* in its return type.  $A :> B$  hence implies that for any concrete type  $T_1$ ,

```
(B → T1) :> (A → T1), and  
(T1 → A) :> (T1 → B)
```

where  $\rightarrow$  here can mean either a function or operation, and we omit any annotations for characteristics.

Substituting  $A$  with  $(B \rightarrow T_2)$  and  $(T_2 \rightarrow A)$  respectively, and substituting  $B$  with  $(A \rightarrow T_2)$  and  $(T_2 \rightarrow B)$  respectively leads to the conclusion that for any concrete type  $T_2$ ,

```
((A → T2) → T1) :> ((B → T2) → T1), and  
((T2 → B) → T1) :> ((T2 → A) → T1), and  
(T1 → (B → T2)) :> (T1 → (A → T2)), and  
(T1 → (T2 → A)) :> (T1 → (T2 → B))
```

By induction, it follows that every additional indirection reverses the variance of the argument type, and leaves the variance of the return type unchanged.

### Discussion

This also makes it clear what the variance behavior of arrays needs to be; retrieving items via an item access operator corresponds to invoking a function of type `(Int -> TItem)`, where `TItem` is the type of the elements in the array. Since this function is implicitly passed when passing an array, it follows that arrays need to be covariant in their item type. The same considerations also hold for tuples, which are immutable and thus covariant with respect to each item type. If arrays weren't immutable, the existence of a construct that would allow to set items in an array and thus takes an argument of type `TItem` would imply that arrays also need to be contravariant. The only option for data types that support getting and setting items is hence to be *invariant*, meaning there is no subtyping relation whatsoever; `B[]` is *not* a subtype of `A[]` even if `B` is a subtype of `A`.

Despite that arrays in Q# are [immutable](#), they are invariant rather than covariant. This, e.g., means that a value of type `(Qubit => Unit is Adj)[]` cannot be passed to a callable that requires an argument of type `(Qubit => Unit)[]`. Keeping arrays invariant allows for more flexibility related to how arrays are handled and optimized in the runtime, but it may be possible to revise that in the future.

# Type Parameterizations

3/29/2021 • 5 minutes to read • [Edit Online](#)

Q# supports type-parameterized operations and functions. The Q# standard libraries make heavy use of type parameterized callables to provide a host of useful abstractions, including functions like `Mapped` and `Fold` that are familiar from functional languages.

## Discussion

To motivate the concept of type parameterizations, consider the example of the function `Mapped`, which applies a given function to each value in an array and returns a new array with the computed values. This functionality can be perfectly described without specifying the item types of the in- and output array. Since the exact types do not change the implementation of the function `Mapped`, it makes sense that it should be possible to define this implementation for arbitrary item types; we want to define a *factory* or *template* that given the concrete types for the items in the in- and output array returns the corresponding function implementation. This notion is formalized in the form of type parameters.

Any operation or function declaration may specify one or more type parameters that can be used as the types or part of the types of the callable's input and/or output. The exception are entry points, which must be concrete and cannot be type parameterized. Type parameter names start with a tick ('') and may appear multiple times in the input and output types. All arguments that correspond to the same type parameter in the callable signature must be of the same type.

A type parametrized callable needs to be concretized — i.e. provided with the necessary type argument(s) — before it can be assigned or passed as argument, such that all type parameters can be replaced with concrete types. A type is considered to be concrete if it is either one of the built-in types, a user defined type, or if it is concrete within the current scope. The following example illustrates what it means for a type to be concrete within the current scope, and is explained in more detail below:

```
function Mapped<'T1, 'T2> (
    mapper : 'T1 -> 'T2,
    array : 'T1[]
) : 'T2[] {

    mutable mapped = new 'T2[Length(array)];
    for (i in IndexRange(array)) {
        set mapped w/= i <- mapper(array[i]);
    }
    return mapped;
}

function AllCControlled<'T3> (
    ops : ('T3 => Unit)[]
) : ((Bool, 'T3) => Unit)[] {

    return Mapped(CControlled<'T3>, ops);
}
```

The function `CControlled` is defined in the `Microsoft.Quantum.Canon` namespace. It takes an operation `op` of type `'TIn => Unit` as argument and returns a new operation of type `(Bool, 'TIn) => Unit` that applies the original operation provided a classical bit (of type `Bool`) is set to true; this is often referred to as the classically controlled version of `op`.

The function `Mapped` takes an array of an arbitrary item type `'T1` as argument, applies the given `mapper` function to each item and returns a new array of type `'T2[]` containing the mapped items. It is defined in the `Microsoft.Quantum.Array` namespace. For the purpose of the example, the type parameters are numbered to avoid making the discussion more confusing by giving the type parameters in both functions the same name. This is not necessary; type parameters for different callables may have the same name, and the chosen name is only visible and relevant within the definition of that callable.

The function `AllCControlled` takes an array of operations and returns a new array containing the classically controlled versions of these operations. The call of `Mapped` resolves its type parameter `'T1` to `'T3 => Unit`, and its type parameter `'T2` to `(Bool, 'T3) => Unit`. The resolving type arguments are inferred by the compiler based on the type of the given argument. We say that they are *implicitly* defined by the argument of the call expression. Type arguments can also be specified explicitly as it is done for `CControlled` in the same line. The explicit concretization `CControlled<'T3>` is necessary when the type arguments cannot be inferred.

The type `'T3` is concrete within the context of `AllCControlled`, since it is known for each *invocation* of `AllCControlled`. That means that as soon as the entry point of the program - which cannot be type parametrized - is known, so is the concrete type `'T3` for each call to `AllCControlled`, such that a suitable implementation for that particular type resolution can be generated; once the entry point to a program is known, all usages of type parameters can be eliminated at compile-time. We refer to this process as *monomorphization*.

A couple of restrictions are needed to ensure that this can indeed be done at compile-time opposed to only at run time.

## Discussion

Consider the following example:

```
operation Foo<'TArg> (
    op : 'TArg => Unit,
    arg : 'TArg
) : Unit {

    let cbit = RandomInt(2) == 0;
    Foo(CControlled(op), (cbit, arg));
}
```

Ignoring that an invocation of `Foo` will result in an infinite loop, it serves the purpose of illustration. `Foo` invokes itself with the classically controlled version of the original operation `op` that has been passed in as well as a tuple containing a random classical bit in addition to the original argument.

For each iteration in the recursion, the type parameter `'TArg` of the next call is resolved to `(Bool, 'TArg)`, where `'TArg` is the type parameter of the current call. Concretely, suppose `Foo` is invoked with the operation `H` and an argument `arg` of type `Qubit`. `Foo` will then invoke itself with a type argument `(Bool, Qubit)`, which will then invoke `Foo` with a type argument `(Bool, (Bool, Qubit))`, and so on. Clearly, in this case `Foo` cannot be monomorphized at compile-time.

Additional restrictions apply to cycles in the call graph that involve only type parametrized callables. Each callable needs to be invoked with the same set of type arguments after traversing the cycle.

## Discussion

It would be possible to be less restrictive and require that for each callable in the cycle, there is a finite number of cycles after which it is invoked with the original set of type arguments, such as it is the case for the following function:

```
function Bar<'T1,'T2,'T3>(a1:'T1, a2:'T2, a3:'T3) : Unit{
    Bar<'T2,'T3,'T1>(a2, a3, a1);
}
```

For simplicity, the more restrictive requirement is enforced. Note that for cycles that involve at least one concrete callable without any type parameter, such a callable will ensure that the type parametrized callables within that cycle are always called with a fixed set of type arguments.

# Grammar

3/29/2021 • 2 minutes to read • [Edit Online](#)

A reference implementation of the Q# grammar is available in the [ANTLR4](#) format. The grammar source files are listed below:

- [QSharpLexer.g4](#) describes the lexical structure of Q#.
- [QSharpParser.g4](#) describes the syntax of Q#.

The Q# grammar uses *actions* and *semantic predicates*. These features allow grammars to include custom source code in the ANTLR-generated parser, which means that the code needs to be written in the same language as the ANTLR target language. If you are using the Q# grammar to generate parsers in a language other than Java, you may need to update the code used by the actions and semantic predicates to match the target language. Target-specific code is marked by curly braces `{ }`  in the grammar.

# Quantum simulators

3/5/2021 • 2 minutes to read • [Edit Online](#)

Quantum simulators are software programs that run on classical computers and act as the *target machine* for a Q# program, making it possible to run and test quantum programs in an environment that predicts how qubits will react to different operations. This article describes which quantum simulators are included with the Quantum Development Kit (QDK), and different ways that you can pass a Q# program to the quantum simulators, for example, via the command line or by using driver code written in a classical language.

## The Quantum Development Kit (QDK) quantum simulators

The quantum simulator is responsible for providing implementations of quantum primitives for an algorithm. This includes primitive operations such as `H`, `CNOT`, and `Measure`, as well as qubit management and tracking. The QDK includes different classes of quantum simulators representing different ways of simulating the same quantum algorithm.

Each type of quantum simulator can provide different implementations of these primitives. For example, the [full state simulator](#) runs the quantum algorithm by fully simulating the [quantum state vector](#), whereas the [quantum computer trace simulator](#) doesn't consider the actual quantum state at all. Rather, it tracks gate, qubit, and other resource usage for the algorithm.

### Quantum machine classes

In the future, the QDK will define additional quantum machine classes to support other types of simulation and to support running on quantum hardware. Allowing the algorithm to stay constant while varying the underlying machine implementation makes it easy to test and debug an algorithm in simulation and then run it on real hardware with confidence that the algorithm hasn't changed.

The QDK includes several quantum machine classes, all defined in the `Microsoft.Quantum.Simulation.Simulators` namespace.

SIMULATOR	CLASS	DESCRIPTION
Full state simulator	<code>QuantumSimulator</code>	Runs and debugs quantum algorithms, and is limited to about 30 qubits.
Simple resources estimator	<code>ResourcesEstimator</code>	Performs a top level analysis of the resources needed to run a quantum algorithm, and supports thousands of qubits.
Trace-based resource estimator	<code>QCTraceSimulator</code>	Runs advanced analysis of resources consumptions for the algorithm's entire call-graph, and supports thousands of qubits.
Toffoli simulator	<code>ToffoliSimulator</code>	Simulates quantum algorithms that are limited to <code>X</code> , <code>CNOT</code> , and multi-controlled <code>X</code> quantum operations, and supports million of qubits.

## Invoking the quantum simulator

In [Ways to run a Q# program](#), three ways of passing the Q# code to the quantum simulator are demonstrated:

- Using the command line
- Using a Python host program
- Using a C# host program

Quantum machines are instances of normal .NET classes, so they are created by invoking their constructor, just like any .NET class. How you do this depends on how you run the Q# program.

## Next steps

- For details about how to invoke target machines for Q# programs in different environments, see [Ways to run a Q# program](#).

# Quantum Development Kit (QDK) full state simulator

7/9/2021 • 2 minutes to read • [Edit Online](#)

The QDK provides a full state simulator that simulates a quantum machine on your local computer. You can use the full state simulator to run and debug quantum algorithms written in Q#, utilizing up to 30 qubits. The full state simulator in its functionality is similar to the quantum simulator used in the [IQ\\$Ui\ranglengle\\$](#) platform from Microsoft Research.

## Invoking and running the full state simulator

You expose the full state simulator via the `QuantumSimulator` class. For additional details, see [Ways to run a Q# program](#).

### Invoking the simulator from C#

Create an instance of the `QuantumSimulator` class and then pass it to the `Run` method of a quantum operation, along with any additional parameters.

```
use var sim = new QuantumSimulator()
{
    var res = myOperation.Run(sim).Result;
    //...
}
```

Because the `QuantumSimulator` class implements the `IDisposable` interface, you must call the `Dispose` method once you do not need the instance of the simulator anymore. The best way to do this is to wrap the simulator declaration and operations within a `using` statement, which automatically calls the `Dispose` method.

### Invoking the simulator from Python

Use the `simulate()` method from the Q# Python library with the imported Q# operation:

```
qubit_result = myOperation.simulate()
```

### Invoking the simulator from the command line

When running a Q# program from the command line, the full state simulator is the default target machine. Optionally, you can use the `--simulator` (or `-s` shortcut) parameter to specify the desired target machine. Both of the following commands run a program using the full state simulator.

```
dotnet run
dotnet run -s QuantumSimulator
```

### Invoking the simulator from Jupyter Notebooks

Use the IQ# magic command `%simulate` to run the Q# operation.

```
%simulate myOperation
```

## Seeding the simulator

By default, the full state simulator uses a random number generator to simulate quantum randomness. For testing purposes, it is sometimes useful to have deterministic results. In a C# program, you can accomplish this by providing a seed for the random number generator in the `QuantumSimulator` constructor via the `randomNumberGeneratorSeed` parameter.

```
use var sim = new QuantumSimulator(randomNumberGeneratorSeed: 42)
{
    var res = myOperationTest.Run(sim).Result;
    //...
}
```

## Simulator options

The behavior of the full state simulator can be adjusted via the following parameters to the C# constructor:

- `throwOnReleasingQubitsNotInZeroState` : The simulator can warn you if qubits have not been returned to the `zero` state before release by throwing an exception. Resetting or measuring qubits before release is required by the Q# spec - not doing so may lead to computational errors! The default is `true`.
- `randomNumberGeneratorSeed` : Obtain deterministic behavior by seeding the simulator as described above.
- `disableBorrowing` : If you don't want to use `borrowed qubits` for this simulation, you can disable this feature by setting this parameter to `true`. Borrowed qubits will instead be replaced with regular clean qubits. The default is `false`.

The code below shows a possible configuration of the parameters.

```
var sim = new QuantumSimulator (
    throwOnReleasingQubitsNotInZeroState: false,
    randomNumberGeneratorSeed: 42,
    disableBorrowing: true
)
```

## Configuring threads

The full state simulator uses `OpenMP` to parallelize the linear algebra required. By default, OpenMP uses all available hardware threads, which means that programs with small numbers of qubits often runs slowly because the coordination that is required dwarfs the actual work. You can fix this by setting the environment variable `OMP_NUM_THREADS` to a small number. As a rule of thumb, configure one thread for up to four qubits, and then one additional thread per qubit. You might need to adjust the variable depending on your algorithm.

## See also

- [Quantum resources estimator](#)
- [Quantum Toffoli simulator](#)
- [Quantum trace simulator](#)

# Quantum Development Kit (QDK) resources estimator

7/9/2021 • 6 minutes to read • [Edit Online](#)

As the name implies, the `ResourcesEstimator` class estimates the resources required to run a given instance of a Q# operation on a quantum computer. It accomplishes this by running the quantum operation without actually simulating the state of a quantum computer; for this reason, it is able to estimate resources for Q# operations that use thousands of qubits, provided that the classical part of the code runs in a reasonable time.

The resources estimator is built on top of the [Quantum trace simulator](#), which provides a richer set of metrics and tools to help debug Q# programs.

## Invoking and running the resources estimator

The resources estimator can be thought of as another type of simulation target. As such, there exists a variety of methods with which to invoke this target, which are presented below. For additional details, you can also have a look at the [Ways to run a Q# program](#).

### Invoking the resources estimator from C#

As with other targets, you first create an instance of the `ResourcesEstimator` class and then pass it as the first parameter of an operation's `Run` method.

Note that, unlike the `QuantumSimulator` class, the `ResourcesEstimator` class does not implement the `IDisposable` interface, and thus you do not need to enclose it within a C# `using` statement.

```
using System;
using Microsoft.Quantum.Simulation.Simulators;

namespace Quantum.MyProgram
{
    class Driver
    {
        static void Main(string[] args)
        {
            ResourcesEstimator estimator = new ResourcesEstimator();
            MyOperation.Run(estimator).Wait();
            Console.WriteLine(estimator.ToTSV());
        }
    }
}
```

As the example shows, `ResourcesEstimator` provides the `ToTSV()` method, which generates a table with tab-separated values (TSV). You can save the table to a file or display it to the console for analysis. The following is a sample output from the preceding program:

Metric	Sum	Max
CNOT	1000	1000
QubitClifford	1000	1000
R	0	0
Measure	4002	4002
T	0	0
Depth	0	0
Width	2	2
QubitCount	2	2
BorrowedWidth	0	0

For more information about the collected metrics, see the description of the [Reported Metrics](#). The default configuration used for the `ResourcesEstimator` counts only `T` gates. Other configurations can be set by creating and customizing a `QCTraceSimulator` instance.

#### NOTE

A `ResourcesEstimator` instance does not reset its calculations on every run. If you use the same instance to run another operation, it aggregates the new results with the existing results in the `Sum` column. Additionally, the `Max` column will contain the highest value encountered so far for each metric. If you need to reset calculations between runs, create a new instance for every run.

### Invoking the resources estimator from Python

Use the `estimate_resources()` method from the Python library with the imported Q# operation:

```
qubit_result = myOperation.estimate_resources()
```

### Invoking the resources estimator from the command line

When running a Q# program from the command line, use the `--simulator` (or `-s` shortcut) parameter to specify the `ResourcesEstimator` target machine. The following command runs a program using the resources estimator:

```
dotnet run -s ResourcesEstimator
```

### Invoking the resources estimator from Jupyter Notebooks

Use the IQ# magic command `%estimate` to run the Q# operation.

```
%estimate myOperation
```

## Programmatically retrieving the estimated data

In addition to a TSV table, you can programmatically retrieve the resources estimated during the run via the `Data` property of the resources estimator. The `Data` property provides a `System.DataTable` instance with three columns: `Metric`, `Sum`, and `Max`, indexed by the metrics' names.

The following code shows how to retrieve and print the total number of `QubitClifford`, `T` and `CNOT` operations used by a Q# operation:

```

using System;
using Microsoft.Quantum.Simulation.Simulators;

namespace Quantum.MyProgram
{
    class Driver
    {
        static void Main(string[] args)
        {
            ResourcesEstimator estimator = new ResourcesEstimator();
            MyOperation.Run(estimator).Wait();

            var data = estimator.Data;
            Console.WriteLine($"QubitCliffords: {data.Rows.Find("QubitClifford")["Sum"]}");
            Console.WriteLine($"Ts: {data.Rows.Find("T")["Sum"]}");
            Console.WriteLine($"CNOTs: {data.Rows.Find("CNOT")["Sum"]}");
        }
    }
}

```

## Metrics Reported

The resources estimator tracks the following metrics:

METRIC	DESCRIPTION
CNOT	The run count of <code>CNOT</code> operations (also known as Controlled Pauli X operations).
QubitClifford	The run count of any single qubit Clifford and Pauli operations.
Measure	The run count of any measurements.
R	The run count of any single-qubit rotations, excluding <code>T</code> , Clifford and Pauli operations.
T	The run count of <code>T</code> operations and their conjugates, including the <code>T</code> operations, $T_x = H.T.H$ , and $T_y = H.y.T.H.y$ .
Depth	Depth of the quantum circuit run by the Q# operation (see <a href="#">below</a> ). By default, the depth metric only counts <code>T</code> gates. For more details, see <a href="#">Depth Counter</a> .
Width	Width of the quantum circuit run by the Q# operation (see <a href="#">below</a> ). By default, the depth metric only counts <code>T</code> gates. For more details, see <a href="#">Width Counter</a> .
QubitCount	The lower bound for the maximum number of qubits allocated during the run of the Q# operation. This metric might not be compatible with <b>Depth</b> (see below).
BorrowedWidth	The maximum number of qubits borrowed inside the Q# operation.

## Depth, Width, and QubitCount

Reported Depth and Width estimates are compatible with each other. (Previously both numbers were achievable, but different circuits would be required for Depth and for Width.) Currently both metrics in this pair can be achieved by the same circuit at the same time.

The following metrics are reported:

**Depth:** For the root operation - time it takes to execute it assuming specific gate times. For operations called or subsequent operations - time difference between latest qubit availability time at the beginning and the end of the operation.

**Width:** For the root operation - number of qubits actually used to execute it (and operation it calls). For operations called or subsequent operations - how many more qubits were used in addition to the qubits already used at the beginning of the operation.

Please note, that reused qubits do not contribute to this number. For example, if a few qubits have been released before operation *A* starts, and all qubit demanded by this operation *A* (and operations called from *A*) were satisfied by reusing previously released qubits, the **Width** of operation *A* is reported as 0. Successfully borrowed qubits do not contribute to the Width either.

**QubitCount:** For the root operation - minimum number of qubits that would be sufficient to execute this operation (and operations called from it). For operations called or subsequent operations - minimum number of qubits that would be sufficient to execute this operation separately. This number doesn't include input qubits. It does include borrowed qubits.

Two modes of operation are supported. Mode is selected by setting `QCTraceSimulatorConfiguration.OptimizeDepth`.

**OptimizeDepth=true:** QubitManager is discouraged from qubit reuse and allocates new qubit every time it is asked for a qubit. For the root operation **Depth** becomes the minimum depth (lower bound). Compatible **Width** is reported for this depth (both can be achieved at the same time). Note that this width will likely be not optimal given this depth. **QubitCount** may be lower than **Width** for the root operation because it assumes reuse.

**OptimizeDepth=false:** QubitManager is encouraged to reuse qubits and will reuse released qubits before allocating new ones. For the root operation **Width** becomes the minimal width (lower bound). Compatible **Depth** is reported on which it can be achieved. **QubitCount** will be the same as **Width** for the root operation assuming no borrowing.

## Providing the probability of measurement outcomes

You can use [AssertMeasurementProbability operation](#) from the [Microsoft.Quantum.Diagnostics namespace](#) namespace to provide information about the expected probability of a measurement operation. For more information, see [Quantum Trace Simulator](#)

## See also

- [Quantum trace simulator](#)
- [Quantum Toffoli simulator](#)
- [Quantum full state simulator](#)

# Microsoft Quantum Development Kit (QDK) quantum trace simulator

5/27/2021 • 3 minutes to read • [Edit Online](#)

The QDK `QCTraceSimulator` class runs a quantum program without actually simulating the state of a quantum computer. For this reason, the quantum trace simulator is able to run quantum programs that use thousands of qubits. It is useful for two main purposes:

- Debugging classical code that is part of a quantum program.
- Estimating the resources required to run a given instance of a quantum program on a quantum computer. In fact, the [Resources estimator](#), which provides a more limited set of metrics, is built upon the trace simulator.

## Invoking the quantum trace simulator

You can use the quantum trace simulator to run any Q# operation.

As with other target machines, you first create an instance of the `QCTraceSimulator` class and then pass it as the first parameter of an operation's `Run` method.

Note that, unlike the `QuantumSimulator` class, the `QCTraceSimulator` class does not implement the `IDisposable` interface, and thus you do not need to enclose it within a `using` statement.

```
using Microsoft.Quantum.Simulation.Core;
using Microsoft.Quantum.Simulation.Simulators;
using Microsoft.Quantum.Simulation.Simulators.QCTraceSimulators;

namespace Quantum.MyProgram
{
    class Driver
    {
        static void Main(string[] args)
        {
            QCTraceSimulator sim = new QCTraceSimulator();
            var res = MyQuantumProgram.Run(sim).Result;
            System.Console.WriteLine("Press any key to continue...");
            System.Console.ReadKey();
        }
    }
}
```

## Providing the probability of measurement outcomes

Because the quantum trace simulator does not simulate the actual quantum state, it cannot calculate the probability of measurement outcomes within an operation.

Therefore, if an operation includes measurements, you must explicitly provide these probabilities using the [AssertMeasurementProbability operation](#) operation from the `Microsoft.Quantum.Diagnostics` namespace. The following example illustrates this:

```

operation TeleportQubit(source : Qubit, target : Qubit) : Unit {
    use qubit = Qubit();
    H(qubit);
    CNOT(qubit, target);
    CNOT(source, qubit);
    H(source);

    AssertMeasurementProbability([PauliZ], [source], Zero, 0.5, "Outcomes must be equally likely", 1e-5);
    AssertMeasurementProbability([PauliZ], [q], Zero, 0.5, "Outcomes must be equally likely", 1e-5);

    if M(source) == One { Z(target); X(source); }
    if M(q) == One { X(target); X(q); }
}

```

When the quantum trace simulator encounters `AssertMeasurementProbability` it records that measuring `Pauliz` on `source` and `q` should give an outcome of `Zero`, with probability `0.5`. When it runs the `M` operation later, it finds the recorded values of the outcome probabilities, and `M` returns `Zero` or `One`, with probability `0.5`. When the same code runs on a simulator that keeps track of the quantum state, that simulator checks that the provided probabilities in `AssertMeasurementProbability` are correct.

Note that if there is at least one measurement operation that is not annotated using

`AssertMeasurementProbability`, the simulator throws an `UnconstrainedMeasurementException`.

## Quantum trace simulator tools

The QDK includes five tools that you can use with the quantum trace simulator to detect bugs in your programs and perform quantum program resource estimates:

TOOL	DESCRIPTION
<a href="#">Distinct inputs checker</a>	Checks for potential conflicts with shared qubits
<a href="#">Invalidated qubits use checker</a>	Checks if the program applies an operation to a qubit that is already released
<a href="#">Primitive operations counter</a>	Counts the number of primitives used by every operation invoked in a quantum program
<a href="#">Depth counter</a>	Gathers counts that represent the lower bound of the depth of every operation invoked in a quantum program
<a href="#">Width counter</a>	Counts the number of qubits allocated and borrowed by each operation in a quantum program

Each of these tools is enabled by setting appropriate flags in `QCTraceSimulatorConfiguration` and then passing the configuration to the `QCTraceSimulator` declaration. For information on using each of these tools, see the links in the preceding list. For more information about configuring `QCTraceSimulator`, see [QCTraceSimulatorConfiguration](#).

## QCTraceSimulator methods

`QCTraceSimulator` has several built-in methods to retrieve the values of the metrics tracked during a quantum operation. Examples of the `QCTraceSimulator.GetMetric` and the `QCTraceSimulator.ToCSV` methods can be found in the [Primitive operations counter](#), [Depth counter](#), and [Width counter](#) articles. For more information on all available methods, see `QCTraceSimulator` in the Q# API reference.

## See also

- [Quantum resources estimator](#)
- [Quantum Toffoli simulator](#)
- [Quantum full state simulator](#)

# Quantum trace simulator: distinct inputs checker

3/5/2021 • 2 minutes to read • [Edit Online](#)

The distinct inputs checker is a part of the Quantum Development Kit [Quantum trace simulator](#). You can use it to detect potential bugs in the code caused by conflicts with shared qubits.

## Conflicts with shared qubits

Consider the following piece of Q# code to illustrate the issues detected by the distinct inputs checker:

```
operation ApplyBoth(
    q1 : Qubit,
    q2 : Qubit,
    op1 : (Qubit => Unit),
    op2 : (Qubit => Unit))
: Unit {
    op1(q1);
    op2(q2);
}
```

When you look at this program, you can assume that the order in which it calls `op1` and `op2` does not matter, because `q1` and `q2` are different qubits and operations acting on different qubits commute.

Now, consider this example:

```
operation ApplyWithNonDistinctInputs() : Unit {
    use qubits = Qubit[3];
    let op1 = CNOT(_, qubits[1]);
    let op2 = CNOT(qubits[1],_);
    ApplyBoth(qubits[0], qubits[2], op1, op2);
}
```

Note that `op1` and `op2` are both obtained using partial application and share a qubit. When you call `ApplyBoth` in this example, the result of the operation depends on the order of `op1` and `op2` inside `ApplyBoth` - not what you would expect to happen. When you enable the distinct inputs checker, it detects such situations and throws a `DistinctInputsCheckerException`. For more information, see [DistinctInputsCheckerException](#) in the Q# API library.

## Invoking the distinct inputs checker

To run the quantum trace simulator with the distinct inputs checker you must create a `QCTraceSimulatorConfiguration` instance, set the `UseDistinctInputsChecker` property to `true`, and then create a new `QCTraceSimulator` instance with `QCTraceSimulatorConfiguration` as the parameter.

```
var config = new QCTraceSimulatorConfiguration();
config.UseDistinctInputsChecker = true;
var sim = new QCTraceSimulator(config);
```

## Using the distinct inputs checker in a C# host program

The following is an example of C# host program that uses the quantum trace simulator with the distinct inputs

checker enabled:

```
using Microsoft.Quantum.Simulation.Core;
using Microsoft.Quantum.Simulation.Simulators;
using Microsoft.Quantum.Simulation.Simulators.QCTraceSimulators;

namespace Quantum.MyProgram
{
    class Driver
    {
        static void Main(string[] args)
        {
            var traceSimCfg = new QCTraceSimulatorConfiguration();
            traceSimCfg.UseDistinctInputsChecker = true; //enables distinct inputs checker
            QCTraceSimulator sim = new QCTraceSimulator(traceSimCfg);
            var res = MyQuantumProgram.Run().Result;
            System.Console.WriteLine("Press any key to continue...");
            System.Console.ReadKey();
        }
    }
}
```

## See also

- The Quantum Development Kit [Quantum trace simulator](#) overview.
- The [QCTraceSimulator](#) API reference.
- The [QCTraceSimulatorConfiguration](#) API reference.
- The [DistinctInputsCheckerException](#) API reference.

# Quantum trace simulator: invalidated qubits use checker

4/17/2021 • 2 minutes to read • [Edit Online](#)

The invalidated qubits use checker is a part of the Quantum Development Kit [Quantum trace simulator](#). You can use it to detect potential bugs in the code caused by invalid qubits.

## Invalid qubits

Consider the following piece of Q# code to illustrate the issues detected by the invalidated qubits use checker:

```
operation UseReleasedQubit() : Unit {
    mutable q = new Qubit[1];
    use ans = Qubit() {
        set q w/= 0 <- ans;
    }
    H(q[0]);
}
```

When you apply the `H` operation to `q[0]`, it points to an already released qubit, which can cause undefined behavior. When the Invalidated Qubits Use Checker is enabled, it throws the exception

`InvalidatedQubitsUseCheckerException` if the program applies an operation to an already released qubit. For more information, see [InvalidatedQubitsUseCheckerException](#).

## Invoking the invalidated qubits use checker

To run the quantum trace simulator with the invalidated qubits use checker you must create a `QCTraceSimulatorConfiguration` instance, set the `UseInvalidatedQubitsUseChecker` property to `true`, and then create a new `QCTraceSimulator` instance with `QCTraceSimulatorConfiguration` as the parameter.

```
var config = new QCTraceSimulatorConfiguration();
config.UseInvalidatedQubitsUseChecker = true;
var sim = new QCTraceSimulator(config);
```

## Using the invalidated qubits use checker in a C# host program

The following is an example of C# host programs that uses the quantum trace simulator with the invalidated qubits use checker enabled:

```
using Microsoft.Quantum.Simulation.Core;
using Microsoft.Quantum.Simulation.Simulators;
using Microsoft.Quantum.Simulation.Simulators.QCTraceSimulators;

namespace Quantum.MyProgram
{
    class Driver
    {
        static void Main(string[] args)
        {
            var traceSimCfg = new QCTraceSimulatorConfiguration();
            traceSimCfg.UseInvalidateQubitsUseChecker = true; // enables UseInvalidateQubitsUseChecker
            QCTraceSimulator sim = new QCTraceSimulator(traceSimCfg);
            var res = MyQuantumProgram.Run().Result;
            System.Console.WriteLine("Press any key to continue...");
            System.Console.ReadKey();
        }
    }
}
```

## See also

- The Quantum Development Kit [Quantum trace simulator](#) overview.
- The [QCTraceSimulator](#) API reference.
- The [QCTraceSimulatorConfiguration](#) API reference.
- The [InvalidateQubitsUseCheckerException](#) API reference.

# Quantum trace simulator: primitive operations counter

5/27/2021 • 2 minutes to read • [Edit Online](#)

The primitive operation counter is a part of the Quantum Development Kit [Quantum trace simulator](#). It counts the number of primitive processes used by every operation invoked in a quantum program.

All [Microsoft.Quantum.Intrinsic namespace](#) operations are expressed in terms of single-qubit rotations, T operations, single-qubit Clifford operations, CNOT operations, and measurements of multi-qubit Pauli observables. The Primitive Operations Counter aggregates and collects statistics over all the edges of the operation's [call graph](#).

## Invoking the primitive operation counter

To run the quantum trace simulator with the primitive operation counter, you must create a [QCTraceSimulatorConfiguration](#) instance, set the `UsePrimitiveOperationsCounter` property to `true`, and then create a new [QCTraceSimulator](#) instance with the `QCTraceSimulatorConfiguration` as the parameter.

```
var config = new QCTraceSimulatorConfiguration();
config.UsePrimitiveOperationsCounter = true;
var sim = new QCTraceSimulator(config);
```

## Using the primitive operation counter in a C# host program

The C# example that follows in this section counts how many [T operation](#) operations are needed to implement the [CCNOT operation](#) operation, based on the following Q# sample code:

```
open Microsoft.Quantum.Intrinsic;
operation ApplySampleWithCCNOT() : Unit {

    use qubits = Qubit[3];
    CCNOT(qubits[0], qubits[1], qubits[2]);
    T(qubits[0]);
}
```

To check that `ccnot` requires seven `T` operations and that `ApplySampleWithCCNOT` runs eight `T` operations, use the following C# code:

```
// using Microsoft.Quantum.Simulation.Simulators.QCTraceSimulators;
// using System.Diagnostics;
var config = new QCTraceSimulatorConfiguration();
config.UsePrimitiveOperationsCounter = true;
var sim = new QCTraceSimulator(config);
var res = ApplySampleWithCCNOT.Run(sim).Result;

double tCountAll = sim.GetMetric<ApplySampleWithCCNOT>(PrimitiveOperationsGroupsNames.T);
double tCount = sim.GetMetric<Primitive.CCNOT, ApplySampleWithCCNOT>(PrimitiveOperationsGroupsNames.T);
```

The first part of the program runs `ApplySampleWithCCNOT`. The second part uses the `QCTraceSimulator.GetMetric` method to retrieve the number of `T` operations run by `ApplySampleWithCCNOT`:

When you call `GetMetric` with two type parameters, it returns the value of the metric associated with a given call graph edge. In the preceding example, the program calls the `Primitive.CCNOT` operation within `ApplySampleWithCCNOT` and therefore the call graph contains the edge `<Primitive.CCNOT, ApplySampleWithCCNOT>`.

To retrieve the number of `CNOT` operations used, add the following line:

```
double cxCount = sim.GetMetric<Primitive.CCNOT, ApplySampleWithCCNOT>(PrimitiveOperationsGroupsNames.CX);
```

Finally, you can output all the statistics collected by the Primitive Operations Counter in CSV format using the following:

```
string csvSummary = sim.ToCSV()[MetricsCountersNames.primitiveOperationsCounter];
```

## See also

- The Quantum Development Kit [Quantum trace simulator](#) overview.
- The [QCTraceSimulator](#) API reference.
- The [QCTraceSimulatorConfiguration](#) API reference.
- The [PrimitiveOperationsGroupsNames](#) API reference.

# Quantum trace simulator: depth counter

5/27/2021 • 2 minutes to read • [Edit Online](#)

The depth counter is a part of the Quantum Development Kit [Quantum trace simulator](#). You can use it to gather counts that represent the lower bound of the depth of every operation invoked in a quantum program.

## Depth values

By default, all operations have a depth of **0** except the `T` operation, which has a depth of 1. This means that by default, only the `T` depth of operations is computed (which is often desirable). The depth counter aggregates and collects statistics over all the edges of the operation's [call graph](#).

All [Microsoft.Quantum.Intrinsic namespace](#) operations are expressed in terms of single-qubit rotations, `T` operation operations, single-qubit Clifford operations, `CNOT` operation operations, and measurements of multi-qubit Pauli observables. Users can set the depth for each of the primitive operations via the `gateTimes` field of `QCTraceSimulatorConfiguration`.

## Invoking the depth counter

To run the quantum trace simulator with the depth counter, you must create a `QCTraceSimulatorConfiguration` instance, set its `UseDepthCounter` property to `true`, and then create a new `QCTraceSimulator` instance with `QCTraceSimulatorConfiguration` as the parameter.

```
var config = new QCTraceSimulatorConfiguration();
config.UseDepthCounter = true;
var sim = new QCTraceSimulator(config);
```

## Using the depth counter in a C# host program

The C# example that follows in this section computes the `T` depth of the `CCNOT` operation, based on the following Q# sample code:

```
open Microsoft.Quantum.Intrinsic;

operation ApplySampleWithCCNOT() : Unit {
    use qubits = Qubit[3];
    CCNOT(qubits[0], qubits[1], qubits[2]);
    T(qubits[0]);
}
```

To check that `CCNOT` has `T` depth 5 and `ApplySampleWithCCNOT` has `T` depth 6, use the following C# code:

```
using Microsoft.Quantum.Simulation.Simulators.QCTraceSimulators;
using System.Diagnostics;
var config = new QCTraceSimulatorConfiguration();
config.UseDepthCounter = true;
var sim = new QCTraceSimulator(config);
var res = ApplySampleWithCCNOT.Run(sim).Result;

double tDepth = sim.GetMetric<Intrinsic.CCNOT, ApplySampleWithCCNOT>(DepthCounter.Metrics.Depth);
double tDepthAll = sim.GetMetric<ApplySampleWithCCNOT>(DepthCounter.Metrics.Depth);
```

The first part of the program runs `ApplySampleWithCCNOT`. The second part uses the `GetMetric` method to retrieve the `T` depth of `ccnot` and `ApplySampleWithCCNOT`.

Finally, you can output all the statistics collected by the depth counter in CSV format using the following:

```
string csvSummary = sim.ToCSV()[MetricsCountersNames.depthCounter];
```

## See also

- The Quantum Development Kit [Quantum trace simulator](#) overview.
- The [QCTraceSimulator](#) API reference.
- The [QCTraceSimulatorConfiguration](#) API reference.
- The [MetricsNames.DepthCounter](#) API reference.

# Quantum trace simulator: width counter

5/27/2021 • 2 minutes to read • [Edit Online](#)

The width counter is a part of the Quantum Development Kit [Quantum trace simulator](#). You can use it to count the number of qubits allocated and borrowed by each operation in a Q# program. Some primitive operations can allocate extra qubits, for example, multiply controlled  operations or controlled  operations.

## Invoking the width counter

To run the quantum trace simulator with the width counter, you must create a [QCTraceSimulatorConfiguration](#) instance, set the `UseWidthCounter` property to `true`, and then create a new [QCTraceSimulator](#) instance with the `QCTraceSimulatorConfiguration` as the parameter.

```
var config = new QCTraceSimulatorConfiguration();
config.UseWidthCounter = true;
var sim = new QCTraceSimulator(config);
```

## Using the width counter in a C# host program

The C# example that follows in this section computes the number of extra qubits allocated by the implementation of a multiply controlled [X operation](#) operation, based on the following Q# sample code:

```
open Microsoft.Quantum.Intrinsic;
open Microsoft.Quantum.Arrays;
operation ApplyMultiControlledX( numberOfQubits : Int ) : Unit {
    use qubits = Qubit[numberOfQubits];
    Controlled X (Rest(qubits), Head(qubits));
}
```

The multiply controlled [X operation](#) operation acts on a total of five qubits, allocates two [auxiliary qubits](#), and has an input width of 5. Use the following C# program to verify the counts:

```
var config = new QCTraceSimulatorConfiguration();
config.UseWidthCounter = true;
var sim = new QCTraceSimulator(config);
int totalNumberOfQubits = 5;
var res = ApplyMultiControlledX.Run(sim, totalNumberOfQubits).Result;

double allocatedQubits =
    sim.GetMetric<Primitive.X, ApplyMultiControlledX>(
        WidthCounter.Metrics.ExtraWidth,
        functor: OperationFunctor.Controlled);

double inputWidth =
    sim.GetMetric<Primitive.X, ApplyMultiControlledX>(
        WidthCounter.Metrics.InputWidth,
        functor: OperationFunctor.Controlled);
```

The first part of the program runs the `ApplyMultiControlledX` operation. The second part uses the `QCTraceSimulator.GetMetric` method to retrieve the number of allocated qubits as well as the number of qubits that the `Controlled X` operation received as input.

Finally, you can output all the statistics collected by the width counter in CSV format using the following:

```
string csvSummary = sim.ToCSV()[MetricsCountersNames.widthCounter];
```

## See also

- The Quantum Development Kit [Quantum trace simulator](#) overview.
- The [QCTraceSimulator](#) API reference.
- The [QCTraceSimulatorConfiguration](#) API reference.
- The [MetricsNames.WidthCounter](#) API reference.

# Quantum Development Kit (QDK) Toffoli simulator

5/27/2021 • 2 minutes to read • [Edit Online](#)

The QDK Toffoli simulator is a special-purpose simulator with a limited scope and only supports `X`, `CNOT`, and multi-controlled `X` quantum operations. All classical logic and computations are available.

While the Toffoli simulator is more restricted in functionality than the [full state simulator](#), it has the advantage of being able to simulate far more qubits. The Toffoli simulator can be used with millions of qubits, while the full state simulator is limited to about 30 qubits. This is useful, for example, with oracles that evaluate Boolean functions - they can be implemented using the limited set of supported algorithms and tested on a large number of qubits.

## Invoking the Toffoli simulator

You expose the Toffoli simulator via the `ToffoliSimulator` class. For additional details, see [Ways to run a Q# program](#).

### Invoking the Toffoli simulator from C#

As with other target machines, you first create an instance of the `ToffoliSimulator` class and then pass it as the first parameter of an operation's `Run` method.

Note that, unlike the `QuantumSimulator` class, the `ToffoliSimulator` class does not implement the `IDisposable` interface, and thus you do not need to enclose it within a `using` statement.

```
var sim = new ToffoliSimulator();
var res = myOperation.Run(sim).Result;
//...
```

### Invoking the Toffoli simulator from Python

Use the `toffoli_simulate()` method from the Python library with the imported Q# operation:

```
qubit_result = myOperation.toffoli_simulate()
```

### Invoking the Toffoli simulator from the command line

When running a Q# program from the command line, use the `--simulator` (or `-s` shortcut) parameter to specify the Toffoli simulator target machine. The following command runs a program using the resources estimator:

```
dotnet run -s ToffoliSimulator
```

### Invoking the Toffoli simulator from Jupyter Notebooks

Use the IQ# magic command `%toffoli` to run the Q# operation.

```
%toffoli myOperation
```

## Supported operations

The Toffoli simulator supports:

- Rotations and exponentiated Paulis, such as `R` and `Exp`, when the resulting operation equals `x` or the identity matrix.
- Measurement and `assert` operations, but only in the Pauli `z` basis. Note that a measurement operation's probability is always either `0` or `1`; there is no randomness in the Toffoli simulator.
- `DumpMachine` and `DumpRegister` functions. Both functions output the current `z`-basis state of each qubit, one qubit per line.

## Specifying the number of qubits

By default, a `ToffoliSimulator` instance allocates space for 65,536 qubits. If your algorithm requires more qubits than this, you can specify the qubit count by providing a value for the `qubitCount` parameter to the constructor. Each additional qubit requires only one byte of memory, so there is no significant cost to overestimating the number of qubits you'll need.

For example:

```
var sim = new ToffoliSimulator(qubitCount: 1000000);
var res = myLargeOperation.Run(sim).Result;
```

## See also

- [Quantum Resources Estimator](#)
- [Quantum Trace simulator](#)
- [Quantum Full State simulator](#)

# Overview of Q# Libraries

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Quantum Development Kit (QDK) is provided with several libraries to make it easier to develop quantum applications in Q#. In this section of the documentation, we describe these libraries and how to use them in your programs.

- **Standard libraries:** This section describes the prelude, which defines the interface between Q# programs and target machines, and the canon, a Q# library that provides general-purpose operations and functions for use in writing Q# programs.
- **Quantum chemistry library:** This section describes the quantum chemistry library, which provides a data model for loading representations of fermionic Hamiltonians and quantum simulation operations and functions which act on these representations.
- **Quantum numerics library:** This section describes the quantum numerics library, which provides implementations for a host of mathematical functions. It supports integer (signed & unsigned) and fixed-point representations.
- **Quantum machine learning library:** This section describes the quantum machine learning library, which provides an implementation of the sequential classifiers that take advantage of quantum computing to understand data.

Sources of the libraries as well as code samples can be obtained from GitHub. See [Licensing](#) for further information. Note that package references ("binaries") are available also for the libraries and offer another way of including the libraries in projects. A convenient way of obtaining them is via [NuGet](#).

# Introduction to the Q# Standard Libraries

3/5/2021 • 2 minutes to read • [Edit Online](#)

Q# is supported by a range of different useful operations, functions, and user-defined types that comprise the *Q# standard libraries*. The [Microsoft.Quantum.Development.Kit](#) NuGet package installed during [installation and validation](#) provides the Q# compiler, and parts of the standard library that are implemented by the target machines. The [Microsoft.Quantum.Standard](#) package provides the portion of the Q# standard libraries that are portable across target machines.

The symbols defined by the Q# standard libraries are defined in much greater and more exhaustive detail in the [API documentation](#).

In the following sections, we will outline the most salient features of each part of the standard library and provide some context about how each feature might be used in practice.

# The Prelude

5/27/2021 • 12 minutes to read • [Edit Online](#)

The Q# compiler and the target machines included with the Quantum Development Kit provide a set of intrinsic functions and operations that can be used when writing quantum programs in Q#.

## Intrinsic Operations and Functions

The intrinsic operations defined in the standard library roughly fall into one of several categories:

- Essential classical functions, collected in the [Microsoft.Quantum.Core namespace](#) namespace.
- Operations representing unitaries composed of [Clifford](#) and [\\$T\\$ gates](#).
- Operations representing rotations about various operators.
- Operations implementing measurements.

Since the Clifford + \$T\$ gate set is [universal](#) for quantum computing, these operations suffice to approximately implement any quantum algorithm within negligibly small error. By providing rotations as well, Q# allows the programmer to work within the single qubit unitary and CNOT gate library. This library is much easier to think about because it does not require the programmer to directly express the Clifford + \$T\$ decomposition and because highly efficient methods exist for compiling single qubit unitaries into Clifford and \$T\$ gates (see [here](#) for more information).

Where possible, the operations defined in the prelude which act on qubits allow for applying the [Controlled](#) variant, such that the target machine will perform the appropriate decomposition.

Many of the functions and operations defined in this portion of the prelude are in the `@"microsoft.quantum.intrinsic"` namespace, such that most Q# source files will have an

`open Microsoft.Quantum.Intrinsic;` directive immediately following the initial namespace declaration. The [Microsoft.Quantum.Core namespace](#) namespace is automatically opened, so that functions such as [Length function](#) can be used without an `open` statement at all.

### Common Single-Qubit Unitary Operations

The prelude also defines many common [single-qubit operations](#). All of these operations allow both the [Controlled](#) and [Adjoint](#) functors.

#### Pauli Operators

The [X operation](#) operation implements the Pauli \$X\$ operator. This is sometimes also known as the [NOT](#) gate. It has signature `(Qubit => Unit is Adj + ct1)`. It corresponds to the single-qubit unitary:

```
\begin{equation} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{equation}
```

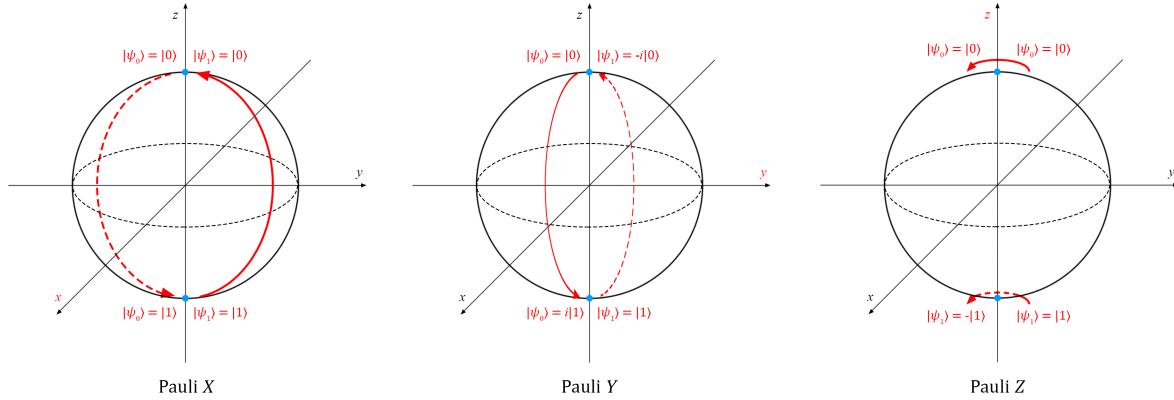
The [Y operation](#) operation implements the Pauli \$Y\$ operator. It has signature `(Qubit => Unit is Adj + Ctl)`. It corresponds to the single-qubit unitary:

```
\begin{equation} \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \end{equation}
```

The [Z operation](#) operation implements the Pauli \$Z\$ operator. It has signature `(Qubit => Unit is Adj + Ctl)`. It corresponds to the single-qubit unitary:

```
\begin{equation} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \end{equation}
```

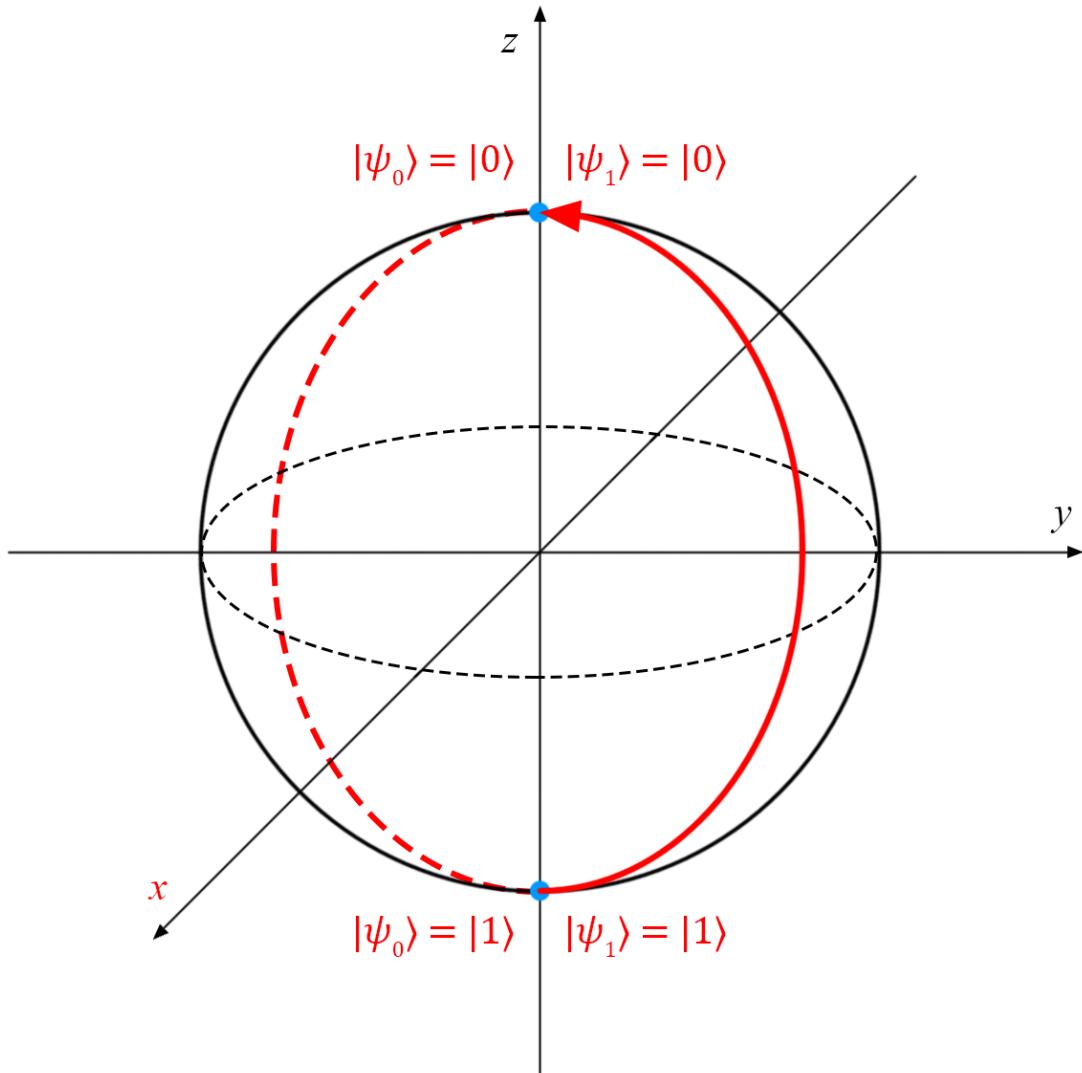
Below we see these transformations mapped to the **Bloch sphere** (the rotation axis in each case is highlighted red):



It is important to note that applying the same Pauli gate twice to the same qubit cancels out the operation (because you have now performed a full rotation of  $2\pi$  (360°) over the surface to the Bloch Sphere, thus arriving back at the starting point). This brings us to the following identity:

$\$ \$ X^2=Y^2=Z^2=\backslash boldone \$ \$$

This can be visualised on the Bloch sphere:



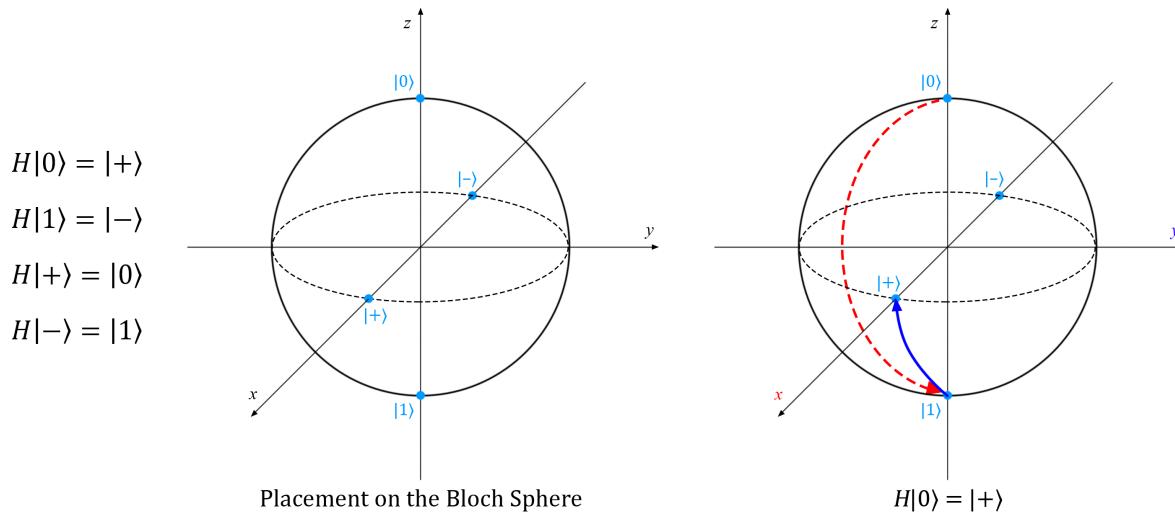
$$XX = X^2 = \mathbb{I}$$

## Other Single-Qubit Cliffords

The [H operation](#) implements the Hadamard gate. This interchanges the Pauli  $\text{X}$  and  $\text{Z}$  axes of the target qubit, such that  $\text{H}|\text{ket}{0}\rangle = |\text{ket}{+}\rangle$  and  $\text{H}|\text{ket}{+}\rangle = |\text{ket}{0}\rangle$ . It has signature `(Qubit => Unit is Adj + Ctl)`, and corresponds to the single-qubit unitary:

```
\begin{equation} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{equation} % FIXME: this currently uses the quadwhack hack.
```

The Hadamard gate is particularly important as it can be used to create a superposition of the  $|\text{ket}{0}\rangle$  and  $|\text{ket}{1}\rangle$  states. In the Bloch sphere representation, it is easiest to think of this as a rotation of  $|\text{ket}{\psi}\rangle$  around the  $x$ -axis by  $\pi$  radians ( $180^\circ$ ) followed by a (clockwise) rotation around the  $y$ -axis by  $\pi/2$  radians ( $90^\circ$ ):



The [S operation](#) implements the phase gate  $S$ . This is the matrix square root of the Pauli  $Z$  operation. That is,  $S^2 = Z$ . It has signature `(Qubit => Unit is Adj + Ctl)`, and corresponds to the single-qubit unitary:

```
\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} % FIXME: this currently uses the quadwhack hack.
```

## Rotations

In addition to the Pauli and Clifford operations above, the Q# prelude provides a variety of ways of expressing rotations. As described in [single-qubit operations](#), the ability to rotate is critical to quantum algorithms.

We start by recalling that we can express any single-qubit operation using the  $H$  and  $T$  gates, where  $H$  is the Hadamard operation, and where  $T = \exp(i\pi/4)$ . This is the square root of the [S operation](#) operation, such that  $T^2 = S$ . The  $T$  gate is in turn implemented by the [T operation](#) operation, and has signature `(Qubit => Unit is Adj + Ctl)`, indicating that it is a unitary operation on a single-qubit.

Even though this is in principle sufficient to describe any arbitrary single-qubit operation, different target machines may have more efficient representations for rotations about Pauli operators, such that the prelude includes a variety of ways to conveniently express such rotations. The most basic of these is the [R operation](#) operation, which implements a rotation around a specified Pauli axis,  $R(\sigma, \phi) = \exp(-i\phi\sigma/2)$ , where  $\sigma$  is a Pauli operator,  $\phi$  is an angle, and where  $\exp$  represents the matrix exponential. It has signature `((Pauli, Double, Qubit) => Unit is Adj + Ctl)`, where the first two parts of the input represent the classical arguments  $\sigma$  and  $\phi$  needed to specify the unitary operator  $R(\sigma, \phi)$ . We can partially apply  $\sigma$  and  $\phi$  to obtain an operation whose type is that of a single-qubit unitary. For example, `R(PauliZ, PI() / 4, _)` has type `(Qubit => Unit is Adj + Ctl)`.

## NOTE

The `R operation` operation divides the input angle by 2 and multiplies it by -1. For  $\$Z\$$  rotations, this means that the  $\$|\text{ket}\{0\}\$$  eigenstate is rotated by  $-\phi / 2$  and the  $\$|\text{ket}\{1\}\$$  eigenstate is rotated by  $\phi / 2$ , so that the  $\$|\text{ket}\{1\}\$$  eigenstate is rotated by  $\phi$  relative to the  $\$|\text{ket}\{0\}\$$  eigenstate.

In particular, this means that `T` and `R(PauliZ, PI() / 8, _)` differ only by an irrelevant [global phase](#). For this reason, `T` is sometimes known as the  $\$|\frac{\phi}{8}\$$ -gate.

Note also that rotating around `PauliI` simply applies a global phase of  $\phi / 2$ . While such phases are irrelevant, as argued in [the conceptual documents](#), they are relevant for controlled `PauliI` rotations.

Within quantum algorithms, it is often useful to express rotations as dyadic fractions, such that  $\phi = \pi k / 2^n$  for some  $k \in \mathbb{Z}$  and  $n \in \mathbb{N}$ . The `RFrac` operation implements a rotation around a specified Pauli axis using this convention. It differs from `R operation` in that the rotation angle is specified as two inputs of type `Int`, interpreted as a dyadic fraction. Thus, `RFrac` has signature `((Pauli, Int, Int, Qubit) => Unit is Adj + Ctl)`. It implements the single-qubit unitary  $\exp(i \pi k \sigma_z / 2^n)$ , where  $\sigma_z$  is the Pauli matrix corresponding to the first argument,  $k$  is the second argument, and  $n$  is the third argument. `RFrac(_,k,n,_)` is the same as `R(_, -πk/2^n, _)`; note that the angle is the *negative* of the fraction.

The `Rx` operation implements a rotation around the Pauli  $X$  axis. It has signature

`((Double, Qubit) => Unit is Adj + Ctl)`. `Rx(_, _)` is the same as `R(PauliX, _, _)`.

The `Ry` operation implements a rotation around the Pauli  $Y$  axis. It has signature

`((Double, Qubit) => Unit is Adj + Ctl)`. `Ry(_, _)` is the same as `R(PauliY, _, _)`.

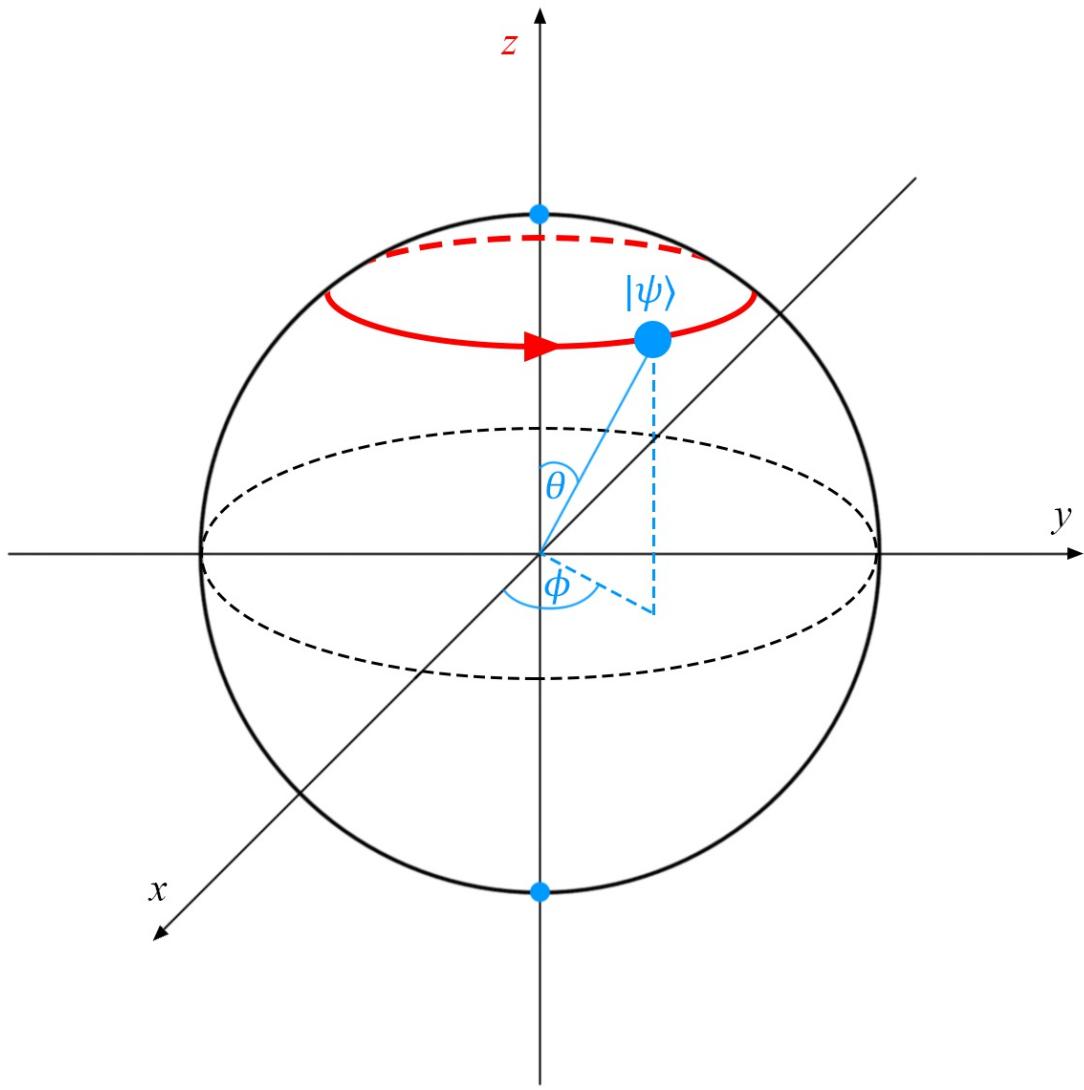
The `Rz` operation implements a rotation around the Pauli  $Z$  axis. It has signature

`((Double, Qubit) => Unit is Adj + Ctl)`. `Rz(_, _)` is the same as `R(PauliZ, _, _)`.

The `R1` operation implements a rotation by the given amount around  $\$|\text{ket}\{1\}\$$ , the  $-1$  eigenstate of  $\$Z\$$ . It has signature `((Double, Qubit) => Unit is Adj + Ctl)`. `R1(phi, _)` is the same as `R(PauliZ, phi, _)` followed by `R(PauliI, -phi, _)`.

The `R1Frac` operation implements a fractional rotation by the given amount around  $Z=1$  eigenstate. It has signature `((Int, Int, Qubit) => Unit is Adj + Ctl)`. `R1Frac(k, n, _)` is the same as `RFrac(PauliZ, -k.n+1, _)` followed by `RFrac(PauliI, k, n+1, _)`.

An example of a rotation operation (around the Pauli  $Z$  axis, in this instance) mapped onto the Bloch sphere is shown below:



### Multi-Qubit Operations

In addition to the single-qubit operations above, the prelude also defines several multi-qubit operations.

First, the [CNOT operation](#) operation performs a standard controlled-[NOT](#) gate,  $\begin{array}{l} \operatorname{CNOT} \\ \operatorname{mathrel{\coloneqq}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$ . It has signature  $((\text{Qubit}, \text{Qubit}) \Rightarrow \text{Unit} \text{ is Adj + Ctl})$ , representing that  $\operatorname{CNOT}$  acts unitarily on two individual qubits.  $\operatorname{CNOT}(q_1, q_2)$  is the same as  $(\text{Controlled } X)([q_1], q_2)$ . Since the [Controlled](#) functor allows for controlling on a register, we use the array literal  $[q_1]$  to indicate that we want only the one control.

The [CCNOT operation](#) operation performs doubly-controlled NOT gate, sometimes also known as the Toffoli gate. It has signature  $((\text{Qubit}, \text{Qubit}, \text{Qubit}) \Rightarrow \text{Unit} \text{ is Adj + Ctl})$ .  $\operatorname{CCNOT}(q_1, q_2, q_3)$  is the same as  $(\text{Controlled } X)([q_1, q_2], q_3)$ .

The [SWAP operation](#) operation swaps the quantum states of two qubits. That is, it implements the unitary matrix  $\begin{array}{l} \operatorname{SWAP} \\ \operatorname{mathrel{\coloneqq}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$ . It has signature  $((\text{Qubit}, \text{Qubit}) \Rightarrow \text{Unit} \text{ is Adj + Ctl})$ .  $\operatorname{SWAP}(q_1, q_2)$  is equivalent to  $\operatorname{CNOT}(q_1, q_2)$  followed by  $\operatorname{CNOT}(q_2, q_1)$  and then  $\operatorname{CNOT}(q_1, q_2)$ .

### NOTE

The SWAP gate is *not* the same as rearranging the elements of a variable with type `Qubit[]`. Applying `SWAP(q1, q2)` causes a change to the state of the qubits referred to by `q1` and `q2`, while `let swappedRegister = [q2, q1];` only affects how we refer to those qubits. Moreover, `(Controlled SWAP)([q0], (q1, q2))` allows for `SWAP` to be conditioned on the state of a third qubit, which we cannot represent by rearranging elements. The controlled-SWAP gate, also known as the Fredkin gate, is powerful enough to include all classical computation.

Finally, the prelude provides two operations for representing exponentials of multi-qubit Pauli operators. The `Exp operation` operation performs a rotation based on a tensor product of Pauli matrices, as represented by the multi-qubit unitary  $\begin{aligned} & \text{\textbackslash begin\{equation\} \operatornamename\{Exp\}(\text{\textbackslash vec\{\sigma\}}, \phi) \mathrel{:=} \exp\left(i \phi \right.} \\ & \left. \sigma_0 \otimes \sigma_1 \otimes \cdots \otimes \sigma_n \right), \end{aligned}$  where  $\text{\textbackslash vec\{\sigma\}} = (\sigma_0, \sigma_1, \dots, \sigma_n)$  is a sequence of single-qubit Pauli operators, and where  $\phi$  is an angle. The `Exp` rotation represents  $\text{\textbackslash vec\{\sigma\}}$  as an array of `Pauli` elements, such that it has signature `((Pauli[], Double, Qubit[]) => Unit is Adj + Ctl)`.

The `ExpFrac operation` operation performs the same rotation, using the dyadic fraction notation discussed above. It has signature `((Pauli[], Int, Int, Qubit[]) => Unit is Adj + Ctl)`.

### WARNING

Exponentials of the tensor product of Pauli operators are not the same as tensor products of the exponentials of Pauli operators. That is,  $e^{i(Z \otimes Z)\phi} \neq e^{iZ\phi} \otimes e^{iZ\phi}$ .

## Measurements

When measuring, the  $+1$  eigenvalue of the operator being measured corresponds to a `Zero` result, and the  $-1$  eigenvalue to a `One` result.

### NOTE

While this convention might seem odd, it has two very nice advantages. First, observing the outcome  $|\ket{0}|$  is represented by the `Result` value `Zero`, while observing  $|\ket{1}|$  corresponds to `One`. Second, we can write out that the eigenvalue  $\lambda$  corresponding to a result  $r$  is  $\lambda = (-1)^r$ .

Measurement operations support neither the `Adjoint` nor the `Controlled` functor.

The `Measure operation` operation performs a joint measurement of one or more qubits in the specified product of Pauli operators. If the Pauli array and qubit array are different lengths, then the operation fails. `Measure` has signature `((Pauli[], Qubit[]) => Result)`.

Note that a joint measurement is not the same as measuring each qubit individually. For example, consider the state  $|\ket{11}\rangle = |\ket{1}\otimes|\ket{1}\rangle = X\otimes X|\ket{00}\rangle$ . Measuring  $Z_0$  and  $Z_1$  each individually, we get the results  $r_0 = 1$  and  $r_1 = 1$ . Measuring  $Z_0 Z_1$ , however, we get the single result  $r_{\text{Joint}} = 0$ , representing that the parity of  $|\ket{11}\rangle$  is positive. Put differently,  $(-1)^{r_0 + r_1} = (-1)^{r_{\text{Joint}}}$ . Critically, since we *only* learn the parity from this measurement, any quantum information represented in the superposition between the two two-qubit states of positive parity,  $|\ket{00}\rangle$  and  $|\ket{11}\rangle$ , is preserved. This property will be essential later, as we discuss error correction.

For convenience, the prelude also provides two other operations for measuring qubits. First, since performing single-qubit measurements is quite common, the prelude defines a shorthand for this case. The `M operation` operation measures the Pauli `Z` operator on a single qubit, and has signature `(Qubit => Result)`. `M(q)` is equivalent to `Measure([Pauliz], [q])`.

The [MultiM operation](#) measures the Pauli `$Z$` operator *separately* on each of an array of qubits, returning the array of `Result` values obtained for each qubit. In some cases this can be optimized. It has signature (`Qubit[] => Result[]`) . `MultiM(qs)` is equivalent to:

```
mutable rs = new Result[Length(qs)];
for (index in 0..Length(qs)-1)
{
    set rs[index] = M(qs[index]);
}
return rs;
```

## Extension Functions and Operations

In addition, the prelude defines a rich set of mathematical and type conversion functions at the .NET level for use within Q# code. For instance, the [Microsoft.Quantum.Math namespace](#) defines useful operations such as [Sin function](#) and [Log function](#). The implementation provided by the Quantum Development Kit uses the classical .NET base class library, and thus may involve an additional communications round trip between quantum programs and their classical drivers. While this does not present a problem for a local simulator, this can be a performance issue when using a remote simulator or actual hardware as a target machine. That said, an individual target machine may mitigate this performance impact by overriding these operations with versions that are more efficient for that particular system.

### Math

The [Microsoft.Quantum.Math namespace](#) provides many useful functions from the .NET base class library's `System.Math` class. These functions can be used in the same manner as any other Q# functions:

```
open Microsoft.Quantum.Math;
// ...
let y = Sin(theta);
```

Where a .NET static method has been overloaded based on the type of its arguments, the corresponding Q# function is annotated with a suffix indicating the type of its input:

```
let x = AbsI(-3); // x : Int = 3
let y = AbsD(-PI()); // y : Double = 3.1415...
```

### Bitwise Operations

Finally, the [Microsoft.Quantum.Bitwise namespace](#) namespace provides several useful functions for manipulating integers through bitwise operators. For instance, [Parity function](#) returns the bitwise parity of an integer as another integer.

# Classical Mathematical Functions

3/5/2021 • 2 minutes to read • [Edit Online](#)

These functions are primarily used to work with the Q# built-in data types `Int`, `Double`, and `Range`.

The [Random operation](#) operation has signature `(Double[] => Int)`. It takes an array of doubles as input, and returns a randomly-selected index into the array as an `Int`. The probability of selecting a specific index is proportional to the value of the array element at that index. Array elements that are equal to zero are ignored and their indices are never returned. If any array element is less than zero, or if no array element is greater than zero, then the operation fails.

# Type Conversions

3/5/2021 • 2 minutes to read • [Edit Online](#)

Q# is a **strongly-typed** language. In particular, Q# does not implicitly cast between distinct types. For instance, `1 + 2.0` is not a valid Q# expression. Rather, Q# provides a variety of type conversion functions for constructing new values of a given type.

For example, [Length function](#) has an output type of `Int`, so its output must first be converted to a `Double` before it can be used as a part of a floating-point expression. This can be done using the [IntAsDouble function](#) function:

```
open Microsoft.Quantum.Convert as Convert;

function HalfLength<'T>(arr : 'T[]) : Double {
    return Convert.IntAsDouble(Length(arr)) / 2.0;
}
```

The [Microsoft.Quantum.Convert namespace](#) namespace provides common type conversion functions for working with the basic built-in types, such as `Int`, `Double`, `BigInt`, `Result`, and `Bool`:

```
let bool = Convert.ResultAsBool(One);           // true
let big = Convert.IntAsBigInt(271);             // 271L
let indices = Convert.RangeAsIntArray(0..4); // [0, 1, 2, 3, 4]
```

The [Microsoft.Quantum.Convert namespace](#) namespace also provides some more exotic conversions, such as [FunctionAsOperation](#), which converts functions `'T -> 'U` into new operations `'T => 'U`.

Finally, the Q# standard library provides a number of user-defined types such as [Complex user defined type](#) and [LittleEndian user defined type](#). Along with these types, the standard library provides functions such as [BigEndianAsLittleEndian function](#):

```
open Microsoft.Quantum.Arithmetic as Arithmetic;

let register = Arithmetic.BigEndian(qubits);
IncrementByInteger(Arithmetic.BigEndianAsLittleEndian(register));
```

# Higher-Order Control Flow

3/5/2021 • 9 minutes to read • [Edit Online](#)

One of the primary roles of the standard library is to make it easier to express high-level algorithmic ideas as [quantum programs](#). Thus, the Q# canon provides a variety of different flow control constructs, each implemented using partial application of functions and operations. Jumping immediately into an example, consider the case in which one wants to construct a "CNOT ladder" on a register:

```
let nQubits = Length(register);
CNOT(register[0], register[1]);
CNOT(register[1], register[2]);
// ...
CNOT(register[nQubits - 2], register[nQubits - 1]);
```

We can express this pattern by using iteration and `for` loops:

```
for idxQubit in 0..nQubits - 2 {
    CNOT(register[idxQubit], register[idxQubit + 1]);
}
```

Expressed in terms of [ApplyToEachCA operation](#) and array manipulation functions such as [Zipped function](#), however, this is much shorter and easier to read:

```
ApplyToEachCA(CNOT, Zip(register[0..nQubits - 2], register[1..nQubits - 1]));
```

In the rest of this section, we will provide a number of examples of how to use the various flow control operations and functions provided by the canon to compactly express quantum programs.

## Applying Operations and Functions over Arrays and Ranges

One of the primary abstractions provided by the canon is that of iteration. For instance, consider a unitary of the form  $\$U \otimes U \otimes \dots \otimes U\$$  for a single-qubit unitary  $\$U\$$ . In Q#, we might use [IndexRange function](#) to represent this as as a `for` loop over a register:

```
/// # Summary
/// Applies $H$ to all qubits in a register.
operation ApplyHadamardToAll(
    register : Qubit[])
: Unit is Adj + Ctl {
    for qubit in register {
        H(qubit);
    }
}
```

We can then use this new operation as `HAll(register)` to apply  $\$H \otimes H \otimes \dots \otimes H\$$ .

This is cumbersome to do, however, especially in a larger algorithm. Moreover, this approach is specialized to the particular operation that we wish to apply to each qubit. We can use the fact that operations are first-class to express this algorithmic concept more explicitly:

```
ApplyToEachCA(H, register);
```

Here, the suffix `CA` indicates that the call to `ApplyToEach` is itself adjointable and controllable. Thus, if `U` supports `Adjoint` and `Controlled`, the following lines are equivalent:

```
Adjoint ApplyToEachCA(U, register);
ApplyToEachCA(Adjoint U, register);
```

In particular, this means that calls to `ApplyToEachCA` can appear in operations for which an adjoint specialization is auto-generated. Similarly, [ApplyToEachIndex operation](#) is useful for representing patterns of the form `U(0, targets[0]); U(1, targets[1]); ...`, and offers versions for each combination of functors supported by its input.

#### TIP

`ApplyToEach` is type-parameterized such that it can be used with operations that take inputs other than `Qubit`. For example, suppose that `codeBlocks` is an array of `LogicalRegister` user defined type values that need to be recovered. Then `ApplyToEach(Recover(code, recoveryFn, _), codeBlocks)` will apply the error-correcting code `code` and recovery function `recoveryFn` to each block independently. This holds even for classical inputs:  
`ApplyToEach(R(_,_), qubit, [(PauliX, PI() / 2.0); (PauliY(), PI() / 3.0)])` will apply a rotation of  $\pi/2$  about `X` followed by a rotation of  $\pi/3$  about `Y`.

The Q# canon also provides support for classical enumeration patterns familiar to functional programming. For instance, [Fold function](#) implements the pattern `f(f(f(s_{\text{initial}}, x_0), x_1), \dots)` for reducing a function over a list. This pattern can be used to implement sums, products, minima, maxima and other such functions:

```
open Microsoft.Quantum.Arrays;
function Plus(a : Int, b : Int) : Int { return a + b; }
function Sum(xs : Int[]) {
    return Fold(Sum, 0, xs);
}
```

Similarly, functions like [Mapped function](#) and [MappedByIndex function](#) can be used to express functional programming concepts in Q#.

## Composing Operations and Functions

The control flow constructs offered by the canon take operations and functions as their inputs, such that it is helpful to be able to compose several operations or functions into a single callable. For instance, the pattern `UVU^{\dagger}` is extremely common in quantum programming, such that the canon provides the operation [ApplyWith operation](#) as an abstraction for this pattern. This abstraction also allows for more efficient compilation into circuits, as `Controlled` acting on the sequence `U(qubit); V(qubit); Adjoint U(qubit);` does not need to act on each `U`. To see this, let `c(U)` be the unitary representing `Controlled U([control], target)` and let `c(V)` be defined in the same way. Then for an arbitrary state `\ket{\psi}`, `\begin{align} c(U) c(V) c(U)^{\dagger} \ket{\psi} = \ket{\psi} \otimes (UVU^{\dagger}) \ket{\psi} \end{align}` and `\begin{align} (\boldsymbol{\text{boldone}} \otimes U) (c(V)) (\boldsymbol{\text{boldone}} \otimes U^{\dagger}) \ket{\psi} = \boldsymbol{\text{boldone}} \otimes (U c(V) U^{\dagger}) \ket{\psi} \end{align}` by the definition of `Controlled`. On the other hand, `\begin{align} c(U) c(V) c(U)^{\dagger} \ket{\psi} = \ket{\psi} \otimes (UU^{\dagger}) \ket{\psi} \end{align}` and `\begin{align} (\boldsymbol{\text{boldone}} \otimes U) (c(V)) (\boldsymbol{\text{boldone}} \otimes U^{\dagger}) \ket{\psi} = \boldsymbol{\text{boldone}} \otimes (U c(V) U^{\dagger}) \ket{\psi} \end{align}`. By linearity, we can conclude that we can factor `U` out in this way for all input states. That is, `\begin{align} c(UVU^{\dagger}) = U c(V) U^{\dagger} \end{align}`. Since controlling operations can be expensive in general, using controlled variants such as `WithC` and `WithCA` can help reduce the number of control functors

that need to be applied.

#### NOTE

One other consequence of factoring out `$U$` is that we need not even know how to apply the `Controlled` functor to `U`. `ApplyWithCA` therefore has a weaker signature than might be expected:

```
ApplyWithCA<'T> : (('T => Unit is Adj),  
                      ('T => Unit is Adj + Ctl), 'T) => Unit
```

Similarly, [Bound function](#) produces operations which apply a sequence of other operations in turn. For instance, the following are equivalent:

```
H(qubit); X(qubit);  
Bound([H, X], qubit);
```

Combining with iteration patterns can make this especially useful:

```
// Bracket the quantum Fourier transform with $XH$ on each qubit.  
ApplyWith(ApplyToEach(Bound([H, X]), _), QFT, _);
```

#### Time-Ordered Composition

We can go still further by thinking of flow control in terms of partial application and classical functions, and can model even fairly sophisticated quantum concepts in terms of classical flow control. This analogy is made precise by the recognition that unitary operators correspond exactly to the side effects of calling operations, such that any decomposition of unitary operators in terms of other unitary operators corresponds to constructing a particular calling sequence for classical subroutines which emit instructions to act as particular unitary operators. Under this view, `Bound` is precisely the representation of the matrix product, since `Bound([A, B])(target)` is equivalent to `A(target); B(target);`, which in turn is the calling sequence corresponding to `$BA$`.

A more sophisticated example is the [Trotter–Suzuki expansion](#). As discussed in the [Dynamical Generator Representation](#) section of [data structures](#), the Trotter–Suzuki expansion provides a particularly useful way of expressing matrix exponentials. For instance, applying the expansion at its lowest order yields that for any operators `$A$` and `$B$` such that `$A = A^\dagger$` and `$B = B^\dagger$`,

$$\exp(i [A + B] t) = \lim_{n \rightarrow \infty} \left( \exp(i A t / n) \exp(i B t / n) \right)^n.$$

Colloquially, this says that we can approximately evolve a state under `$A + B$` by alternately evolving under `$A$` and `$B$` alone. If we represent evolution under `$A$` by an operation

`A : (Double, Qubit[]) => Unit` that applies `$e^{i A t}$`, then the representation of the Trotter–Suzuki expansion in terms of rearranging calling sequences becomes clear. Concretely, given an operation

`U : ((Int, Double, Qubit[]) => Unit is Adj + Ctl)` such that `A = U(0, _, _)` and `B = U(1, _, _)`, we can define a new operation representing the integral of `U` at time `$t$` by generating sequences of the form

```
U(0, time / Float(nSteps), target);  
U(1, time / Float(nSteps), target);  
U(0, time / Float(nSteps), target);  
U(1, time / Float(nSteps), target);  
// ...
```

At this point, we can now reason about the Trotter–Suzuki expansion *without reference to quantum mechanics at all*. The expansion is effectively a very particular iteration pattern motivated by `\eqref{eq:trotter-suzuki-0}`. This iteration pattern is implemented by [DecomposedIntoTimeStepsCA function](#):

```
// The 2 indicates how many terms we need to decompose,
// while the 1 indicates that we are using the
// first-order Trotter-Suzuki decompositon.
DecomposeIntoTimeStepsCA((2, U), 1);
```

The signature of `DecomposeIntoTimeStepsCA` follows a common pattern in Q#, where collections that may be backed either by arrays or by something which compute elements on the fly are represented by tuples whose first elements are `Int` values indicating their lengths.

## Putting it Together: Controlling Operations

Finally, the canon builds on the `Controlled` functor by providing additional ways to condition quantum operations. It is common, especially in quantum arithmetic, to condition operations on computational basis states other than  $\lvert 0 \cdots 0 \rangle$ . Using the control operations and functions introduced above, we can more general quantum conditions in a single statement. Let's jump in to how `ControlledOnBitString` function does it (sans type parameters), then we'll break down the pieces one by one. The first thing we'll need to do is to define an operation which actually does the heavy lifting of implementing the control on arbitrary computational basis states. We won't call this operation directly, however, and so we add `_` to the beginning of the name to indicate that it's an implementation of another construct elsewhere.

```
operation _ControlledOnBitString(
    bits : Bool[],
    oracle: (Qubit[] => Unit is Adj + Ctl),
    controlRegister : Qubit[],
    targetRegister: Qubit[])
: Unit is Adj + Ctl
```

Note that we take a string of bits, represented as a `Bool` array, that we use to specify the conditioning that we want to apply to the operation `oracle` that we are given. Since this operation actually does the application directly, we also need to take the control and target registers, both represented here as `Qubit[]`.

Next, we note that controlling on the computational basis state  $\lvert \vec{s} \rangle = \lvert s_0 s_1 \dots s_{n-1} \rangle$  for a bit string  $\vec{s}$  can be realized by transforming  $\lvert \vec{s} \rangle$  into  $\lvert 0 \cdots 0 \rangle$ . In particular,  $\lvert \vec{s} \rangle = X^{s_0} \otimes X^{s_1} \otimes \dots \otimes X^{s_{n-1}} \lvert 0 \cdots 0 \rangle$ . Since  $X^{\dagger} = X$ , this implies that  $\lvert 0 \cdots 0 \rangle = X^{s_0} \otimes X^{s_1} \otimes \dots \otimes X^{s_{n-1}} \lvert \vec{s} \rangle$ . Thus, we can apply  $P = X^{s_0} \otimes X^{s_1} \otimes \dots \otimes X^{s_{n-1}}$ , apply `Controlled oracle`, and transform back to  $\lvert \vec{s} \rangle$ . This construction is precisely `ApplyWith`, so we write the body of our new operation accordingly:

```
{
    ApplyWithCA(
        ApplyPauliFromBitString(PauliX, false, bits, _),
        (Controlled oracle)(_, targetRegister),
        controlRegister
    );
}
```

Here, we have used `ApplyPauliFromBitString` operation to apply  $P$ , partially applying over its target for use with `ApplyWith`. Note, however, that we need to transform the *control* register to our desired form, so we partially apply the inner operation `(Controlled oracle)` on the *target*. This leaves `ApplyWith` acting to bracket the control register with  $P$ , exactly as we desired.

At this point, we could be done, but it is somehow unsatisfying that our new operation does not "feel" like applying the `Controlled` functor. Thus, we finish defining our new control flow concept by writing a function

that takes the oracle to be controlled and that returns a new operation. In this way, our new function looks and feels very much like `Controlled`, illustrating that we can easily define powerful new control flow constructs using Q# and the canon together:

```
function ControlledOnBitString(
    bits : Bool[],
    oracle: (Qubit[] => Unit is Adj + Ctl))
: ((Qubit[], Qubit[]) => Unit is Adj + Ctl) {
    return _ControlledOnBitString(bits, oracle, _,_);
}
```

# Data Structures and Modeling

3/5/2021 • 16 minutes to read • [Edit Online](#)

## Classical Data Structures

Along with user-defined types for representing quantum concepts, the canon also provides operations, functions, and types for working with classical data used in the control of quantum systems. For instance, the [Reversed function](#) function takes an array as input and returns the same array in reverse order. This can then be used on an array of type `Qubit[]` to avoid having to apply unnecessary `\operatorname{SWAP}` gates when converting between quantum representations of integers. Similarly, we saw in the previous section that types of the form `(Int, Int -> T)` can be useful for representing random access collections, so the [LookupFunction function](#) function provides a convenient way of constructing such types from array types.

### Pairs

The canon supports functional-style notation for pairs, complementing accessing tuples by deconstruction:

```
let pair = (PauliZ, register); // type (Pauli, Qubit[])
ApplyToEach(H, Snd(pair)); // No need to deconstruct to access the register.
```

### Arrays

The canon provides several functions for manipulating arrays. These functions are type-parameterized, and thus can be used with arrays of any Q# type. For instance, the [Reversed function](#) function returns a new array whose elements are in reverse order from its input. This can be used to change how a quantum register is represented when calling operations:

```
let leRegister = LittleEndian(register);
// QFT expects a BigEndian, so we can reverse before calling.
QFT(BigEndian(Reversed(leRegister!)));
// This is how the LittleEndianAsBigEndian function is implemented:
QFT(LittleEndianAsBigEndian(leRegister));
```

Similarly, the [Subarray function](#) function can be used to reorder or take subsets of the elements of an array:

```
// Applies H to qubits 2 and 5.
ApplyToEach(H, Subarray([2, 5], register));
```

When combined with flow control, array manipulation functions such as [Zipped function](#) can provide a powerful way to express quantum programs:

```
// Applies X, Y, Z, to a register of any size.
ApplyToEach(
    ApplyPauli(_, register),
    Map(
        EmbedPauli(_, _, Length(register)),
        Zipped([PauliX, PauliY, PauliZ], [3, 1, 7])
    )
);
```

## Oracles

In the [phase estimation](#) and [amplitude amplification](#) literature the concept of an oracle appears frequently. Here the term oracle refers to a quantum subroutine that acts upon a set of qubits and returns the answer as a phase. This subroutine often can be thought of as an input to a quantum algorithm that accepts the oracle, in addition to some other parameters, and applies a series of quantum operations and treating a call to this quantum subroutine as if it were a fundamental gate. Obviously, in order to actually implement the larger algorithm a concrete decomposition of the oracle into fundamental gates must be provided but such a decomposition is not needed in order to understand the algorithm that calls the oracle. In Q#, this abstraction is represented by using that operations are first-class values, such that operations can be passed to implementations of quantum algorithms in a black-box manner. Moreover, user-defined types are used to label the different oracle representations in a type-safe way, making it difficult to accidentally conflate different kinds of black box operations.

Such oracles appear in a number of different contexts, including famous examples such as [Grover's search](#) and quantum simulation algorithms. Here we focus on the oracles needed for just two applications: amplitude amplification and phase estimation. We will first discuss amplitude amplification oracles before proceeding to phase estimation.

### Amplitude Amplification Oracles

The amplitude amplification algorithm aims to perform a rotation between an initial state and a final state by applying a sequence of reflections of the state. In order for the algorithm to function, it needs a specification of both of these states. These specifications are given by two oracles. These oracles work by breaking the inputs into two spaces, a "target" subspace and an "initial" subspace. The oracles identify such subspaces, similar to how Pauli operators identify two spaces, by applying a  $\pm 1$  phase to these spaces. The main difference is that these spaces need not be half-spaces in this application. Also note that these two subspaces are not usually mutually exclusive: there will be vectors that are members of both spaces. If this were not true then amplitude amplification would have no effect so we need the initial subspace to have non-zero overlap with the target subspace.

We will denote the first oracle that we need for amplitude amplification to be  $P_0$ , defined to have the following action. For all states  $|x\rangle$  in the "initial" subspace  $P_0|x\rangle = -|x\rangle$  and for all states  $|y\rangle$  that are not in this subspace we have  $P_0|y\rangle = |y\rangle$ . The oracle that marks the target subspace,  $P_1$ , takes exactly the same form. For all states  $|x\rangle$  in the target subspace (that is, for all states that you'd like the algorithm to output),  $P_1|x\rangle = -|x\rangle$ . Similarly, for all states  $|y\rangle$  that are not in the target subspace  $P_1|y\rangle = |y\rangle$ . These two reflections are then combined to form an operator that enacts a single step of amplitude amplification,  $Q = -P_0 P_1$ , where the overall minus sign is only important to consider in controlled applications. Amplitude amplification then proceeds by taking an initial state,  $|\psi\rangle$  that is in the initial subspace and then performs  $|\psi\rangle \mapsto Q^m |\psi\rangle$ . Performing such an iteration guarantees that if one starts with an initial state that has overlap  $|\sin^2(\theta)|$  with the marked space then after  $m$  iterations this overlap becomes  $|\sin^{2m+1}(\theta)|$ . We therefore typically wish to choose  $m$  to be a free parameter such that  $[2m+1]\theta = \pi/2$ ; however, such rigid choices are not as important for some forms of amplitude amplification such as fixed point amplitude amplification. This process allows us to prepare a state in the marked subspace using quadratically fewer queries to the marking function and the state preparation function than would be possible on a strictly classical device. This is why amplitude amplification is a significant building block for many applications of quantum computing.

In order to understand how to use the algorithm, it is useful to provide an example that gives a construction of the oracles. Consider performing Grover's algorithm for database searches in this setting. In Grover's search the goal is to transform the state  $|+\rangle^{\otimes n} = H^{\otimes n}|0\rangle$  into one of (potentially) many marked states. To further simplify, let's just look at the case where the only marked state is  $|0\rangle$ . Then we have design two oracles: one that only marks the initial state  $|+\rangle^{\otimes n}$  with a minus sign and another that marks the marked state  $|0\rangle$  with a minus sign. The latter gate can be implemented using the following process operation, by using the control flow operations in the canon:

```

operation ReflectAboutAllZeros(register : Qubit[]) : Unit
is Adj + Ctl {

    // Apply $X$ gates to every qubit.
    ApplyToEach(X, register);

    // Apply an $n-1$ controlled $Z$-gate to the $n^{\text{th}}$ qubit.
    // This gate will lead to a sign flip if and only if every qubit is
    // $1$, which happens only if each of the qubits were $0$ before step 1.
    Controlled Z(Most(register), Tail(register));

    // Apply $X$ gates to every qubit.
    ApplyToEach(X, register);
}

```

This oracle is then a special case of the [RA111 operation](#) operation, which allows for rotating by an arbitrary phase instead of the reflection case  $\phi = \pi$ . In this case, [RA111](#) is similar to the [R1 operation](#) prelude operation, in that it rotates about  $|\psi\rangle$  instead of the single-qubit state  $|1\rangle$ .

The oracle that marks the initial subspace can be constructed similarly. In pseudocode:

1. Apply  $H$  gates to every qubit.
2. Apply  $X$  gates to every qubit.
3. Apply an  $n-1$  controlled  $Z$ -gate to the  $n^{\text{th}}$  qubit.
4. Apply  $X$  gates to every qubit.
5. Apply  $H$  gates to every qubit.

This time, we also demonstrate using [ApplyWith operation](#) together with the [RA111 operation](#) operation discussed above:

```

operation ReflectAboutInitial(register : Qubit[]) : Unit
is Adj + Ctl {
    ApplyWithCA(ApplyToEach(H, _), ApplyWith(ApplyToEach(X, _), RA111(_, PI()), _), register);
}

```

We can then combine these two oracles together to rotate between the two states and deterministically transform  $|\psi\rangle$  to  $|0\rangle$  using a number of layers of Hadamard gates that is proportional to  $\sqrt{2^n}$  (ie  $m \propto \sqrt{2^n}$ ) versus the roughly  $2^n$  layers that would be needed to non-deterministically prepare the  $|0\rangle$  state by preparing and measuring the initial state until the outcome  $0$  is observed.

## Phase Estimation Oracles

For phase estimation the oracles are somewhat more natural. The aim in phase estimation is to design a subroutine that is capable of sampling from the eigenvalues of a unitary matrix. This method is indispensable in quantum simulation because for many physical problems in chemistry and material science these eigenvalues give the ground-state energies of quantum systems which provides us valuable information about the phase diagrams of materials and reaction dynamics for molecules. Every flavor of phase estimation needs an input unitary. This unitary is customarily described by one of two types of oracles.

### TIP

Both of the oracle types described below are covered in the samples. To learn more about continuous query oracles, please see the [PhaseEstimation sample](#). To learn more about discrete query oracles, please see the [IsingPhaseEstimation sample](#).

The first type of oracle, which we call a discrete query oracle and represent with the user-defined type [DiscreteOracle user defined type](#), simply involves a unitary matrix. If  $\$U\$$  is the unitary whose eigenvalues we wish to estimate then the oracle for  $\$U\$$  is simply a stand-in for a subroutine that implements  $\$U\$$ . For example, one could take  $\$U\$$  to be the oracle  $\$Q\$$  defined above for amplitude estimation. The eigenvalues of this matrix can be used to estimate the overlap between the initial and target states,  $\$|\sin^2(\theta)|$ , using quadratically fewer samples than one would need otherwise. This earns the application of phase estimation using the Grover oracle  $\$Q\$$  as input the moniker of amplitude estimation. Another common application, widely used in quantum metrology, involves estimating a small rotation angle. In other words, we wish to estimate  $\$|\theta|$  for an unknown rotation gate of the form  $\$R_z(\theta)\$$ . In such cases, the subroutine that we would interact with in order to learn this fixed value of  $\$|\theta|$  for the gate is  $\$ \$ \begin{aligned} U &= R_z(\theta) \\ &= \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}. \end{aligned} \$ \$$

The second type of oracle used in phase estimation is the continuous query oracle, represented by the [ContinuousOracle user defined type](#) type. A continuous query oracle for phase estimation takes the form  $\$U(t)\$$  where  $\$t\$$  is a classically known real number. If we let  $\$U\$$  be a fixed unitary then the continuous query oracle takes the form  $\$U(t) = U^t\$$ . This allows us to query matrices such as  $\$|\sqrt{U}|\$$ , which could not be implemented directly in the discrete query model.

This type of oracle is valuable when you're not probing a particular unitary, but rather wish to learn the properties of the generator of the unitary. For example, in dynamical quantum simulation the goal is to devise quantum circuits that closely approximate  $\$U(t)=e^{-iHt}\$$  for a Hermitian matrix  $\$H\$$  and evolution time  $\$t\$$ . The eigenvalues of  $\$U(t)\$$  are directly related to the eigenvalues of  $\$H\$$ . To see this, consider an eigenvector of  $\$H\$:  $\$H|\psi\rangle = E|\psi\rangle$  then it is easy to see from the power-series definition of the matrix exponential that  $\$U(t)|\psi\rangle = e^{-iEt}|\psi\rangle$ . Thus estimating the eigenphase of  $\$U(t)\$$  gives the eigenvalue  $\$E\$$  assuming the eigenvector  $\$|\psi\rangle\$$  is input into the phase estimation algorithm. However, in this case the value  $\$t\$$  can be chosen at the user's discretion since for any sufficiently small value of  $\$t\$$  the eigenvalue  $\$E\$$  can be uniquely inverted through  $\$E=-\phi/t\$$ . Since quantum simulation methods provide the ability to perform a fractional evolution, this grants phase estimation algorithms an additional freedom when querying the unitary, specifically while the discrete query model allows only unitaries of the form  $\$U^j\$$  to be applied for integer  $\$j\$$  the continuous query oracle allows us to approximate unitaries of the form  $\$U^t\$$  for any real valued  $\$t\$$ . This is important to squeeze every last ounce of efficiency out of phase estimation algorithms because it allows us to choose precisely the experiment that would provide the most information about  $\$E\$$ ; whereas methods based on discrete queries must make do with compromising by choosing the best integer number of queries in the algorithm.$

As a concrete example of this, consider the problem of estimating not the rotation angle of a gate but the procession frequency of a rotating quantum system. The unitary that describes such quantum dynamics is  $\$U(t)=R_z(2\omega t)\$$  for evolution time  $\$t\$$  and unknown frequency  $\$|\omega|\$$ . In this context, we can simulate  $\$U(t)\$$  for any  $\$t\$$  using a single  $\$R_z\$$  gate and as such do not need to restrict ourselves to only discrete queries to the unitary. Such a continuous model also has the property that frequencies greater than  $\$2\pi\$$  can be learned from phase estimation processes that use continuous queries because phase information that would otherwise be masked by the branch-cuts of the logarithm function can be revealed from the results of experiments performed on non-commensurate values of  $\$t\$$ . Thus for problems such as this continuous query models for the phase estimation oracle are not only appropriate but are also preferable to the discrete query model. For this reason Q# has functionality for both forms of queries and leave it to the user to decide upon a phase estimation algorithm to fit their needs and the type of oracle that is available.

## Dynamical Generator Modeling

Generators of time-evolution describe how states evolve through time. For instance, the dynamics of a quantum state  $\$|\psi\rangle\$$  is governed by the Schrödinger equation  $\$ i\frac{d}{dt}|\psi(t)\rangle = H|\psi(t)\rangle \$$  with a Hermitian matrix  $\$H\$$ , known as the Hamiltonian, as the generator of motion. Given an initial state  $\$|\psi(0)\rangle\$$  at time  $\$t=0\$$ , the formal solution to this equation at time  $\$t\$$  may be, in

principle, written  $\begin{aligned} \ket{\psi(t)} = U(t)\ket{\psi(0)}, \end{aligned}$  where the matrix exponential  $U(t)=e^{-i H t}$  is known as the unitary time-evolution operator. Though we focus on generators of this form in the following, we emphasize that the concept applies more broadly, such as to the simulation of open quantum systems, or to more abstract differential equations.

A primary goal of dynamical simulation is to implement the time-evolution operator on some quantum state encoded in qubits of a quantum computer. In many cases, the Hamiltonian may be broken into a sum of some simpler terms

$$\begin{aligned} H &= \sum^{d-1}_{j=0} H_j, \end{aligned}$$

where time-evolution by each term alone is easy to implement on a quantum computer. For instance, if  $H_j$  is a Pauli  $X_1X_2$  operator acting on the 1st and 2nd elements of the qubit register `qubits`, time-evolution by it for any time  $t$  may be implemented simply by calling the operation `Exp([PauliX,PauliX], t, qubits[1..2])`, which has signature `((Pauli[], Double, Qubit[]) => Unit is Adj + ct1)`. As discussed later in Hamiltonian Simulation, one solution then is to approximate time-evolution by  $H$  with a sequence of simpler operations

$$\begin{aligned} U(t) &= \left( e^{-iH_0 t / r} e^{-iH_1 t / r} \cdots e^{-iH_{d-1} t / r} \right)^r + \mathcal{O}(d^2 \max_j |H_j|^2 t^{2/r}), \end{aligned}$$

where the integer  $r > 0$  controls the approximation error.

The dynamical generator modeling library provides a framework for systematically encoding complicated generators in terms of simpler generators. Such a description may then be passed to, say, the simulation library to implement time-evolution by a simulation algorithm of choice, with many details automatically taken care of.

#### TIP

The dynamical generator library described below is covered in the samples. For an example based on the Ising model, please see the [IsingGenerators](#) sample. For an example based on molecular Hydrogen, please see the [H2SimulationCmdLine](#) and [H2SimulationGUI](#) samples.

## Complete Description of a Generator

At the top level, a complete description of a Hamiltonian is contained in the `EvolutionGenerator` user-defined type which has two components.:

```
newtype EvolutionGenerator = (EvolutionSet, GeneratorSystem);
```

The `GeneratorSystem` user-defined type is a classical description of the Hamiltonian.

```
newtype GeneratorSystem = (Int, (Int -> GeneratorIndex));
```

The first element `Int` of the tuple stores the number of terms  $d$  in the Hamiltonian, and the second element `(Int -> GeneratorIndex)` is a function that maps an integer index in  $\{0, 1, \dots, d-1\}$  to a `GeneratorIndex` user-defined type which uniquely identifies each primitive term in the Hamiltonian. Note that by expressing the collection of terms in the Hamiltonian as a function rather than as an array `GeneratorIndex[]`, this allows for on-the-fly computation of the `GeneratorIndex` which is especially useful when describing Hamiltonians with a large number of terms.

Crucially, we do not impose a convention on what primitive terms identified by the `GeneratorIndex` are easy-to-simulate. For instance, primitive terms could be Pauli operators as discussed above, but they could also be Fermionic annihilation and creation operators commonly used in quantum chemistry simulation. By itself, a `GeneratorIndex` is meaningless as it does not describe how time-evolution by the term it points to may be implemented as a quantum circuit.

This is resolved by specifying an `EvolutionSet` user-defined type that maps any `GeneratorIndex`, drawn from some canonical set, to a unitary operator, the `EvolutionUnitary`, expressed as a quantum circuit. The `EvolutionSet` defines the convention of how a `GeneratorIndex` is structured, and also defines the set of possible `GeneratorIndex`.

```
newtype EvolutionSet = (GeneratorIndex -> EvolutionUnitary);
```

## Pauli Operator Generators

A concrete and useful example of generators are Hamiltonians that are a sum of Pauli operators, each possibly with a different coefficient.  $\hat{H} = \sum_{j=0}^{d-1} a_j \hat{H}_j$  where each  $\hat{H}_j$  is now drawn from the Pauli group. For such systems, we provide the `PauliEvolutionSet()` of type `EvolutionSet` that defines a convention for how an element of the Pauli group and a coefficient may be identified by a `GeneratorIndex`, which has the following signature.

```
newtype GeneratorIndex = ((Int[], Double[]), Int[]);
```

In our encoding, the first parameter `Int[]` specifies a Pauli string, where  $\hat{X} \rightarrow 0$ ,  $\hat{Y} \rightarrow 1$ ,  $\hat{Z} \rightarrow 2$ . The second parameter `Double[]` stores the coefficient of the Pauli string in the Hamiltonian. Note that only the first element of this array is used. The third parameter `Int[]` indexes the qubits that this Pauli string acts on, and must have no duplicate elements. Thus the Hamiltonian term  $0.4 \hat{X}_0 \hat{Y}_8 \hat{Z}_1$  may be represented as

```
let generatorIndexExample = GeneratorIndex(([1,2,0,3], [0.4]), [0,8,2,1]);
```

The `PauliEvolutionSet()` is a function that maps any `GeneratorIndex` of this form to an `EvolutionUnitary` with the following signature.

```
newtype EvolutionUnitary = ((Double, Qubit[]) => Unit) is Adj + Ctl;
```

The first parameter represents a time-duration, that will be multiplied by the coefficient in the `GeneratorIndex`, of unitary evolution. The second parameter is the qubit register the unitary acts on.

## Time-Dependent Generators

In many cases, we are also interested in modelling time-dependent generators, as might occur in the Schrödinger equation  $i\frac{d|\psi(t)\rangle}{dt} = \hat{H}(t)|\psi(t)\rangle$  where the generator  $\hat{H}(t)$  is now time-dependent. The extension from the time-independent generators above to this case is straightforward. Rather than having a fixed `GeneratorSystem` describing the Hamiltonian for all times  $t$ , we instead have the `GeneratorSystemTimeDependent` user-defined type.

```
newtype GeneratorSystemTimeDependent = (Double -> GeneratorSystem);
```

The first parameter is a continuous schedule parameter  $s \in [0,1]$ , and functions of this type return a `GeneratorSystem` for that schedule. Note that the schedule parameter may be linearly related to the physical time parameter, for example,  $s = t / T$ , for some total time of simulation  $T$ . In general however, this need not be the case.

Similarly, a complete description of this generator requires an `EvolutionSet`, and so we define an `EvolutionSchedule` user-defined type.

```
newtype EvolutionSchedule = (EvolutionSet, GeneratorSystemTimeDependent);
```

# Quantum Algorithms

5/27/2021 • 14 minutes to read • [Edit Online](#)

## Amplitude Amplification

*Amplitude Amplification* is one of the fundamental tools of Quantum Computing. It is the fundamental idea that underlies Grover's search, amplitude estimation and many quantum machine learning algorithms. There are many variants, and in Q# we provide a general version based on Oblivious Amplitude Amplification with Partial Reflections to allow for the widest area of application.

The central idea behind amplitude amplification is to amplify the probability of a desired outcome occurring by performing a sequence of reflections. These reflections rotate the initial state closer towards a desired target state, often called a marked state. Specifically, if the probability of measuring the initial state to be in a marked state is  $\sin^2(\theta)$  then after applying amplitude amplification  $m$  times the probability of success becomes  $\sin^2((2m+1)\theta)$ . This means that if  $\theta = \pi/[2(2n+1)]$  for some value of  $n$  then amplitude amplification is capable of boosting the probability of success to  $100\%$  after  $n$  iterations of amplitude amplification. Since  $\theta = \sin^{-1}(\sqrt{\Pr(\text{success})})$  this means that the number of iterations needed to obtain a success deterministically is quadratically lower than the expected number needed to find a marked state non-deterministically using random sampling.

Each iteration of Amplitude amplification requires that two reflection operators be specified. Specifically, if  $Q$  is the amplitude amplification iterate and  $P_0$  is a projector operator onto the initial subspace and  $P_1$  is the projector onto the marked subspace then  $Q = -(P_0)(P_1)$ . Recall that a projector is a Hermitian operator that has eigenvalues  $+1$  and  $0$  and as a result  $(P_0)$  is unitary because it has eigenvalues that are roots of unity (in this case  $\pm 1$ ). As an example, consider the case of Grover's search with initial state  $H^{\otimes n} |0\rangle$  and marked state  $|m\rangle$ ,  $P_0 = H^{\otimes n}|0\rangle\langle 0|H^{\otimes n}$  and  $P_1 = |m\rangle\langle m|$ . In most applications of amplitude amplification  $P_0$  will be a projector onto an initial state meaning that  $P_0 = \frac{1}{2}(|\psi\rangle\langle\psi| - I)$  for some vector  $|\psi\rangle$ ; however, for oblivious amplitude amplification  $P_0$  will typically project onto many quantum states (for example, the multiplicity of the  $+1$  eigenvalue of  $P_0$  is greater than  $1$ ).

The logic behind amplitude amplification follows directly from the eigen-decomposition of  $Q$ . Specifically, the eigenvectors of  $Q$  that the initial state has non-zero support over can be shown to be linear combinations of the  $+1$  eigenvectors of  $P_0$  and  $P_1$ . Specifically, the initial state for amplitude amplification (assuming it is a  $+1$  eigenvector of  $P_0$ ) can be written as  $|\psi\rangle = \frac{1}{\sqrt{2}}(|\psi_+\rangle + e^{i\theta}|\psi_-\rangle)$ , where  $|\psi_{\pm}\rangle$  are eigenvectors of  $Q$  with eigenvalues  $e^{\pm i\theta}$  and only have support on the  $+1$  eigenvectors of  $P_0$  and  $P_1$ . The fact that the eigenvalues are  $e^{\pm i\theta}$  implies that the operator  $Q$  performs a rotation in a two-dimensional subspace specified by the two projectors and the initial state where the rotation angle is  $2\theta$ . This is why after  $m$  iterations of  $Q$  the success probability is  $\sin^2((2m+1)\theta)$ .

Another useful property that comes out of this is that the eigenvalue  $\theta$  is directly related to probability that the initial state would be marked (in the case where  $P_0$  is a projector onto only the initial state). Since the eigenphases of  $Q$  are  $2\theta = 2\sin^{-1}(\sqrt{\Pr(\text{success})})$  it then follows that if we apply phase estimation to  $Q$  then we can learn the probability of success for a unitary quantum procedure. This is useful because it requires quadratically fewer applications of the quantum procedure to learn the success probability than would otherwise be needed.

Q# introduces amplitude amplification as a specialization of oblivious amplitude amplification. Oblivious amplitude amplification earns this moniker because the projector onto the initial eigenspace need not be a projector onto the initial state. In this sense, the protocol is oblivious to the initial state. The key application of

oblivious amplitude amplification is in certain *linear combinations of unitary* Hamiltonian simulation methods, wherein the initial state is unknown but becomes entangled with an auxiliary register in the simulation protocol. If this auxiliary register were to be measured to be a fixed value, say  $\$0\$$ , then such simulation methods apply the desired unitary transformation to the remaining qubits (called the system register). All other measurement outcomes lead to failure however. Oblivious amplitude amplification allows the probability of success of this measurement to be boosted to  $\$100\%\$$  using the above reasoning. Further, ordinary amplitude amplification corresponds to the case where the system register is empty. This is why Q# uses oblivious amplitude amplification as its fundamental amplitude amplification subroutine.

The general routine (`AmpAmpObliviousByReflectionPhases`) has two registers that we call `ancillaRegister` and `systemRegister`. It also accepts two oracles for the necessary reflections. The `ReflectionOracle` acts only on the `ancillaRegister` while the `ObliviousOracle` acts jointly on both registers. The input to `ancillaRegister` must be initialized to a -1 eigenstate of the first reflection operator  $\$|boldone -2P_1\$$ .

Typically, the oracle prepares the state in the computational basis  $\$|ket{0...0}\$$ . In our implementation, the `ancillaRegister` consists of one qubit (`flagQubit`) that controls the `stateOracle` and the rest of the desired auxiliary qubits. The `stateOracle` is applied when the `flagQubit` is  $\$|ket{1}\$$ .

One may also provide oracles `StateOracle` and `ObliviousOracle` instead of reflections via a call to `AmpAmpObliviousByOraclePhases`.

As mentioned, traditional Amplitude Amplification is just a special case of these routines where `obliviousOracle` is the identity operator and there are no system qubits (for example, `systemRegister` is empty). If you wish to obtain phases for partial reflections (for example, for Grover search), the function `AmpAmpPhasesStandard` is available. Please refer to `DatabaseSearch.qs` for a sample implementation of Grover's algorithm.

We relate the single-qubit rotation phases to the reflection operator phases as described in the paper by [G.H. Low, I. L. Chuang](#). The fixed point phases that are used are detailed in [Yoder, Low and Chuang](#) along with the phases in [Low, Yoder and Chuang](#).

For background, you could start from [Standard Amplitude Amplification](#) then move to an introduction to [Oblivious Amplitude Amplification](#) and finally generalizations presented in [Low and Chuang](#). A nice overview presentation of this entire area (as it relates to Hamiltonian Simulation) was given by [Dominic Berry](#).

## Quantum Fourier Transform

The Fourier transform is a fundamental tool of classical analysis and is just as important for quantum computations. In addition, the efficiency of the *quantum Fourier transform* (QFT) far surpasses what is possible on a classical machine making it one of the first tools of choice when designing a quantum algorithm.

As an approximate generalization of the QFT, we provide the `ApproximateQFT` operation operation that allows for further optimizations by pruning rotations that aren't strictly necessary for the desired algorithmic accuracy. The approximate QFT requires the dyadic  $\$Z\$$ -rotation operation `RFrac` operation as well as the `H` operation operation. The input and output are assumed to be encoded in big endian encoding---that is, the qubit with index `0` is encoded in the left-most (highest) bit of the binary integer representation. This aligns with `ket notation`, as a register of three qubits in the state  $\$|ket{100}\$$  corresponds to  $\$q_0\$$  being in the state  $\$|ket{1}\$$  while  $\$q_1\$$  and  $\$q_2\$$  are both in state  $\$|ket{0}\$$ . The approximation parameter  $\$a\$$  determines the pruning level of the  $\$Z\$$ -rotations, for example,  $\$a \in [0..n]\$$ . In this case all  $\$Z\$$ -rotations  $\$2\pi/2^k\$$  where  $\$k > a\$$  are removed from the QFT circuit. It is known that for  $\$k \geq \log_2(n) + \log_2(1/\epsilon) + 3\$$ . one can bound  $\$|\operatorname{operatorname}{QFT} - \operatorname{operatorname}{AQFT}| < \epsilon\$$ . Here  $\$|\cdot|\$$  is the operator norm which in this case is the square root of the largest `eigenvalue` of  $\$(\operatorname{operatorname}{QFT} - \operatorname{operatorname}{AQFT})^\dagger(\operatorname{operatorname}{QFT} - \operatorname{operatorname}{AQFT})\$$ .

## Arithmetic

Just as arithmetic plays a central role in classical computing, it is also indispensable in quantum computing. Algorithms such as Shor's factoring algorithm, quantum simulation methods as well as many oracular algorithms rely upon coherent arithmetic operations. Most approaches to arithmetic build upon quantum adder circuits. The simplest adder takes a classical input  $b$  and adds the value to a quantum state holding an integer  $|a\rangle$ . Mathematically, the adder (which we denote  $\text{Add}(b)$  for classical input  $b$ ) has the property that

$\text{Add}(b)|a\rangle = |a + b\rangle$ . This basic adder circuit is more of an incrementer than an adder. It can be converted into an adder that has two quantum inputs via

$$\begin{aligned} \text{Add}(a)\text{Add}(b)|a\rangle|b\rangle &= |\Lambda_a\rangle|\Lambda_{a+b}\rangle \\ &= |\Lambda_a\rangle|\Lambda_b\rangle|\Lambda_{a+b}\rangle \\ &\vdots \\ &= |\Lambda_{a_{n-1}}\rangle|\Lambda_{a_{n-1}+b}\rangle \\ &= |\Lambda_a\rangle|\Lambda_b\rangle|\Lambda_{a+b}\rangle \end{aligned}$$

for  $n$ -bit integers  $a$  and  $b$  and addition modulo  $2^n$ . Recall that the notation  $|\Lambda_x(A)\rangle$  refers, for any operation  $A$ , to the controlled version of that operation with the qubit  $x$  as control.

Similarly, classically controlled multiplication (a modular form of which is essential for Shor's factoring algorithm) can be performed by using a similar series of controlled additions:

$$\begin{aligned} \text{Mult}(a)|x\rangle|b\rangle &= |\Lambda_{x_0}\rangle|\Lambda_{(2^0 a)}\rangle \\ &= |\Lambda_{x_1}\rangle|\Lambda_{(2^1 a)}\rangle|\Lambda_{(2^2 a)}\rangle \\ &\vdots \\ &= |\Lambda_{x_{n-1}}\rangle|\Lambda_{(2^{n-1} a)}\rangle|\Lambda_{(2^n a)}\rangle \\ &= |\Lambda_x\rangle|\Lambda_{b+ax}\rangle \end{aligned}$$

There is a subtlety with multiplication on quantum computers that you may notice from the definition of  $\text{Mult}$  above. Unlike addition, the quantum version of this circuit stores the product of the inputs in an auxiliary register rather than in the input register. In this example, the register is initialized with the value  $b$ , but typically it will start holding the value zero. This is needed because in general there is not a multiplicative inverse for general  $a$  and  $x$ . Since all quantum operations, save measurement, are reversible we need to keep enough information around to invert the multiplication. For this reason the result is stored in a separate array. This trick of saving the output of an irreversible operation, like multiplication, in a separate register is known as the "Bennett trick" after Charlie Bennett and is a fundamental tool in both reversible and quantum computing.

Many quantum circuits have been proposed for addition and each explores a different tradeoff in terms of the number of qubits (space) and the number of gate operations (time) required. We review two highly space efficient adders below known as the Draper adder and the Beauregard adder.

### Draper Adder

The Draper adder is arguably one of the most elegant quantum adders, as it directly invokes quantum properties to perform addition. The insight behind the Draper adder is that the Fourier transform can be used to translate phase shifts into a bit shift. It then follows that by applying a Fourier transform, applying appropriate phase shifts, and then undoing the Fourier transform you can implement an adder. Unlike many other adders that have been proposed, the Draper adder explicitly uses quantum effects introduced through the quantum Fourier transform. It does not have a natural classical counterpart. The specific steps of the Draper adder are given below.

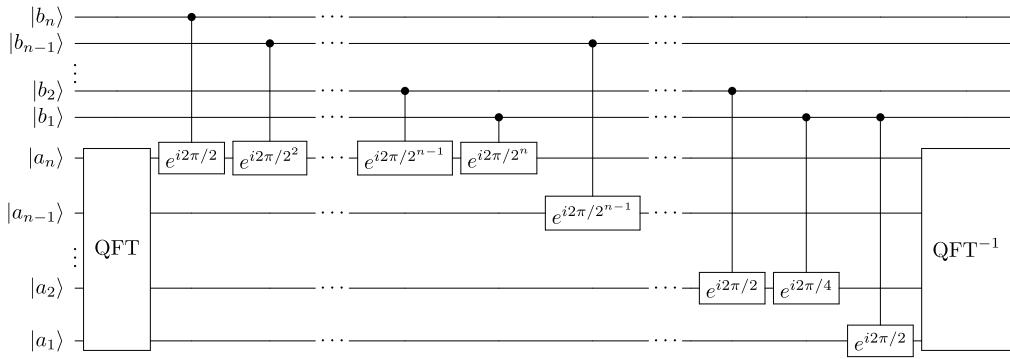
Assume that you have two  $n$ -bit qubit registers storing the integers  $a$  and  $b$  then for all  $a$   $b$

$$\begin{aligned} \text{QFT}|a\rangle &= \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} e^{i2\pi(j/a)/2^n} |j\rangle \\ \text{QFT}^{-1}|b\rangle &= \frac{1}{\sqrt{2^n}} \left( |0\rangle + e^{i2\pi(b/2^n)} |1\rangle \right) \end{aligned}$$

If we define  $|\phi_k(a)\rangle = \frac{1}{\sqrt{2^n}} \left( |\Lambda_a\rangle + e^{i2\pi(a/2^n)} |\Lambda_{a+k}\rangle \right)$  then after some algebra you can see that  $\text{QFT}^{-1}\text{QFT}|a\rangle = |\phi_1(a)\rangle \otimes \dots \otimes |\phi_n(a)\rangle$ . The path towards performing an adder then becomes clear after observing that the sum of the inputs can be written as  $|\Lambda_{a+b}\rangle = \text{QFT}^{-1}|\phi_1(a+b)\rangle \otimes \dots \otimes |\phi_n(a+b)\rangle$ . The integers  $b$  and  $a$  can then be added by performing controlled-phase rotation on each of the qubits in the decomposition using the bits of  $b$  as controls.

This expansion can be further simplified by noting that for any integer  $j$  and real number  $x$ ,

$e^{i2\pi(x+j)} = e^{i2\pi x}$ . This is because if you rotate  $360^\circ$  degrees ( $2\pi$  radians) in a circle then you end up precisely where you started. The only important part of  $x$  for  $e^{i2\pi x}$  is therefore the fractional part of  $x$ . Specifically, if we have a binary expansion of the form  $x = y + 0.x_0x_1\dots x_n$  then  $e^{i2\pi x} = e^{i2\pi (0.x_0x_1\dots x_{n-1})}$  and hence  $\langle \phi_k | \phi_{k+a+b} \rangle = \frac{1}{\sqrt{2}} (\langle \phi_0 | + e^{i2\pi [a/2^k + b/2^k]}) \langle \phi_1 | \dots \langle \phi_n |$ . This means that if we perform addition by incrementing each of the tensor factors in the expansion of the Fourier transform of  $\langle \phi_a |$  then the number of rotations shrinks as  $k$  decreases. This substantially reduces the number of quantum gates needed in the adder. We denote the Fourier transform, phase addition and the inverse Fourier transform steps that comprise the Draper adder as  $\text{QFT}^{-1}(\phi) \text{ADD}(\phi) \text{QFT}$ . A quantum circuit that uses this simplification to implement the entire process can be seen below.



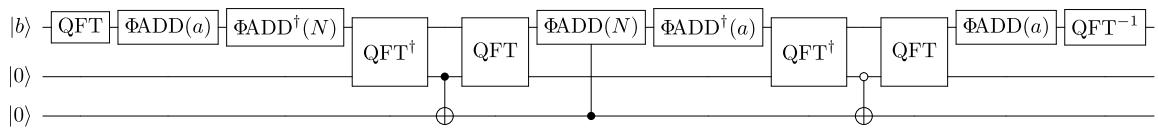
Each controlled  $e^{i2\pi/k}$  gate in the circuit refers to a controlled-phase gate. Such gates have the property that on the pair of qubits on which they act,  $\langle \phi_0 | \phi_0 \rangle \mapsto \langle \phi_0 | \phi_0 \rangle$  but  $\langle \phi_1 | \phi_1 \rangle \mapsto e^{i2\pi/k} \langle \phi_1 | \phi_1 \rangle$ . This circuit allows us to perform addition using no additional qubits apart from those needed to store the inputs and the outputs.

### Beauregard Adder

The Beauregard adder is a quantum modular adder that uses the Draper adder in order to perform addition modulo  $N$  for an arbitrary value positive integer  $N$ . The significance of quantum modular adders, such as the Beauregard adder, stems to a large extent from their use in the modular exponentiation step within Shor's algorithm for factoring. A quantum modular adder has the following action for quantum input  $\langle \phi_b |$  and classical input  $a$  where  $a$  and  $b$  are promised to be integers mod  $N$ , meaning that they are in the interval  $[0, N-1]$ .

$$\langle \phi_b | \rightarrow \langle \phi_{b+a} | \text{mod } N = \begin{cases} \langle \phi_{b+a} | & b+a < N \\ \langle \phi_{b+a-N} | & (b+a) \geq N \end{cases}$$

The Beauregard adder uses the Draper adder, or more specifically  $\text{ADD}(\phi)$ , to add  $a$  and  $b$  in phase. It then uses the same operation to identify whether  $a+b < N$  by subtracting  $N$  and testing if  $a+b-N < 0$ . The circuit stores this information in an auxiliary qubit and then adds  $N$  back the register if  $a+b < N$ . It then concludes by uncomputing this auxiliary qubit (this step is needed to ensure that the auxiliary qubit can be de-allocated after calling the adder). The circuit for the Beauregard adder is given below.



Here the gate  $\Phi \text{ADD}(a)$  takes the same form as  $\Phi \text{ADD}(\phi)$  except that in this context the input is classical rather than quantum. This allows the controlled phases in  $\Phi \text{ADD}(a)$  to be replaced with phase gates that can then be compiled together into fewer operations to reduce both the number of qubits and number of gates needed for the adder.

For more details, please refer to [M. Roetteler, Th. Beth](#) and [D. Coppersmith](#).

### Quantum Phase Estimation

One particularly important application of the quantum Fourier transform is to learn the eigenvalues of unitary operators, a problem known as *phase estimation*. Consider a unitary  $U$  and a state  $|\phi\rangle$  such that  $|\phi\rangle$  is an eigenstate of  $U$  with unknown eigenvalue  $\phi$ . If we only have access to  $U$  as an oracle, then we can learn the phase  $\phi$  by utilizing that  $Z$  rotations applied to the target of a controlled operation propagate back onto the control.

Suppose that  $V$  is a controlled application of  $U$ , such that  $V(|0\rangle \otimes |\phi\rangle) = |\phi\rangle \otimes |\phi\rangle$ . Then, by linearity,  $V(|+\rangle \otimes |\phi\rangle) = e^{i\phi} |+\rangle \otimes |\phi\rangle$ . We can collect terms to find that  $V(|+\rangle \otimes |\phi\rangle) = \frac{(|0\rangle \otimes |\phi\rangle) + e^{i\phi} (|+\rangle \otimes |\phi\rangle)}{\sqrt{2}}$ . We can collect terms to find that  $V(|+\rangle \otimes |\phi\rangle) = \frac{(|0\rangle + e^{i\phi} |+\rangle) \otimes |\phi\rangle}{\sqrt{2}}$ , where  $R_1$  is the unitary applied by the [R1 operation](#) operation. Put differently, the effect of applying  $V$  is precisely the same as applying  $R_1$  with an unknown angle, even though we only have access to  $V$  as an oracle. Thus, for the rest of this discussion we will discuss phase estimation in terms of  $R_1(\phi)$ , which we implement by using so-called *phase kickback*.

Since the control and target register remain untangled after this process, we can reuse  $|\phi\rangle$  as the target of a controlled application of  $U^2$  to prepare a second control qubit in the state  $R_1(2\phi) |\phi\rangle$ .

Continuing in this way, we can obtain a register of the form  $|\psi\rangle = \sum_{j=0}^n R_1(2^j\phi) |\phi\rangle \otimes |+\rangle \otimes \dots \otimes |+\rangle$  where  $n$  is the number of bits of precision that we require, and where we have used  $\{\}$  to indicate that we have suppressed the normalization factor of  $1/\sqrt{2^n}$ .

If we assume that  $\phi = 2\pi p / 2^k$  for an integer  $p$ , then we recognize this as  $|\psi\rangle = \text{QFT}(|p_0 p_1 \dots p_n\rangle)$ , where  $p_j$  is the  $j^{\text{th}}$  bit of  $2\pi\phi$ . Applying the adjoint of the quantum Fourier transform, we therefore obtain the binary representation of the phase encoded as a quantum state.

In Q#, this is implemented by the [QuantumPhaseEstimation operation](#) operation, which takes a [DiscreteOracle user defined type](#) implementing application of  $U^m$  as a function of positive integers  $m$ .

# Diagnostics

5/27/2021 • 8 minutes to read • [Edit Online](#)

As with classical development, it is important to be able to diagnose mistakes and errors in quantum programs. The Q# standard libraries provide a variety of different ways to ensure the correctness of quantum programs, as detailed in [Test and debug quantum programs](#). Largely speaking, this support comes in the form of functions and operations that either instruct the target machine to provide additional diagnostic information to the host program or developer, or enforce the correctness of conditions and invariants expressed by the function or operation call.

## Machine Diagnostics

Diagnostics about classical values can be obtained by using the [Message function](#) function to log a message in a machine-dependent way. By default, this writes the string to the console. Used together with interpolated strings, [Message function](#) makes it easy to report diagnostic information about classical values:

```
let angle = Microsoft.Quantum.Math.PI() * 2.0 / 3.0;
Message($"About to rotate by an angle of {angle}...");
```

### NOTE

`Message` has signature `(String -> Unit)`, again representing that emitting a debug log message cannot be observed from within Q#.

The [DumpMachine function](#) and [DumpRegister function](#) callables instruct target machines to provide diagnostic information about all currently allocated qubits or about a specific register of qubits, respectively. Each target machine varies in what diagnostic information is provided in response to a dump instruction. The [full state simulator](#) target machine, for instance, provides the host program with the state vector that it uses internally to represent a register of qubits. By comparison, the [Toffoli simulator](#) target machine provides a single classical bit for each qubit.

To learn more about the [full state simulator's](#) `DumpMachine` output, take a look at the dump functions section of our [testing and debugging article](#).

## Facts and Assertions

As discussed in [Testing and Debugging](#), a function or operation with signature `Unit -> Unit` or `Unit => Unit`, respectively, can be marked as a *unit test*. Each unit test generally consists of a small quantum program, along with one or more conditions that check the correctness of that program. These conditions can come in the form of either *facts*, which check the values of their inputs, or *assertions*, which check the states of one or more qubits passed as input.

For example, `EqualityFactI(1 + 1, 2, "1 + 1 != 2")` represents the mathematical fact that  $1 + 1 = 2$ , while `AssertQubit(One, qubit)` represents the condition that measuring `qubit` will return a `One` with certainty. In the former case, we can check the correctness of the condition given only its values, while in the latter, we must know something about the state of the qubit in order to evaluate the assertion.

The Q# standard libraries provide several different functions for representing facts, including:

- [Fact function](#)

- [EqualityWithinToleranceFact](#) function
- [NearEqualityFactC](#) function
- [EqualityFactI](#) function

## Testing Qubit States

In practice, assertions rely on the fact that classical simulations of quantum mechanics need not obey the [no-cloning theorem](#), such that we can make unphysical measurements and assertions when using a simulator for our target machine. Thus, we can test individual operations on a classical simulator before deploying on hardware. On target machines which do not allow evaluation of assertions, calls to [AssertMeasurement operation](#) can be safely ignored.

More generally, the [AssertMeasurement](#) operation asserts that measuring the given qubits in the given Pauli basis will always have the given result. If the assertion fails, the run ends by calling `fail` with the given message. By default, this operation is not implemented; simulators that can support it should provide an implementation that performs runtime checking. `AssertMeasurement` has signature

`((Pauli[], Qubit[], Result, String) -> ())`. Since `AssertMeasurement` is a function with an empty tuple as its output type, no effects from having called `AssertMeasurement` are observable within a Q# program.

The [AssertMeasurementProbability](#) operation function asserts that measuring the given qubits in the given Pauli basis will have the given result with the given probability, within some tolerance. Tolerance is additive (for example, `abs(expected-actual) < tol`). If the assertion fails, the run ends by calling `fail` with the given message. By default, this operation is not implemented; simulators that can support it should provide an implementation that performs runtime checking. `AssertMeasurementProbability` has signature

`((Pauli[], Qubit[], Result, Double, String, Double) -> Unit)`. The first of `Double` parameters gives the desired probability of the result, and the second one the tolerance.

We can do more than assert a single measurement, using that the classical information used by a simulator to represent the internal state of a qubit is amenable to copying, such that we do not need to actually perform a measurement to test our assertion. In particular, this allows us to reason about *incompatible* measurements that would be impossible on actual hardware.

Suppose that `P : Qubit => Unit` is an operation intended to prepare the state  $\lvert \psi \rangle$  when its input is in the state  $\lvert 0 \rangle$ . Let  $\lvert \psi' \rangle$  be the actual state prepared by `P`. Then,  $\lvert \psi' \rangle = \lvert \psi \rangle$  if and only if measuring  $\lvert \psi' \rangle$  in the axis described by  $\lvert \psi \rangle$  always returns `Zero`. That is, 
$$\lvert \psi \rangle = \lvert \psi' \rangle \text{ if and only if } \langle \psi' \mid \psi \rangle = 1.$$
 Using the primitive operations defined in the prelude, we can directly perform a measurement that returns `Zero` if  $\lvert \psi \rangle$  is an eigenstate of one of the Pauli operators.

The operation [AssertQubit](#) operation provides a particularly useful shorthand to do so in the case that we wish to test the assertion  $\lvert \psi \rangle = \lvert 0 \rangle$ . This is common, for instance, when we have uncomputed to return auxiliary qubits to  $\lvert 0 \rangle$  before releasing them. Asserting against  $\lvert 0 \rangle$  is also useful when we wish to assert that two state preparation `P` and `Q` operations both prepare the same state, and when `Q` supports `Adjoint`. In particular,

```
use register = Qubit();
P(register);
Adjoint Q(register);

AssertQubit(Zero, register);
```

More generally, however, we may not have access to assertions about states that do not coincide with eigenstates of Pauli operators. For example,  $\lvert \psi \rangle = (\lvert 0 \rangle + e^{i\pi/8}\lvert 1 \rangle)/\sqrt{2}$  is not an eigenstate of any Pauli operator, such that we cannot use [AssertMeasurementProbability](#) operation to uniquely

determine that a state  $\lvert \psi \rangle$  is equal to  $\lvert \psi \rangle$ . Instead, we must decompose the assertion  $\lvert \psi \rangle = \lvert \psi \rangle$  into assumptions that can be directly tested using the primitives supported by our simulator. To do so, let  $\lvert \psi \rangle = \alpha \lvert 0 \rangle + \beta \lvert 1 \rangle$  for complex numbers  $\alpha = a_r + a_i i$  and  $\beta$ . Note that this expression requires four real numbers  $\{a_r, a_i, b_r, b_i\}$  to specify, as each complex number can be expressed as the sum of a real and imaginary part. Due to the global phase, however, we can choose  $a_i = 0$ , such that we only need three real numbers to uniquely specify a single-qubit state.

Thus, we need to specify three assertions which are independent of each other in order to assert the state that we expect. We do so by finding the probability of observing `Zero` for each Pauli measurement given  $\alpha$  and  $\beta$ , and asserting each independently. Let  $x$ ,  $y$ , and  $z$  be `Result` values for Pauli  $X$ ,  $Y$ , and  $Z$  measurements respectively. Then, using the likelihood function for quantum measurements,

$$\Pr(x = \text{Zero} \mid \alpha, \beta) = \frac{1}{2} + a_r b_r + a_i b_i \\ \Pr(y = \text{Zero} \mid \alpha, \beta) = \frac{1}{2} \left( 1 + a_r^2 + a_i^2 + b_r^2 + b_i^2 \right)$$

The [AssertQubitsInStateWithinTolerance operation](#) implements these assertions given representations of  $\alpha$  and  $\beta$  as values of type [Complex user defined type](#). This is helpful when the expected state can be computed mathematically.

### Asserting Equality of Quantum Operations

Thus far, we have been concerned with testing operations which are intended to prepare particular states. Often, however, we are interested in how an operation acts for arbitrary inputs rather than for a single fixed input. For example, suppose we have implemented an operation `u : ((Double, Qubit[]) => () : Adjoint)` corresponding to a family of unitary operators  $U(t)$ , and have provided an explicit `adjoint` block instead of using `adjoint auto`. We may be interested in asserting that  $U^\dagger(t) = U(-t)$ , as expected if  $t$  represents an evolution time.

Broadly speaking, there are two different strategies that we can follow in making the assertion that two operations `u` and `v` act identically. First, we can check that `u(target); (Adjoint v)(target);` preserves each state in a given basis. Second, we can check that `u(target); (Adjoint v)(target);` acting on half of an entangled state preserves that entanglement. These strategies are implemented by the canon operations [AssertOperationsEqualInPlace operation](#) and [AssertOperationsEqualReferenced operation](#), respectively.

#### NOTE

The referenced assertion discussed above works based on the [Choi–Jamiolkowski isomorphism](#), a mathematical framework which relates operations on  $n$  qubits to entangled states on  $2n$  qubits. In particular, the identity operation on  $n$  qubits is represented by  $n$  copies of the entangled state  $\lvert \beta_{00} \rangle = (\lvert 00 \rangle + \lvert 11 \rangle) / \sqrt{2}$ . The operation [PrepareChoiState operation](#) implements this isomorphism, preparing a state that represents a given operation.

Roughly, these strategies are distinguished by a time–space tradeoff. Iterating through each input state takes additional time, while using entanglement as a reference requires storing additional qubits. In cases where an operation implements a reversible classical operation, such that we are only interested in its behavior on computational basis states, [AssertOperationsEqualInPlaceCompBasis operation](#) tests equality on this restricted set of inputs.

#### TIP

The iteration over input states is handled by the enumeration operations [IterateThroughCartesianProduct operation](#) and [IterateThroughCartesianPower operation](#). These operations are useful more generally for applying an operation to each element of the Cartesian product between two or more sets.

More critically, however, the two approaches test different properties of the operations under examination. Since the in-place assertion calls each operation multiple times, once for each input state, any random choices and measurement results might change between calls. By contrast, the referenced assertion calls each operation exactly once, such that it checks that the operations are equal *in a single shot*. Both of these tests are useful in ensuring the correctness of quantum programs.

## Further Reading

- [Test and debug quantum programs](#)
- [Microsoft.Quantum.Diagnostics namespace](#)

# Quantum characterization and statistics

5/27/2021 • 13 minutes to read • [Edit Online](#)

It is critical to be able to characterize the effects of operations in order to develop useful quantum algorithms. This is challenging because every measurement of a quantum system yields at most one bit of information. In order to learn an eigenvalue, let alone a quantum state, the results of many measurements must be stitched together so that the user can glean the many bits of information needed to represent these concepts. Quantum states are especially vexing because the [no-cloning theorem](#) states that there is no way to learn an arbitrary quantum state from a single copy of the state, because doing so would let you make copies of the state. This obfuscation of the quantum state from the user is reflected in the fact that Q# does not expose or even define what a state *is* to quantum programs. We thus approach quantum characterization by treating operations and states as black-box; this approach shares much in common with the experimental practice of quantum characterization, verification and validation (QCVV).

Characterization is distinct from many of the other libraries discussed previously. The aim here is less to learn classical information about the system, rather than to perform a unitary transformation on a state vector. These libraries must therefore blend both classical and quantum information processing.

## Iterative phase estimation

Viewing quantum programming in terms of quantum characterization suggests a useful alternative to quantum phase estimation. That is, instead of preparing an  $\$n\$$ -qubit register to contain a binary representation of the phase as in quantum phase estimation, we can view phase estimation as the process by which a *classical* agent learns properties of a quantum system through measurements. We proceed as in the quantum case by using phase kickback to turn applications of a black-box operation into rotations by an unknown angle, but will measure the auxiliary qubit that we rotate at each step immediately following the rotation. This has the advantage that we only require a single additional qubit to perform the phase kickback described in the quantum case, as we then learn the phase from the measurement results at each step in an iterative fashion. Each of the methods proposed below uses a different strategy for designing experiments and different data processing methods to learn the phase. They each have unique advantage ranging from having rigorous error bounds, to the abilities to incorporate prior information, tolerate errors or run on memory limited classical computers.

In discussing iterative phase estimation, we will consider a unitary  $\$U\$$  given as a black-box operation. As described in the section on oracles in [data structures](#), the Q# canon models such operations by the [DiscreteOracle user defined type](#) user-defined type, defined by the tuple type

`((Int, Qubit[]) => Unit : Adjoint, Controlled)`. Concretely, if `u : DiscreteOracle`, then `u(m)` implements  $\$U^m\$$  for `m : Int`.

With this definition in place, each step of iterative phase estimation proceeds by preparing an auxiliary qubit in the  $\ket{+}$  state along with the initial state  $\ket{\phi}$  that we assume is an [eigenvector](#) of  $\$U(m)$ , for example,  $\$U(m)\ket{\phi}= e^{im\phi}\ket{\phi}$ .

A controlled application of `u(m)` is then used which prepares the state  $\left|\text{left}(R_1(m \phi) \ket{+} \right\rangle$ . As in the quantum case, the effect of a controlled application of the oracle `u(m)` is precisely the same as the effect of applying  $\$R_1\$$  for the unknown phase on  $\ket{+}$ , such that we can describe the effects of  $\$U\$$  in this simpler fashion. Optionally, the algorithm then rotates the control qubit by applying  $\$R_1(-m\theta)\$$  to obtain a state  $\ket{\psi}=\left|\text{left}(R_1(m [\phi-\theta]) \ket{+}\right\rangle$ . The auxiliary qubit used as a control for `u(m)` is then measured in the  $\$X\$$  basis to obtain a single classical `Result`.

At this point, reconstructing the phase from the `Result` values obtained through iterative phase estimation is a

classical statistical inference problem. Finding the value of  $m$  that maximizes the information gained, given a fixed inference method, is simply a problem in statistics. We emphasize this by briefly describing iterative phase estimation at a theoretical level in the Bayesian parameter estimation formalism before proceeding to describe the statistical algorithms provided in the Q# canon for solving this classical inference problem.

### Iterative phase estimation without eigenstates

If an input state is provided that is not an eigenstate, which is to say that if  $|U(m)\psi\rangle = e^{im\phi_j}|U\psi\rangle$  then the process of phase estimation non-deterministically guides the quantum state towards a single energy eigenstate. The eigenstate it ultimately converges to is the eigenstate that is most likely to produce the observed **Result**.

Specifically, a single step of PE performs the following non-unitary transformation on a state  $\begin{aligned} & \sum_j \sqrt{\Pr(\phi_j)} |j\rangle \mapsto \\ & \sum_j \frac{\sqrt{\Pr(\phi_j)}}{\sqrt{\Pr(\text{Result})}} |j\rangle \langle j| \sqrt{\Pr(\phi_j)} \sum_j \Pr(\text{Result}) |\phi_j\rangle \end{aligned}$ . As this process is iterated over multiple **Result** values, eigenstates that do not have maximal values of  $\prod_k \Pr(\text{Result}_k)$  will be exponentially suppressed. As a result, the inference process will tend to converge to states with a single eigenvalue if the experiments are chosen properly.

Bayes' theorem further suggests that the state that results from phase estimation be written in the form  $\begin{aligned} & \frac{\sqrt{\Pr(\phi_j)}}{\sqrt{\Pr(\text{Result})}} |j\rangle \langle j| \sqrt{\Pr(\phi_j)} \sum_j \Pr(\text{Result}) |\phi_j\rangle = \sum_j \sqrt{\Pr(\phi_j|\text{Result})} |j\rangle. \end{aligned}$  Here  $\Pr(\phi_j|\text{Result})$  can be interpreted as the probability that one would ascribe to each hypothesis about the eigenstates given:

1. knowledge of the quantum state prior to measurement,
2. knowledge of the eigenstates of  $U$  and,
3. knowledge of the eigenvalues of  $U$ .

Learning these three things is often exponentially hard on a classical computer. The utility of phase estimation arises, to no small extent, from the fact that it can perform such a quantum learning task without knowing any of them. Phase estimation for this reason appears within a number of quantum algorithms that provide exponential speedups.

### Bayesian phase estimation

#### TIP

For more details on Bayesian phase estimation in practice, please see the [PhaseEstimation](#) sample.

The idea of Bayesian phase estimation is simple. You collect measurement statistics from the phase estimation protocol and then you process the results using Bayesian inference and provide an estimate of the parameter. This processing gives you an estimate of the eigenvalue as well as the uncertainty in that estimate. It also allows you to perform adaptive experiments and utilize prior information. The methods' principle drawback is that it is computationally demanding.

To understand how this Bayesian inference process works, consider the case of processing a single **zero** result. Note that  $X = |\psi\rangle\langle\psi| - |\phi\rangle\langle\phi|$ , such that  $|\psi\rangle$  is the only positive eigenstate of  $X$  corresponding to **zero**. The probability of observing **zero** for a **PauliX** measurement on the first qubit given an input state  $|\psi\rangle$  is thus  $\Pr(\text{Zero}|\psi) = |\langle\psi|X|\psi\rangle|^2$ . In the case of iterative phase estimation, we have that  $|\psi\rangle = R_1(m|\phi\rangle) + R_1(m|\phi\rangle)$ , such that  $\Pr(\text{Zero}|\phi; m|\theta) = |\langle\phi|X|\phi\rangle|^2 = |\langle\phi|R_1(m|\phi\rangle)|^2 = |\langle\phi|R_1(m|\phi\rangle)|^2 = |\langle\phi|e^{im(\phi-\theta)}|\phi\rangle|^2 = |\langle\phi|e^{im(\phi-\theta)}|\phi\rangle|^2 = \cos^2(m|\phi-\theta|)$ . That is, iterative phase estimation consists of learning the oscillation frequency of a sinusoidal function, given the ability to flip a coin with a bias given by that sinusoid.

Following traditional classical terminology, we call  $\text{\eqref{eq:phase-est-likelihood}}$  the *likelihood function* for iterative phase estimation.

Having observed a `Result` from the iterative phase estimation likelihood function, we can then use Bayes' rule to prescribe what we should believe the phase to be following that observation. Concretely, 
$$\Pr(\phi | d) = \frac{\Pr(d | \phi)}{\Pr(\phi)} \Pr(\phi)$$
 where  $d$  in `{Zero, One}` is a `Result`, and where  $\Pr(\phi)$  describes our prior beliefs about  $\phi$ . This then makes the iterative nature of iterative phase estimation explicit, as the posterior distribution  $\Pr(\phi | d)$  describes our beliefs immediately preceding our observation of the next `Result`.

At any point during this procedure, we can report the phase  $\hat{\phi}$  inferred by the classical controller as 
$$\hat{\phi} := \mathbb{E}[\phi | \text{data}] = \int \phi \Pr(\phi | \text{data}) d\phi$$
 where `data` stands for the entire record of all `Result` values obtained.

Exact Bayesian inference is in practice intractable. To see this imagine we wish to learn an  $n$ -bit variable  $x$ . The prior distribution  $\Pr(x)$  has support over  $2^n$  hypothetical values of  $x$ . This means that if we need a highly accurate estimate of  $x$  then Bayesian phase estimation may need prohibitive memory and processing time. While for some applications, such as quantum simulation, the limited accuracy required does not preclude such methods other applications, such as Shor's algorithm, cannot use exact Bayesian inference within its phase estimation step. For this reason, we also provide implementations for approximate Bayesian methods such as [random walk phase estimation \(RWPE\)](#) and also non-Bayesian approaches such as [robust phase estimation](#).

## Robust phase estimation

A maximum *a posteriori* Bayesian reconstruction of a phase estimate from measurement results is exponentially hard in the worst-case. Thus most practical phase estimation algorithms sacrifice some quality in the reconstruction, in exchange for an amount of classical post-processing that instead scales polynomially with the number of measurements made.

One such example with an efficient classical post-processing step is the [robust phase estimation algorithm](#), with its signature and inputs mentioned above. It assumes that input unitary black-boxes `U` are packaged as `DiscreteOracle` type, and therefore only queries integer powers of controlled-`U`. If the input state in the `Qubit[]` register is an eigenstate  $U|\psi\rangle = e^{i\phi}|\psi\rangle$ , the robust phase estimation algorithm returns an estimate  $\hat{\phi}$  of  $\phi$  as a `Double`.

The most important feature of robust phase estimation, which is shared with most other useful variants, is that the reconstruction quality of  $\hat{\phi}$  is in some sense Heisenberg-limited. This means that if the deviation of  $\hat{\phi}$  from the true value is  $\sigma$ , then  $\sigma$  scales inversely-proportional to the total number of queries  $Q$  made to controlled-`U`, for example,  $\sigma = \mathcal{O}(1/Q)$ . Now, the definition of deviation varies between different estimation algorithms. In some cases, it may mean that with at least  $\mathcal{O}(1)$  probability, the estimation error  $|\hat{\phi} - \phi| \leq \sigma$  on some circular measure  $\circ$ . For robust phase estimation, deviation is precisely the variance  $\sigma^2 = \mathbb{E}[\hat{\phi}^2] - (\mathbb{E}[\hat{\phi}])^2$  if we unwrap periodic phases onto a single finite interval  $(-\pi, \pi]$ . More precisely, the standard deviation in robust phase estimation satisfies the inequalities 
$$2.0 \pi / Q \leq \sigma \leq 2\pi / 2^n \leq 10.7 \pi / Q$$
 where the lower bound is reached in the limit of asymptotically large  $Q$ , and the upper bound is guaranteed even for small sample sizes. Note that  $n$  selected by the `bitsPrecision` input, which implicitly defines  $Q$ .

Other relevant details include, say, the small space overhead of just  $1$  auxiliary qubit, or that the procedure is non-adaptive, meaning the required sequence of quantum experiments is independent of the intermediate measurement outcomes. In this and forthcoming examples where the choice of phase estimation algorithm is important, one should refer to the documentation such as [RobustPhaseEstimation operation](#) and the referenced publications therein for more information and for their implementation.

## TIP

There are many samples where robust phase estimation is used. For phase estimation in extracting the ground state energy of various physical system, please see the [H2 simulation sample](#), the [SimpleIsing sample](#), and the [Hubbard model sample](#).

## Continuous oracles

We can also generalize from the oracle model used above to allow for continuous-time oracles, modeled by the canon type [ContinuousOracle user defined type](#). Consider that instead of a single unitary operator  $\$U\$$ , we have a family of unitary operators  $\$U(t)\$$  for  $t \in \mathbb{R}$  such that  $\$U(t) U(s)\$ = \$U(t + s)\$$ . This is a weaker statement than in the discrete case, since we can construct a [DiscreteOracle user defined type](#) by restricting  $t = m, \delta t$  for some fixed  $\delta t$ . By [Stone's theorem](#),  $\$U(t) = \exp(i H t)\$$  for some operator  $\$H\$$ , where  $\exp$  is the matrix exponential as described in [advanced matrices](#). An eigenstate  $\ket{\phi}$  of  $\$H\$$  such that  $\$H \ket{\phi} = \phi \ket{\phi}\$$  is then also an eigenstate of  $\$U(t)\$$  for all  $t$ ,  $\begin{aligned} U(t) \ket{\phi} &= e^{i \phi t} \ket{\phi}. \end{aligned}$

The exact same analysis discussed for [Bayesian phase estimation](#) can be applied, and the likelihood function is precisely the same for this more general oracle model:  $\Pr(\text{Zero} | \phi, \theta) = \cos^2(\frac{\phi - \theta}{2})$ . Moreover, if  $\$U\$$  is a simulation of a dynamical generator, as is the case for [Hamiltonian simulation](#), we interpret  $\phi$  as an energy. Thus, using phase estimation with continuous queries allows us to learn the simulated [energy spectrum of molecules, materials or field theories](#) without having to compromise our choice of experiments by requiring  $t$  to be an integer.

## Random walk phase estimation

Q# provides a useful approximation of Bayesian phase estimation designed for use close to quantum devices that operates by conditioning a random walk on the data record obtained from iterative phase estimation. This method is both adaptive and entirely deterministic, allowing for near-optimal scaling of errors in the estimated phase  $\hat{\phi}$  with very low memory overheads.

The protocol uses an approximate Bayesian inference method that assumes the prior distribution is Gaussian. This Gaussian assumption allows us to use an analytical formula for the experiment that minimizes the posterior variance. The algorithm then, based on the outcome of that experiment, shifts the estimate of  $\phi$  left or right by a pre-determined amount and shrinks the variance by a pre-determined amount. This mean and variance give all the information that is needed to specify a Gaussian prior on  $\phi$  for the next experiment. Unexpected measurement failures, or the true result being on the tails of the initial prior, can cause this method to fail. It recovers from failure by performing experiments to test whether the current mean and standard deviation are appropriate for the system. If they are not, then the algorithm does an inverse step of the walk and the process continues. The ability to step backwards also allows the algorithm to learn even if the initial prior standard deviation is inappropriately small.

## Calling phase estimation algorithms

Each phase estimation operation provided with the Q# canon takes a different set of inputs parameterizing the quality that we demand out of the final estimate  $\hat{\phi}$ . These various inputs, however, all share several inputs in common, such that partial application over the quality parameters results in a common signature. For example, the [RobustPhaseEstimation operation](#) operation discussed in the next section has the following signature:

```
operation RobustPhaseEstimation(bitsPrecision : Int, oracle : DiscreteOracle, eigenstate : Qubit[]) : Double
```

The `bitsPrecision` input is unique to `RobustPhaseEstimation`, while `oracle` and `eigenstate` are in common.

Thus, as seen in **H2Sample**, an operation can accept an iterative phase estimation algorithm with an input of the form `(DiscreteOracle, Qubit[]) => Unit` to allow a user to specify arbitrary phase estimation algorithms:

```
operation H2EstimateEnergy(
    idxBondLength : Int,
    trotterStepSize : Double,
    phaseEstAlgorithm : ((DiscreteOracle, Qubit[]) => Double))
: Double
```

These myriad phase estimation algorithms are optimized for different properties and input parameters, which must be understood to make the best choice for the target application. For instance, some phase estimation algorithms are adaptive, meaning that future steps are classically controlled by the measurement results of previous steps. Some require the ability to exponentiate its black-box unitary oracle by arbitrary real powers, and others only require integer powers but are only able to resolve a phase estimate modulo  $2\pi$ . Some require many auxiliary qubits, and others require only one.

Similarly, using random walk phase estimation proceeds in much the same way as for other algorithms provided with the canon:

```
operation ApplyExampleOracle(
    eigenphase : Double,
    time : Double,
    register : Qubit[])
: Unit is Adj + Ctl {
    Rz(2.0 * eigenphase * time, register[0]);
}

operation EstimateBayesianPhase(eigenphase : Double) : Double {
    let oracle = ContinuousOracle(ApplyExampleOracle(eigenphase, _,_));
    use eigenstate = Qubit();
    X(eigenstate);
    // The additional inputs here specify the mean and variance of the prior, the number of
    // iterations to perform, how many iterations to perform as a maximum, and how many
    // steps to roll back on an approximation failure.
    let est = RandomWalkPhaseEstimation(0.0, 1.0, 61, 100000, 1, oracle, [eigenstate]);
    Reset(eigenstate);
    return est;
}
```

# Error Correction

5/27/2021 • 6 minutes to read • [Edit Online](#)

## Introduction

In classical computing, if one wants to protect a bit against errors, it can often suffice to represent that bit by a *logical bit* by repeating the data bit. For instance, let  $\overline{0} = 000$  be the encoding of the data bit 0, where we use the a line above the label 0 to indicate that it is an encoding of a bit in the 0 state. If we similarly let  $\overline{1} = 111$ , then we have a simple repetition code that protects against any one bit flip error. That is, if any of the three bits are flipped, then we can recover the state of the logical bit by taking a majority vote. Though classical error correction is a much richer subject than this particular example (we recommend [Lint's introduction to coding theory](#)), the repetition code above already points to a possible problem in protecting quantum information. Namely, the [no-cloning theorem](#) implies that if we measure each individual qubit and take a majority vote by analogy to classical code above, then we have lost the precise information that we are trying to protect.

In the quantum setting, we will see that the measurement is problematic. We can still implement the encoding above. It is helpful to do so to see how we can generalize error correction to the quantum case. Thus, let  $\ket{\overline{0}} = \ket{000} = \ket{0} \otimes \ket{0} \otimes \ket{0}$ , and let  $\ket{\overline{1}} = \ket{111}$ . Then, by linearity, we have defined our repetition code for all inputs; for instance,  $\ket{\overline{+}} = (\ket{\overline{0}} + \ket{\overline{1}}) / \sqrt{2} = (\ket{000} + \ket{111}) / \sqrt{2}$ . In particular, letting a bit-flip error  $X_1$  act on the middle qubit, we see that the correction needed in both branches is precisely  $X_1$ :  
$$\begin{aligned} X_1 \ket{\overline{+}} &= \frac{1}{\sqrt{2}} (\ket{000} + \ket{111}) \\ &= \frac{1}{\sqrt{2}} (\ket{000} + \ket{010} + \ket{101}) \end{aligned}$$

To see how we can identify that this is the case without measuring the very state we are trying to protect, it is helpful to write down what each different bit flip error does to our logical states:

ERROR \$E\$	$\ket{\overline{0}}$	$\ket{\overline{1}}$
$\boldsymbol{\text{done}}$	$\ket{000}$	$\ket{111}$
$X_0$	$\ket{100}$	$\ket{011}$
$X_1$	$\ket{010}$	$\ket{101}$
$X_2$	$\ket{001}$	$\ket{110}$

In order to protect the state that we're encoding, we need to be able to distinguish the three errors from each other and from the identity  $\text{done}$  without distinguishing between  $\ket{\overline{0}}$  and  $\ket{\overline{1}}$ . For example, if we measure  $Z_0$ , we get a different result for  $\ket{\overline{0}}$  and  $\ket{\overline{1}}$  in the no-error case, so that collapses the encoded state. On the other hand, consider measuring  $Z_0 Z_1$ , the parity of the first two bits in each computational basis state. Recall that each measurement of a Pauli operator checks which eigenvalue the state being measured corresponds to, so for each state  $\ket{\psi}$  in the table above, we can compute  $Z_0 Z_1 \ket{\psi}$  to see if we get  $\pm \ket{\psi}$ . Note that  $Z_0 Z_1 \ket{000} = \ket{000}$  and that  $Z_0 Z_1 \ket{111} = -\ket{111}$ , so that we conclude that this measurement does the same thing to both encoded states. On the other hand,  $Z_0 Z_1 \ket{100} = -\ket{100}$  and  $Z_0 Z_1 \ket{011} = \ket{011}$ , so the result of measuring  $Z_0 Z_1$  reveals useful information about which error occurred.

To emphasize this, we repeat the table above, but add the results of measuring  $Z_0 Z_1$  and  $Z_1 Z_2$  on each row. We denote the results of each measurement by the sign of the eigenvalue that is observed, either  $+$  or  $-$ , corresponding to the Q# `Result` values of `Zero` and `One`, respectively.

ERROR \$E\$	$E\ket{\overline{0}}$	$E\ket{\overline{1}}$	RESULT OF $Z_0 Z_1$	RESULT OF $Z_1 Z_2$
$\boldsymbol{done}$	$\ket{000}$	$\ket{111}$	$+$	$+$
$X_0$	$\ket{100}$	$\ket{011}$	$-$	$+$
$X_1$	$\ket{010}$	$\ket{101}$	$-$	$-$
$X_2$	$\ket{001}$	$\ket{110}$	$+$	$-$

Thus, the results of the two measurements uniquely determines which bit-flip error occurred, but without revealing any information about which state we encoded. We call these results a *syndrome*, and refer to the process of mapping a syndrome back to the error that caused it as *recovery*. In particular, we emphasize that recovery is a *classical*/inference procedure which takes as its input the syndrome which occurred, and returns a prescription for how to fix any errors that may have occurred.

#### NOTE

The bit-flip code above can only correct against single bit-flip errors; that is, an `x` operation acting on a single qubit. Applying `x` to more than one qubit will map  $\ket{\overline{0}}$  to  $\ket{\overline{1}}$  following recovery. Similarly, applying a phase flip operation `z` will map  $\ket{\overline{1}}$  to  $-\ket{\overline{1}}$ , and hence will map  $\ket{\overline{+}}$  to  $\ket{\overline{-}}$ . More generally, codes can be created to handle larger number of errors, and to handle  $Z$  errors as well as  $X$  errors.

The insight that we can describe measurements in quantum error correction that act the same way on all code states, is the essence of the *stabilizer formalism*. The Q# canon provides a framework for describing encoding into and decoding from stabilizer codes, and for describing how one recovers from errors. In this section, we describe this framework and its application to a few simple quantum error-correcting codes.

#### TIP

A full introduction to the stabilizer formalism is beyond the scope of this section. We refer readers interested in learning more to [Gottesman 2009](#).

## Representing Error Correcting Codes in Q#

To help specify error correcting codes, the Q# canon provides several distinct user-defined types:

- **LogicalRegister user defined type** `= Qubit[]`: Denotes that a register of qubits should be interpreted as the code block of an error-correcting code.
- **Syndrome user defined type** `= Result[]`: Denotes that an array of measurement results should be interpreted as the syndrome measured on a code block.
- **RecoveryFn user defined type** `= (Syndrome -> Pauli[])`: Denotes that a *classical*/function should be used to interpret a syndrome and return a correction that should be applied.
- **EncodeOp user defined type** `= ((Qubit[], Qubit[]) -> LogicalRegister)`: Denotes that an operation takes qubits representing data along with fresh auxiliary qubits in order to produce a code block of an error-correcting code.

- **DecodeOp user defined type** = `(LogicalRegister => (Qubit[], Qubit[]))` : Denotes than an operation decomposes a code block of an error correcting code into the data qubits and the auxiliary qubits used to represent syndrome information.
- **SyndromeMeasOp user defined type** = `(LogicalRegister => Syndrome)` : Denotes an operation that should be used to extract syndrome information from a code block, without disturbing the state protected by the code.

Finally, the canon provides the [QECC user defined type](#) type to collect the other types required to define a quantum error-correcting code. Associated with each stabilizer quantum code is the code length `n$`, the number `k$` of logical qubits, and the minimum distance `d$`, often conveniently grouped together in the notation `n$, k$, d$`. For example, the [BitFlipCode function](#) function defines the `3, 1` bit flip code:

```
let encodeOp = EncodeOp(BitFlipEncoder);
let decodeOp = DecodeOp(BitFlipDecoder);
let syndMeasOp = SyndromeMeasOp(MeasureStabilizerGenerators([
    [PauliZ, PauliZ, PauliI],
    [PauliI, PauliZ, PauliZ]
], _, MeasureWithScratch));
let code = QECC(encodeOp, decodeOp, syndMeasOp);
```

Notice that the `QECC` type does *not* include a recovery function. This allows us to change the recovery function that is used in correcting errors without changing the definition of the code itself; this ability is in particular useful when incorporating feedback from characterization measurements into the model assumed by recovery.

Once a code is defined in this way, we can use the [Recover operation](#) operation to recover from errors:

```
let code = BitFlipCode();
let fn = BitFlipRecoveryFn();
let X0 = ApplyPauli([PauliX, PauliI, PauliI],_);
use scratch = Qubit[nScratch];
let logicalRegister = encode(data, scratch);
// Cause an error.
X0(logicalRegister);
Recover(code, fn, logicalRegister);
let (decodedData, decodedScratch) = decode(logicalRegister);
ApplyToEach(Reset, decodedScratch);
```

We explore this in more detail in the [bit flip code sample](#).

Aside from the bit-flip code, the Q# canon is provided with implementations of the [five-qubit perfect code](#), and the [seven-qubit code](#), both of which can correct an arbitrary single-qubit error.

# Applications

5/27/2021 • 14 minutes to read • [Edit Online](#)

## Hamiltonian simulation

The simulation of quantum systems is one of the most exciting applications of quantum computation. On a classical computer, the difficulty of simulating quantum mechanics, in general, scales with the dimension  $N$  of its state-vector representation. As this representation grows exponentially with the number of  $n$  qubits  $N=2^n$ , a trait known also known as the [curse of dimensionality](#), quantum simulation on classical hardware is intractable.

However, the situation can be very different on quantum hardware. The most common variation of quantum simulation is called the time-independent Hamiltonian simulation problem. There, one is provided with a description of the system Hamiltonian  $H$ , which is a Hermitian matrix, and some initial quantum state  $|\psi(0)\rangle$  that is encoded in some basis on  $n$  qubits on a quantum computer. As quantum states in closed systems evolve under the Schrödinger equation  $\frac{d|\psi(t)\rangle}{dt} = H|\psi(t)\rangle$  the goal is to implement the unitary time-evolution operator  $U(t)=e^{-iHt}$  at some fixed time  $t$ , where  $|\psi(t)\rangle=U(t)|\psi(0)\rangle$  solves the Schrödinger equation. Analogously, the time-dependent Hamiltonian simulation problem solves the same equation, but with  $H(t)$  now a function of time.

Hamiltonian simulation is a major component of many other quantum simulation problems, and solutions to Hamiltonian simulation problem are algorithms that describes a sequence of primitive quantum gates for synthesizing an approximating unitary  $\tilde{U}$  with error  $\|\tilde{U} - U\| \leq \epsilon$  in the [spectral norm](#). The complexity of these algorithms depend very strongly on how a description of the Hamiltonian of interest is made accessible by a quantum computer. For instance, in the worst-case, if  $H$  acting on  $n$  qubits were to be provided as a list of  $2^n \times 2^n$  numbers, one for each matrix element, simply reading the data would already require exponential time. In the best case, one could assume access to a black-box unitary that  $O|\psi(0)\rangle = |\psi(t)\rangle$  trivially solves the problem. Neither of these input models are particularly interesting -- the former as it is no better than classical approaches, and the latter as the black-box hides the primitive gate complexity of its implementation, which could be exponential in the number of qubits.

### Descriptions of Hamiltonians

Additional assumptions of the format of the input are therefore required. A fine balance must be struck between input models that are sufficiently descriptive to encompass interesting Hamiltonians, such as those for realistic physical systems or interesting computational problems, and input models that are sufficiently restrictive to be efficiently implementable on a quantum computer. A variety of non-trivial input model may be found in the literature, and they range from quantum to classical.

As examples of quantum input models, [sample-based Hamiltonian simulation](#) assumes black-box access to quantum operations that produce copies of a density matrix  $\rho$ , which are taken to be the Hamiltonian  $H$ . In the [unitary access model](#) one supposes that the Hamiltonian instead decomposes into a sum of unitaries  $H = \sum_{j=0}^{d-1} a_j \hat{U}_j$  where  $a_j$  are coefficients, and  $\hat{U}_j$  are unitaries. It is then assumed that one has black-box access to the unitary oracle  $V = \sum_{j=0}^{d-1} \ket{j}\bra{j} \otimes \hat{U}_j$  that selects the desired  $\hat{U}_j$ , and the oracle  $A|\psi\rangle = \sum_{j=0}^{d-1} a_j \sqrt{\sum_{k=0}^{d-1} \alpha_{jk}} \ket{k}$  that create a quantum state encoding these coefficients. In the case of [sparse Hamiltonian simulation](#), one assumes that the Hamiltonian is a sparse matrix with only  $d=\mathcal{O}(\text{polylog}(N))$  non-zero element in every row. Moreover, one assumes the existence of efficient quantum circuits that output the location of these non-zero elements, as well as the their values. The

complexity of [Hamiltonian simulation algorithms](#) is evaluated in terms of number of queries to these black-boxes, and the primitive gate complexity then depends very much on the difficulty of implementing these black-boxes.

#### NOTE

The big-O notation is commonly used to describe the complexity scaling of algorithms. Given two real functions  $f, g$ , the expression  $g(x) = \mathcal{O}(f(x))$  means that there exists an absolute positive constant  $x_0, c > 0$  such that  $|g(x)| \leq c|f(x)|$  for all  $x \geq x_0$ .

In most practical applications to be implemented on a quantum computer, these black-boxes must be efficiently implementable, that is with  $\mathcal{O}(\text{polylog}(N))$  primitive quantum gates. More strongly, efficiently simulable Hamiltonians must have some sufficiently sparse classical description. In one such formulation, it is assumed that the Hamiltonian decomposes into a sum of Hermitian parts  $H = \sum_{j=0}^d H_j$ . Moreover, it is assumed that each part, a Hamiltonian  $H_j$ , is easy to simulate. This means that the unitary  $e^{-iH_j t}$  for any time  $t$  may be implemented exactly using  $\mathcal{O}(1)$  primitive quantum gates. For instance, this is true in the special case where each  $H_j$  are local Pauli operators, meaning that they are of tensor products of  $\mathcal{O}(1)$  non-identity Pauli operators that act on spatially close qubits. This model is particularly applicable to physical systems with bounded and local interaction, as the number of terms is  $d = \mathcal{O}(\text{polylog}(N))$ , and may clearly be written down, that is, classically described, in polynomial time.

#### TIP

Hamiltonians that decompose into a sum of parts may be described using the [Dynamical Generator Representation library](#). For more information, see the [Dynamical Generator Representation section in data structures](#).

## Simulation algorithms

A quantum simulation algorithm converts a given description of a Hamiltonian into a sequence of primitive quantum gates that, as a whole, approximate time-evolution by said Hamiltonian.

In the special case where the Hamiltonian decomposes into a sum of Hermitian parts, the Trotter-Suzuki decomposition is a particularly simple and intuitive algorithm for simulating Hamiltonians that decompose into a sum of Hermitian components. For instance, a first-order integrator of this family approximates  $U(t) = \left( e^{-iH_0 t/r} e^{-iH_1 t/r} \cdots e^{-iH_{d-1} t/r} \right)^r + \mathcal{O}(d^2 \max_j |H_j|^2 t^2/r)$  using a product of  $r$  terms.

#### TIP

Applications of the Trotter-Suzuki simulation algorithm are covered in the samples. For the Ising model using only the intrinsic operations provided by each target machine, please see the [SimpleIsing sample](#). For the Ising model using the Trotter-Suzuki library control structure, please see the [IsingTrotter sample](#). For molecular Hydrogen using the Trotter-Suzuki library control structure, please see the [H2 simulation sample](#).

In many cases, we would like to implement the simulation algorithm, but are not interested in the details of its implementation. For instance, the second-order integrator approximates  $U(t) = \left( e^{-iH_0 t/2r} e^{-iH_1 t/2r} \cdots e^{-iH_{d-1} t/2r} e^{-iH_{d-1} t/2r} e^{-iH_1 t/2r} e^{-iH_0 t/2r} \right)^r + \mathcal{O}(d^3 \max_j |H_j|^3 t^3/r^2)$  using a product of  $2rd$  terms. Larger orders will involve even more terms and optimized variants may require highly non-trivial orderings on the exponentials. Other advanced algorithms may also involve the use of auxiliary qubits in intermediate steps. Thus we package simulation algorithms in the canon as the user-defined type

```
newtype SimulationAlgorithm = ((Double, EvolutionGenerator, Qubit[]) => Unit) is Adj + Ctl;
```

The first parameter `Double` is the time of simulation, the second parameter `EvolutionGenerator`, covered in the Dynamical Generator Representation section of [data-structures](#), is a classical description of a time-independent Hamiltonian packaged with instructions on how each term in the Hamiltonian may be simulated by a quantum circuit. Types of this form approximate the unitary operation  $e^{-iHt}$  on the third parameter `Qubit[]`, which is the register storing the quantum state of the simulated system. Similarly for the time-dependent case, we define a user-defined type with an `EvolutionSchedule` type instead, which is a classical description of a time-dependent Hamiltonian.

```
newtype TimeDependentSimulationAlgorithm = ((Double, EvolutionSchedule, Qubit[]) => Unit) is Adjoint, Controlled;
```

As an example, the Trotter-Suzuki decomposition may be called using the following canon functions, with parameters `trotterStepSize` modifying the duration of simulation in each exponential, and `trotterOrder` for the order of the desired integrator.

```
function TrotterSimulationAlgorithm(
    trotterStepSize : Double,
    trotterOrder : Int)
: SimulationAlgorithm {
    ...
}

function TimeDependentTrotterSimulationAlgorithm(
    trotterStepSize : Double,
    trotterOrder : Int)
: TimeDependentSimulationAlgorithm {
    ...
}
```

#### TIP

Applications of the simulation library are covered in the samples. For phase estimation in the Ising model using `SimulationAlgorithm`, please see the [IsingPhaseEstimation](#) sample. For adiabatic state preparation in the Ising model using `TimeDependentSimulationAlgorithm`, please see the [AdiabaticIsing](#) sample.

### Adiabatic state preparation & phase estimation

One common application of Hamiltonian simulation is adiabatic state preparation. Here, one is provided with two Hamiltonians  $H_{\text{start}}$  and  $H_{\text{end}}$ , and a quantum state  $|\psi(0)\rangle$  that is a ground state of the start Hamiltonian  $H_{\text{start}}$ . Typically,  $H_{\text{start}}$  is chosen such that  $|\psi(0)\rangle$  is easy to prepare from a computational basis state  $|0\rangle$ . By interpolating between these Hamiltonians in the time-dependent simulation problem sufficiently slowly, it is possible to end up, with high probability, in a ground state of the final Hamiltonian  $H_{\text{end}}$ . Though preparing good approximations to Hamiltonian ground states could proceed in this manner by calling upon time-dependent Hamiltonian simulation algorithms as a subroutine, other conceptually different approaches such as the variational quantum eigensolver are possible.

Yet another application ubiquitous in quantum chemistry is estimating the ground state energy of Hamiltonians representing the intermediate steps of chemical reaction. Such a scheme could, for instance, rely on adiabatic state preparation to create the ground state, and then incorporate time-independent Hamiltonian simulation as a subroutine in phase estimation characterization to extract this energy with some finite error and probability of success.

Abstracting simulation algorithms as the user-defined types `SimulationAlgorithm` and `TimeDependentSimulationAlgorithm` allow us to conveniently incorporate their functionality into more sophisticated quantum algorithms. This motivates us to do the same for these commonly used subroutines.

Thus we define the convenient function

```
function InterpolatedEvolution(
    interpolationTime : Double,
    evolutionGeneratorStart : EvolutionGenerator,
    evolutionGeneratorEnd : EvolutionGenerator,
    timeDependentSimulationAlgorithm : TimeDependentSimulationAlgorithm)
: (Qubit[] => Unit is Adj + Ctl) {
    ...
}
```

This returns a unitary operation that implements all steps of adiabatic state preparation. The first parameter `interpolationTime` defines the time over which we linearly interpolate between the start Hamiltonian described by the second parameter `evolutionGeneratorStart` and the end Hamiltonian described by the third parameter `evolutionGeneratorEnd`. The fourth parameter `timeDependentSimulationAlgorithm` is where one makes the choice of simulation algorithm. Note that if `interpolationTime` is long enough, an initial ground state remains an instantaneous ground state of the Hamiltonian over the entire duration of time-dependent simulation, and thus ends in the ground state of the end Hamiltonian.

We also define a helpful operation that automatically performs all steps of a typical quantum chemistry experiment. For instance we have the following, which returns an energy estimate of the state produced by adiabatic state preparation:

```
operation EstimateAdiabaticStateEnergy(
    nQubits : Int,
    statePrepUnitary : (Qubit[] => Unit),
    adiabaticUnitary : (Qubit[] => Unit),
    qpeUnitary: (Qubit[] => Unit is Adj + Ctl),
    phaseEstAlgorithm : ((DiscreteOracle, Qubit[]) => Double))
: Double {
    ...
}
```

`nQubits` is the number of qubits used to encode the initial quantum state. `statePrepUnitary` prepares the start state from the computational basis  $\ket{0\cdots 0}$ . `adiabaticUnitary` is the unitary operation that implements adiabatic state preparation, such as produced by the `InterpolatedEvolution` function. `qpeUnitary` is the unitary operation that is used to perform phase estimation on the resulting quantum state. `phaseEstAlgorithm` is our choice of phase estimation algorithm.

#### TIP

Applications of adiabatic state preparation are covered in the samples. For the Ising model using a manual implementation of adiabatic state preparation versus using the `AdiabaticEvolution` function, please see the [AdiabaticIsing sample](#). For phase estimation and adiabatic state preparation in the Ising model, please see the [IsingPhaseEstimation sample](#).

**TIP**

The simulation of molecular Hydrogen is an interesting and brief sample. The model and experimental results reported in O'Malley et. al. only requires Pauli matrices and takes the form  $\hat{H} = g_0 I_0 I_1 + g_1 Z_0 + g_2 Z_1 + g_3 Z_0 - g_4 Y_0 Y_1 + g_5 X_0 X_1$ . This is an effective Hamiltonian only requiring only 2 qubits, where the constants  $g$  are computed from the distance  $R$  between the two Hydrogen atoms. Using canon functions, the Paulis are converted to unitaries and then evolved over short periods of time using the Trotter-Suzuki decomposition. A good approximation to the  $H_2$  ground state can be created without using adiabatic state preparation, and so the ground state energy may be found directly by utilizing phase estimation from the canon.

## Shor's algorithm

Shor's algorithm remains one of the most significant developments in quantum computing because it showed that quantum computers could be used to solve important, currently classically intractable problems. Shor's algorithm provides a fast way to factor large numbers using a quantum computer, a problem called *factoring*. The security of many present-day cryptosystems is based on the assumption that no fast algorithm exists for factoring. Thus Shor's algorithm has had a profound impact on how we think about security in a post-quantum world.

Shor's algorithm can be thought of as a hybrid algorithm. The quantum computer is used to perform a computationally hard task known as period finding. The results from period finding are then classically processed to estimate the factors. We review these two steps below.

### Period finding

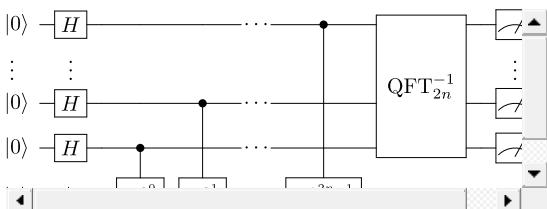
Having seen how the quantum Fourier transform and phase estimation work (see [Quantum algorithms](#)), we can use these tools to solve a classically hard computational problem called *period finding*. In the next section, we will see how to apply period finding to factoring.

Given two integers  $a$  and  $N$ , where  $a < N$ , the goal of period finding, also called order finding, is to find the *order*  $r$  of  $a$  modulo  $N$ , where  $r$  is defined to be the least positive integer such that  $a^r \equiv 1 \pmod{N}$ .

To find the order using a quantum computer, we can use the phase estimation algorithm applied to the following unitary operator  $U_a$ :  $U_a |x\rangle \equiv |ax \pmod{N}\rangle$ . The eigenvectors of  $U_a$  are for integer  $s$  and  $0 \leq s \leq r-1$ ,  $|x_s\rangle \equiv 1/\sqrt{r} \sum_{k=0}^{r-1} e^{-2\pi i sk/r} |x\rangle$ . These are *eigenstates* of  $U_a$ . The eigenvalues of  $U_a$  are  $U_a |x_s\rangle = e^{2\pi i s / r} |x_s\rangle$ .

Phase estimation thus outputs the eigenvalues  $e^{2\pi i s / r}$  from which  $r$  can be learned efficiently using [continued fractions](#) from  $s / r$ .

The circuit diagram for quantum period finding is:



Here  $2n$  qubits are initialized to  $|\psi_0\rangle$  and  $n$  qubits are initialized to  $|\psi_1\rangle$ . The reader again may wonder why the quantum register to hold the eigenstates is initialized to  $|\psi_1\rangle$ . As one does not know the order  $r$  in advance, we cannot actually prepare  $|\psi_s\rangle$  states directly. Luckily, it turns out that  $\sum_{s=0}^{r-1} |\psi_s\rangle = |\psi_1\rangle$ . We don't need to actually prepare  $|\psi_s\rangle$ ! We can just prepare a quantum register of  $n$  qubits in state  $|\psi_1\rangle$ .

The circuit contains the QFT and several controlled gates. The QFT gate has been described [previously](#). The controlled-\$U\_a\$ gate maps \$|ket{x}\rangle\$ to \$|ket{(ax) \text{ mod } N}\rangle\$ if the control qubit is \$|ket{1}\rangle\$, and maps \$|ket{x}\rangle\$ to \$|ket{x}\rangle\$ otherwise.

To achieve \$(a^n) \text{ mod } N\$, we can simply apply controlled-\$U\_{a^n}\$, where we calculate \$a^n \text{ mod } N\$ classically to plug into the quantum circuit.

The circuits to achieve such modular arithmetic have been described in the [quantum arithmetic documentation](#), specifically we require a modular exponentiation circuit to implement the controlled-\$U\_{a^i}\$ operations.

While the circuit above corresponds to [Quantum Phase Estimation](#) and explicitly enables order finding, we can reduce the number of qubits required. We can either follow Beauregard's method for order finding as described [on Page 8 of arXiv:quant-ph/0205095v3](#), or use one of the phase estimation routines available in Microsoft.Quantum.Characterization. For example, [Robust Phase Estimation](#) also uses one extra qubit.

## Factoring

The goal of factoring is to determine the two prime factors of integer \$N\$, where \$N\$ is an \$n\$-bit number. Factoring consists of the steps described below. The steps are split into three parts: a classical preprocessing routine (1-4); a quantum computing routine to find the order of \$a \text{ mod } N\$ (5); and a classical postprocessing routine to derive the prime factors from the order (6-9).

The classical preprocessing routine consists of the following steps:

1. If \$N\$ is even, return the prime factor \$2\$.
2. If \$N=p^q\$ for \$p \geq 1, q \geq 2\$, return the prime factor \$p\$. This step is performed classically.
3. Choose a random number \$a\$ such that \$1 < a < N-1\$.
4. If \$\text{gcd}(a, N) > 1\$, return the prime factor \$\text{gcd}(a, N)\$. This step is computed using Euclid's algorithm. If no prime factor has been returned, we proceed to the quantum routine:
5. Call the quantum period finding algorithm to calculate the order \$r\$ of \$a \text{ mod } N\$. Use \$r\$ in the classical postprocessing routine to determine the prime factors:
6. If \$r\$ is odd, go back to preprocessing step (3).
7. If \$r\$ is even and \$a^{r/2} = -1 \text{ mod } N\$, go back to preprocessing step (3).
8. If \$\text{gcd}(a^{r/2} + 1, N)\$ is a non-trivial factor of \$N\$, return \$\text{gcd}(a^{r/2} + 1, N)\$.
9. If \$\text{gcd}(a^{r/2} - 1, N)\$ is a non-trivial factor of \$N\$, return \$\text{gcd}(a^{r/2} - 1, N)\$.

The factoring algorithm is probabilistic: it can be shown that with probability at least one half that \$r\$ will be even and \$a^{r/2} \neq -1 \text{ mod } N\$, thus producing a prime factor. (See [Shor's original paper](#) for details, or one of the *Basic quantum computing* texts in [For more information](#)). If a prime factor is not returned, then we simply repeat the algorithm from step (1). After \$n\$ tries, the probability that every attempt has failed is at most \$2^{-n}\$. Thus after repeating the algorithm a small number of times success is virtually assured.

# Licensing

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Quantum Development Kit is provided with an extensive collection of open-source functions and operations, licensed under the [MIT license](#). The portions of the standard library that are portable across target machines can be obtained from the [Microsoft/QuantumLibraries](#) repository on GitHub, along with other libraries such as and other libraries, such as the [quantum chemistry library](#).

Microsoft's Quantum Development Kit also provides specialized library functions and operations that are licensed under a [Microsoft Research license](#). These can be obtained from the [Microsoft/Quantum-NC](#) repository on GitHub.

There are also many samples that explain and illustrate the use of functions and operations from the standard library and other libraries. These samples are licensed under the [MIT license](#). The samples can be obtained from the [Microsoft/Quantum](#) repository on GitHub.

## Contributing

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit [CLA](#).

When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (for example, label or comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repos using our CLA.

This project has adopted the [Microsoft Open Source Code of Conduct](#). For more information see the [Code of Conduct FAQ](#) or contact [opencode@microsoft.com](mailto:opencode@microsoft.com) with any additional questions or comments.

# Using Additional Q# Libraries

5/27/2021 • 2 minutes to read • [Edit Online](#)

The Quantum Development Kit (QDK) provides additional domain-specific functionality through *NuGet packages* that can be added to your Q# projects.

Q# LIBRARY	NUGET PACKAGE	NOTES
Q# standard library	<a href="#">Microsoft.Quantum.Standard</a>	Included by default
Quantum chemistry library	<a href="#">Microsoft.Quantum.Chemistry</a>	
Quantum numerics library	<a href="#">Microsoft.Quantum.Numerics</a>	
Quantum machine learning library	<a href="#">Microsoft.Quantum.MachineLearning</a>	

Once you have installed the Quantum Development Kit for use with your preferred environment and host language, you can easily add libraries to individual Q# projects without any further installation.

## NOTE

Some Q# libraries may work well with additional tools that work alongside your Q# programs, or that integrate with your host applications. For example, the [chemistry library installation instructions](#) describe how to use the [Microsoft.Quantum.Chemistry package](#) together with the NWChem computational chemistry platform, and how to install the `qdk-chem` command-line tools for working with quantum chemistry data.

- [Q# applications or .NET interoperability](#)
- [IQ# Notebooks](#)
- [Python interoperability](#)

**Command prompt or Visual Studio Code:** Using the command prompt on its own or from within Visual Studio Code, you can use the `dotnet` command to add a NuGet package reference to your project. For example, to add the [Microsoft.Quantum.Numerics](#) package, run the following command:

```
dotnet add package Microsoft.Quantum.Numerics
```

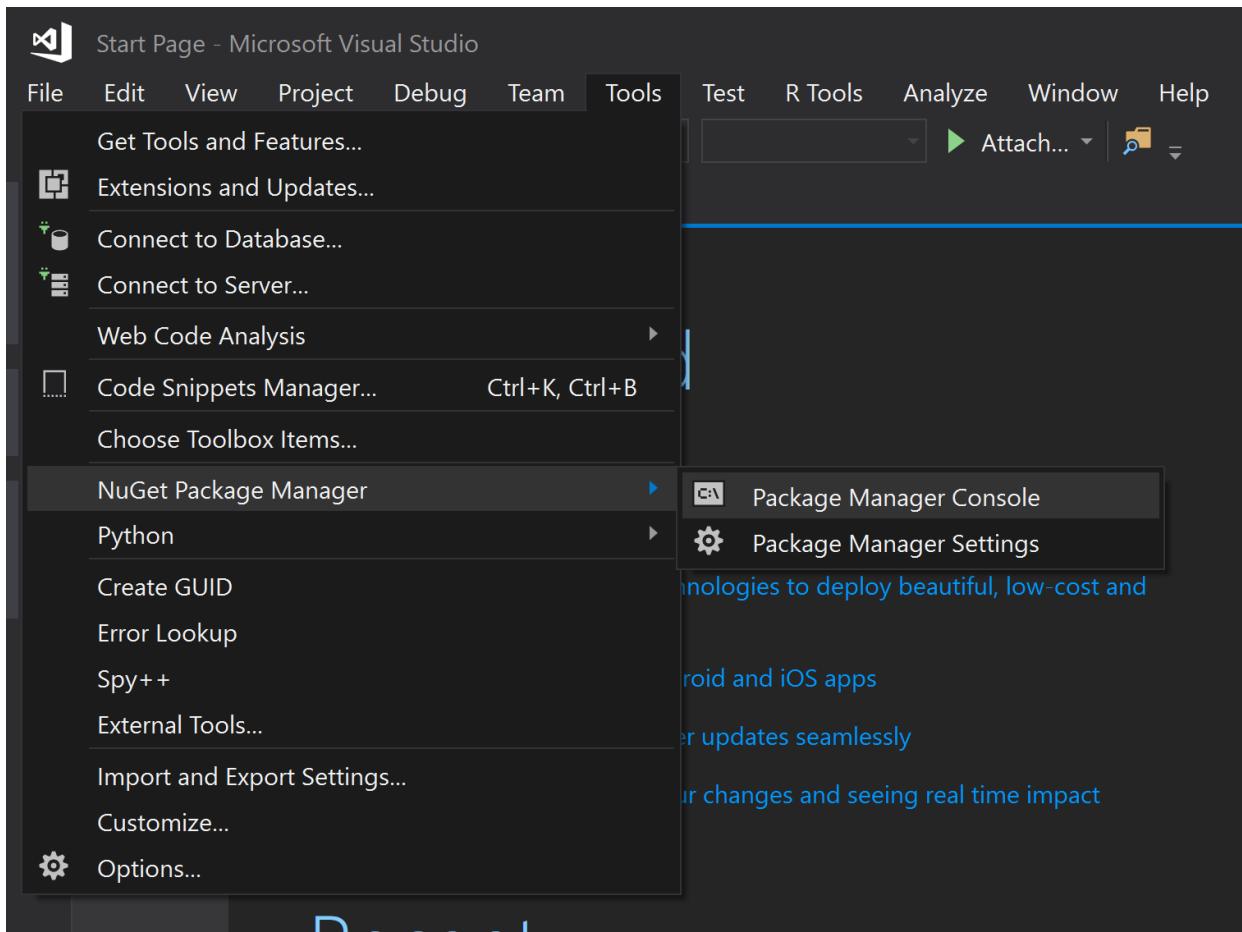
**Visual Studio:** If you are using Visual Studio 2019 or later, you can add additional Q# packages using the NuGet Package Manager. To load a package:

1. With a project open in Visual Studio, select **Manage NuGet Packages...** from the **Project** menu.
2. Click **Browse**, select **Include prerelease** and search for the package name "Microsoft.Quantum.Numerics". This will list the packages available for download.
3. Hover over **Microsoft.Quantum.Numerics** and click the downward-pointing arrow to the right of the version number to install the numerics package.

For more details, see the [Package Manager UI guide](#).

Alternatively, you can use the Package Manager Console to add the numerics library to your project via the

command line interface.



From the Package Manager Console, run the following:

```
Install-Package Microsoft.Quantum.Numerics
```

For more details, see the [Package Manager Console guide](#).

# Introduction to the Quantum Chemistry Library

3/9/2021 • 2 minutes to read • [Edit Online](#)

Simulation of physical systems has long played a central role in quantum computing. This is because quantum dynamics are widely believed to be intractable to simulate on classical computers, meaning that the complexity of simulating the system scales exponentially with the size of the quantum system in question. Simulating problems in chemistry and material science remains perhaps the most evocative application of quantum computing and would allow us to probe chemical reaction mechanisms that hitherto were beyond our ability to measure or simulate. It would also allow us to simulate correlated electronic materials such as high-temperature superconductors. The list of applications in this space is vast.

The purpose of this documentation is to provide a concise introduction to simulating electronic structure problems on quantum computers in order to help the reader understand the role that many aspects of the Hamiltonian simulation library play within the space. We begin with a brief introduction to quantum mechanics and then proceed to discuss how electronic systems are modeled in it. We will then discuss how such quantum dynamics can be emulated using a quantum computer. Once this is complete we will discuss two methods used to compile the quantum dynamics to sequences of elementary gates: Trotter-Suzuki decompositions and qubitization.

# Chemistry library installation

3/5/2021 • 4 minutes to read • [Edit Online](#)

The [MolecularHydrogen sample](#) uses molecular input data that is manually configured. While this is fine for small examples, quantum chemistry at scale requires Hamiltonians with millions or billions of terms. Such Hamiltonians, generated by scalable computational chemistry packages, are too large to import by hand.

The quantum chemistry library for the Quantum Development Kit is designed to work well with computational chemistry packages, most notably the [NWChem](#) computational chemistry platform developed by the Environmental Molecular Sciences Laboratory (EMSL) at Pacific Northwest National Laboratory. In particular, the [Microsoft.Quantum.Chemistry package](#) provides tools for loading instances of quantum chemistry simulation problems represented in the [Broombridge schema](#), also supported for export by recent versions of NWChem.

The Quantum Development Kit chemistry library also provides a command-line tool, `qdk-chem`, for converting between legacy formats and [Broombridge](#).

This section details how to use the Quantum Development Kit with either NWChem and Broombridge, or legacy formats and `qdk-chem`.

## Using the Quantum Development Kit with NWChem

To get up and running using NWChem together with the Quantum Development Kit, use one of the following methods:

- Get started using existing Broombridge files provided with the samples at [IntegralData/YAML](#).
- Use the [EMSL Arrows Builder for the Quantum Development Kit](#) which is a web-based frontend to NWChem, to generate new Broombridge-formated molecular input files.
- Use the [Docker image](#) provided by PNNL to run NWChem, or
- [Compile NWChem](#) for your platform.

See [End-to-end with NWChem](#) for more information on how to work with NWChem to chemical models to analyze with the Quantum Developmen Kit chemistry library.

### Getting started using Broombridge files provided with the samples

The [IntegralData/YAML](#) folder in the Quantum Development Kit Samples repository contains Broombridge-formated molecule data files.

As a simple example, use the chemistry library sample, [GetGateCount](#) to load the Hamiltonian from one of Broombridge files and perform gate estimates of quantum simulation algorithms:

```
cd Quantum/Chemistry/GetGateCount  
dotnet run -- --path=../IntegralData/YAML/h2.yaml --format=YAML
```

See [Loading a Hamiltonian from file](#) for more information on how to input molecules represented by the Broombridge schema.

See [Obtaining resource counts](#) for more information on resource estimation.

### Getting started using the EMSL Arrows Builder

EMSL Arrows is a tool that uses NWChem and chemical computational databases to generate molecule data.

[EMSL Arrows Builder for the Quantum Development Kit](#) allows you to enter your model using multiple molecular model builders and generate the Broombridge datafile to be used by the Quantum Development Kit.

From the EMSL page, click the ['Instructions'] tab, and follow the ['Simple Examples'] instructions to generate Broombridge files. Then try running the ['GetGateCount'] to see the quantum resource estimates for these molecules.

### Installing NWChem from source

Full instructions on how to install NWChem from source [are provided by PNNL](#).

#### TIP

If you wish to use NWChem from Windows 10, the Windows Subsystem for Linux is a great option. Once you have installed [Ubuntu 18.04 LTS for Windows](#), run `ubuntu18.04` from your favorite terminal and follow the instructions above to install NWChem from source.

Once you have compiled NWChem from source, you can run the `yaml_driver` script provided with NWChem to quickly produce Broombridge instances from NWChem input decks:

```
$NWCHEM_TOP/contrib/quasar/yaml_driver input.nw
```

This command will create a new `input.yaml` file in the Broombridge format within your current directory.

### Using NWChem with Docker

Pre-built images for using NWChem are available cross-platform via [Docker Hub](#). To get started, follow the Docker installation instructions for your platform:

- [Install Docker for Ubuntu](#)
- [Install Docker for macOS](#)
- [Install Docker for Windows 10](#)

#### TIP

If you are using Docker for Windows, you will need to share the drive containing your temporary directory (usually this is the `c:\` drive) with the Docker daemon. See the [Docker documentation](#) for more details.

Once you have Docker installed, you can either use the PowerShell module provided with the Quantum Development Kit samples to run the image, or you can run the image manually. We detail using PowerShell here; if you would like to use the Docker image manually, please see the [documentation provided with the image](#).

### Running NWChem through PowerShell Core

To use the NWChem Docker image with the Quantum Development Kit, we provide a small PowerShell module that handles moving files in and out of NWChem for you. If you don't already have PowerShell Core installed, you can download it cross-platform from the [PowerShell repository on GitHub](#).

#### NOTE

PowerShell Core is also used for some samples to demonstrate interoperability with data science and business analytics workflows.

Once you have PowerShell Core installed, import `InvokeNWChem.psm1` to make NWChem commands available in your current session:

```
cd Quantum/utilities/  
Import-Module ./InvokeNWChem.psm1
```

This will make the `Convert-NWChemToBroombridge` command available in your current PowerShell session. To download any needed images from Docker and use them to run NWChem input decks and capture output to Broombridge, run the following:

```
Convert-NWChemToBroombridge ./input.nw
```

This will create a Broombridge file by running NWChem on `input.nw`. By default, the Broombridge output will have the same name and path as the input deck, with the `.nw` extension replaced by `.yaml`. This can be overridden by using the `-DestinationPath` parameter to `Convert-NWChemToBroombridge`. More information can be obtained by using PowerShell's built-in help functionality:

```
Convert-NWChemToBroombridge -?  
Get-Help Convert-NWChemToBroombridge -Full
```

## Using the Quantum Development Kit with `qdk-chem`

To install `qdk-chem`, you can use the .NET Core SDK at the command line:

```
dotnet tool install --global Microsoft.Quantum.Chemistry.Tools
```

Once you have installed `qdk-chem`, you can use the `--help` option to get more details about the functionality offered by the `qdk-chem` tool.

To convert to and from Broombridge, you can use the `qdk-chem convert` command:

```
qdk-chem convert --from fcidump --to broombridge data.fcidump --out data.yml
```

The `qdk-chem convert` command can also accept its data from standard input, and can write to standard output; this is especially useful within scripts and for integrating with tools that export to legacy formats. For example, in Bash:

```
cat data.fcidump | qdk-convert --from fcidump --to broombridge - > data.yml
```

# Quantum Dynamics

3/5/2021 • 4 minutes to read • [Edit Online](#)

Quantum mechanics is largely the study of quantum dynamics, which seeks to understand how an initial quantum state  $|\psi(0)\rangle$  varies over time (see the [conceptual docs](#) on quantum computing for more info on Dirac notation). Specifically, given this initial condition a quantum state, an evolution time and a specification of the quantum dynamical system, a quantum state  $|\psi(t)\rangle$  is sought. Before proceeding to explain quantum dynamics, it is useful to take a step back and think about classical dynamics since it provides insights into how quantum mechanics is really not that different from classical dynamics.

In classical dynamics, we know from Newton's second law of motion that the position of a particle evolves according to  $F(x,t) = ma = m\frac{d^2x}{dt^2}$ , where  $F(x,t)$  is the force,  $m$  is the mass and  $a$  is the acceleration. Then, given an initial position  $x(0)$ , evolution time  $t$ , and description of the forces that act on the particle, we can then find  $x(t)$  by solving the differential equation given by Newton's equations for  $x(t)$ . Specifying the forces in this way is a bit of a pain. So we often express the forces in terms of the potential energy of the system, which gives us  $-\partial_x V(x,t) = m \frac{d^2x}{dt^2}$ . Thus, for a particle, the dynamics of the system are specified only by the potential energy function, the particle mass, and the evolution time.

A broader language is often introduced for classical dynamics that goes beyond  $F=ma$ . One formulation, which is particularly useful in quantum mechanics, is Hamiltonian mechanics. In Hamiltonian mechanics, the total energy of a system and the (generalized) positions and momenta give all the information needed to describe the motion of an arbitrary classical object. Specifically, let  $f(x,p,t)$  be some function of the generalized positions  $x$  and momenta  $p$  of a system and let  $H(x,p,t)$  be the Hamiltonian function. For example, if we take  $f(x,p,t) = x(t)$  and  $H(x,p,t) = p^2/2m - V(x,t)$ , then we would recover the above case of Newtonian dynamics. In generality, we then have that  $\begin{aligned} \frac{df}{dt} &= \partial_t f - (\partial_x H) \partial_p f + (\partial_p H) \partial_x f \\ &\stackrel{\text{defeq}}{=} \partial_t f + \{f, H\}. \end{aligned}$  Here  $\{f, H\}$  is called the [Poisson bracket](#) and appears ubiquitously in classical dynamics because of the central role it plays in defining dynamics.

Quantum dynamics can be described using exactly the same language. The Hamiltonian, or total energy, completely specifies the dynamics of any closed quantum system. There are, however, some substantial differences between the two theories. In classical mechanics  $x$  and  $p$  are just numbers, whereas in quantum mechanics they are not. Indeed, they do not even commute:  $xp \neq px$ .

The right mathematical concept to describe these non-commuting objects is an operator, which in cases where  $x$  and  $p$  can only take a discrete set of values coincides with the concept of a matrix. Thus for simplicity, we will assume that our quantum system is finite so that it can be described using [vectors and matrices](#). We further require that these matrices be Hermitian (meaning that the conjugate transpose of the matrix is the same as the original matrix). This guarantees that the eigenvalues of the matrices are real-valued; a condition which we impose to ensure that when we measure a quantity like position that we don't get back out an imaginary number.

Just as the analogues of position and momentum in quantum mechanics need to be replaced by operators, the Hamiltonian function needs to be similarly replaced by an operator. For example, for a particle in free space we have that  $H(x,p) = p^2/2m$  whereas in quantum mechanics the Hamiltonian operator  $\hat{H}$  is  $\hat{H} = \hat{p}^2/2m$  where  $\hat{p}$  is the momentum operator. From this perspective, going from classical to quantum dynamics merely involves replacing the variables used in ordinary dynamics with operators. Once we have constructed the Hamiltonian operator by translating the ordinary classical Hamiltonian over to quantum language, we can express the dynamics of an arbitrary quantum mechanical quantity (for example, quantum mechanical operator)  $\hat{f}(t)$  via  $\begin{aligned} \frac{d}{dt} \hat{f}(t) &= \partial_t \hat{f} + [\hat{f}, \hat{H}] \end{aligned}$  where  $[\hat{f}, \hat{H}] = \hat{f}\hat{H} - \hat{H}\hat{f}$  is known as the commutator. This expression is exactly like the classical

expression given above with the difference that the Poisson bracket  $\{f, H\}$  being replaced with the commutator between  $f$  and  $H$ . This process of taking a classical Hamiltonian and using it to find a quantum Hamiltonian is known in quantum jargon as canonical quantization.

What operators  $f$  are we most interested in? The answer to this depends on the problem that we want to solve. Perhaps the most useful quantity to find is the quantum state operator, which as discussed in the earlier conceptual documentation can be used to extract everything that we would like learn about the dynamics. After doing this (and simplifying the result to the case where one has a pure state), the Schrödinger equation for the quantum state is found  $i\partial_t |\psi(t)\rangle = \hat{H}(t) |\psi(t)\rangle$ .

This equation, though perhaps less intuitive than that given above, yields perhaps the simplest expression for understanding how to simulate quantum dynamics on either a quantum or classical computer. This is because the solution to the differential equation can be expressed in the following form (for the case where the Hamiltonian is constant in  $t$ )  $|\psi(t)\rangle = e^{-iHt} |\psi(0)\rangle$ . Here  $e^{-iHt}$  is a unitary matrix. This means that there exists a quantum circuit that can be designed to perform it because quantum computers can closely approximate any unitary matrix. This act of finding a quantum circuit to implement the quantum time evolution operator  $e^{-iHt}$  is what is often called quantum simulation, or in particular dynamical quantum simulation.

# Quantum Models for Electronic Systems

3/5/2021 • 3 minutes to read • [Edit Online](#)

In order to simulate electronic systems we need to first begin by specifying the Hamiltonian, which can be found by the canonical quantization procedure described above. Specifically, for  $N_e$  electrons with momenta  $p_i$  (in three dimensions) and mass  $m_e$  and position vectors  $x_i$  along with nuclei with charges  $Z_k e$  at positions  $y_k$ , the Hamiltonian operator reads 
$$\hat{H} = \sum_{i=1}^{N_e} \frac{\hat{p}_i^2}{2m_e} + \frac{1}{2} \sum_{i \neq j} \frac{e^2}{|x_i - x_j|} - \sum_{i,k} \frac{Z_k e^2}{|x_i - y_k|} + \frac{1}{2} \sum_{k \neq k'} \frac{Z_k Z_{k'} e^2}{|y_k - y_{k'}|}. \quad \text{label{eq:Ham}}$$
 The momenta operators  $\hat{p}_i^2$  can be viewed in real space as Laplacian operators, for example,  $\hat{p}_i^2 = -\partial_{x_i}^2 - \partial_{y_i}^2 - \partial_{z_i}^2$ . Here we have made the simplifying assumption that the nuclei are at rest for the molecule. This is known as the Born-Oppenheimer approximation and it tends to be valid for the low-energy energy spectrum of  $\hat{H}$  since the electron mass is about  $1/1836$  the mass of a proton. This Hamiltonian operator can be easily found by writing out the energy for a system of  $N_e$  electrons and applying the canonical quantization process described in [Quantum Dynamics](#).

In order to construct the unitary matrix representation for  $e^{-i\hat{H}t}$  we need to represent the operator  $\hat{H}$  as a matrix. For this, we need to choose a coordinate system or basis to represent the problem in. For example, if  $\psi_j$  are a set of orthogonal basis functions for the  $N_e$  electrons then we can define the matrix

$$\begin{aligned} H_{ij} = & \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \psi_i^*(x_1) \hat{H} \psi_j(x_2) dx_1 dx_2 \\ & \ddots \end{aligned} \quad \text{label{eq:discreteHam}}$$

While in principle the operator  $\hat{H}$  is unbounded and does not act on a finite dimensional space, the matrix with elements  $H_{ij}$  above does. This means that errors are incurred by picking too small of a basis set; however, picking a large basis can make simulating the chemical dynamics impractical. For this reason, choosing a basis that can concisely represent the problem is vital for solving the electronic structure problem.

There are many appropriate bases that can be used and the choice of a good basis to fit the problem is much of the art of quantum chemistry. Perhaps the simplest such basis sets are Slater-Type-Orbitals (STO) which are (orthogonalized) solutions to the Schrödinger equation (for example, eigenfunctions of  $\hat{H}$ ) for hydrogen-like atoms. Other basis sets, such as plane-waves or real-space orbitals, can be used and for more detail we refer the curious reader to the standard text '[Molecular Electronic-Structure Theory](#)' by Helgaker.

While the states used in the above model may seem arbitrary, quantum mechanics places restrictions on the states that can be found in nature. In particular, all valid electronic quantum states must be anti-symmetric under exchange of electron labels. That is to say if  $\psi(x_1, x_2)$  were the wave function for the joint quantum state of two electrons then we must have that  $\psi(x_1, x_2) = -\psi(x_2, x_1)$ . The Pauli exclusion principle which forbids two electrons to ever be in the same quantum state is, fascinatingly, a direct consequence of this law as can be intuited from the fact that if we swap two electrons located at the same position  $\psi(x_1, x_1) \mapsto \psi(x_1, x_1) \neq -\psi(x_1, x_1)$  unless  $\psi(x_1, x_1) = 0$ . Thus the initial states must be chosen to obey this anti-symmetry property and in turn never have two electrons in the same state at the same time. This is crucial for electronic structure because it forbids multiple electrons from being in the same state, and in turn allows quantum computers to use a single quantum bit to store the number of electrons in a given quantum state.

While quantum mechanics can be simulated on a quantum computer by discretizing these states, most work in the field has eschewed this approach because it requires many qubits to store the states and needs a complicated state preparation procedure to prepare an anti-symmetric initial state. Fortunately though, these problems can be sidestepped by viewing the simulation problem from a different perspective.

# Second Quantization

6/30/2021 • 13 minutes to read • [Edit Online](#)

Second quantization looks at the problem of electronic structure through a different lens. Rather than assigning each of the  $N_e$  electrons to a specific state (or orbital), second quantization tracks each orbital and stores whether there is an electron present in each of them and at the same time automatically ensures symmetry properties of the corresponding wave function. This is important because it allows quantum chemistry models to be specified without having to worry about anti-symmetrizing the input state (as is required for fermions) and also because second quantization allows such models to be simulated using small quantum computers.

As an example of second quantization in action, let's assume that  $\psi_0 \cdots \psi_{N-1}$  are an orthonormal set of spatial orbitals. These orbitals are chosen to represent the system as accurately as possible within the finite basis set considered. A common example of such orbitals are atomic orbitals which form an eigenbasis for the hydrogen atom. Because electrons have two spin states, two electrons can be crammed into each such spatial orbital. That is to say, the valid basis states are of the form  $\psi_0\downarrow, \dots, \psi_{N-1}\downarrow, \psi_0\uparrow, \dots, \psi_{N-1}\uparrow$  where  $\downarrow$  and  $\uparrow$  are labels that specify the two eigenstates of the spin degree of freedom. This combined index of  $(j, \sigma)$  for  $\sigma \in \{\downarrow, \uparrow\}$  is called a spin-orbital because it stores both the spatial as well as the spin degree of freedom. In the chemistry library, spin-orbitals are stored in a `SpinOrbital` data structure, and are created as follows.

```
// The code snippets in this section require the following namespaces.  
// Make sure to include these at the top of your file or namespace.  
using Microsoft.Quantum.Chemistry;  
using Microsoft.Quantum.Chemistry.OrbitalIntegrals;
```

```
// First, we assign an orbital index, say `5`. Note that we use 0-indexing,  
// so this is the 6th orbital.  
var orbitalIdx = 5;  
  
// Second, we assign a spin index, say `Spin.u` for spin up or `Spin.d` for spin down.  
var spin = Spin.d;  
  
// the spin-orbital (5, ↓) is then  
var spinOrbital = new SpinOrbital(orbitalIdx, spin);  
  
// A tuple `(int, Spin)` is also implicitly recognized as a spin-orbital.  
(int, Spin) tuple = (orbitalIdx, spin);  
  
// We explicitly specify the type of `spinOrbital1` to demonstrate  
// the implicit cast to `SpinOrbital`.  
SpinOrbital spinOrbital1 = tuple;
```

This means that we can formally think of the basis for both the spin and spatial part of the wave function as  $\psi_0 \cdots \psi_{2N-1}$  where each of the indices now is an enumeration of a  $(j, \sigma)$ . One possible enumeration is  $g(j, \sigma) = j + N\sigma$ . Another possible enumeration is  $h(j, \sigma) = 2^j + \sigma$ . The quantum chemistry library can use these conventions, and the spin-orbitals in such an encoding can be instantiated as follows.

```

// Let us use the spin orbital created in the previous snippet.
var spinOrbital = new SpinOrbital(5, Spin.d);

// Let us set the total number of orbitals to be say, `7`.
var nOrbitals = 7;

// This converts a spin-orbital index to a unique integer, in this case `12`,
// using the formula `g(j,σ)` .
var integerIndexHalfUp = spinOrbital.ToInt(IndexConvention.HalfUp);

// This converts a spin-orbital index to a unique integer, in this case `11` ,
// using the formula `h(j,σ)` .
var integerIndexUpDown = spinOrbital.ToInt(IndexConvention.UpDown);

// The default conversion uses the formula `h(j,σ)` , in this case `11` .
var integerIndexDefault = spinOrbital.ToInt();

```

For fermionic systems, the Pauli exclusion principle prevents more than one electron from being present in any spin-orbital at the same time. This means that we can write the two legal states for  $\psi_1$  as

$$\begin{aligned} \psi_1 \rightarrow \begin{cases} |0\rangle_1 & \text{if } \psi_1 \text{ is not occupied,} \\ |1\rangle_1 & \text{if } \psi_1 \text{ is occupied.} \end{cases} \end{aligned}$$

This encoding is great for quantum computers because it means that we can store the electronic occupation as a single quantum bit.

The occupation states for the  $2N$  spin orbitals can similarly be stored in  $2N$  qubits. As an example, if  $N=2$  then the state

$$|\psi\rangle = |\psi_1\rangle_1 |\psi_2\rangle_2 |\psi_3\rangle_3 |\psi_4\rangle_4$$

would correspond to spin orbitals  $\psi_1$  and  $\psi_2$  being occupied with the remainder empty. Similarly, the state

$$|\psi\rangle = |\psi_1\rangle_1 |\psi_2\rangle_2 |\psi_3\rangle_3 |\psi_4\rangle_4 \dots |\psi_N\rangle_N$$

has no electrons and is known as the 'vacuum state'.

A beautiful side-effect of second quantization is that we no longer have to explicitly keep track of the anti-symmetry of the quantum state. This is because, as we will see, the anti-symmetry of the state represents itself instead through the anti-commutation rules of the operators that create and destroy electronic occupations of a spin orbital.

## Fermionic operators

The two fundamental operators that act on the second-quantized basis vectors are known as creation and annihilation operators. These operators insert or destroy electrons at a particular location. These are denoted  $a^\dagger_j$  and  $a_j$  respectively.

For example,

$$a^\dagger_1 |0\rangle_1 = |1\rangle_1 \quad a^\dagger_1 |1\rangle_1 = 0 \quad a_1 |0\rangle_1 = 0 \quad a_1 |1\rangle_1 = |0\rangle_1$$

Note that here  $a^\dagger_1 |1\rangle_1 = 0$  and  $a_1 |0\rangle_1 = 0$  yield the zero-vector not  $|0\rangle_1$ . Such operators are therefore neither Hermitian nor unitary. We represent general creation and annihilation operators using the [LadderOperator](#) type. For instance, a single creation operator is represented as follow.

```

// The code snippets in this section require the following namespaces.
// Make sure to include these at the top of your file or namespace.
using Microsoft.Quantum.Chemistry;
using Microsoft.Quantum.Chemistry.OrbitalIntegrals;
using Microsoft.Quantum.Chemistry.LadderOperators;
using Microsoft.Quantum.Chemistry.Fermion;
// We load this namespace for convenience methods for manipulating arrays.
using System.Linq;

```

```

// Let us use the spin orbital created in the previous snippet.
var spinOrbitalInteger = new SpinOrbital(5, Spin.d).ToInt();

// We specify either a creation or annihilation operator using
// the enumerable type `RaisingLowering.u` or `RaisingLowering.d`
// respectively;
var creationEnum = RaisingLowering.u;

// The type representing a creation operator is then initialized
// as follows. Here, we index these operators with integers.
// Hence we initialize the generic ladder operator with an
// integer index type.
var ladderOperator0 = new LadderOperator<int>(creationEnum, spinOrbitalInteger);

// An alternate constructor for a LadderOperator instead uses
// a tuple.
var ladderOperator1 = new LadderOperator<int>((creationEnum, spinOrbitalInteger));

```

Also using such operators we can express

$$\$ \$ \langle \text{ket}\{0\} \rangle \langle \text{ket}\{1\} \rangle \langle \text{ket}\{1\} \rangle \langle \text{ket}\{0\} \rangle = a^\dagger \text{dagger}_1 a^\dagger \text{dagger}_2 \langle \text{ket}\{0\} \rangle^{\otimes 4}. \$ \$$$

This sequence of operators would be constructed within the Hamiltonian simulation library using C# code that is similar to the single-spin orbital case considered above:

```

// Let us initialize an array of tuples representing the
// desired sequence of creation operators.
var indices = new[] { (RaisingLowering.u, 1), (RaisingLowering.u, 2) };

// We can convert this array of tuples to a sequence of ladder
// operators using the `ToLadderSequence()` methods.
var ladderSequences = indices.ToLadderSequence();

// Sequences of ladder operators are quite general. For instance,
// they could be bosonic operators, instead of fermionic operators.
// We specialize them by constructing a `FermionTerm` representing
// a fermion creation operator on the index `2` followed by `1`.
var fermionTerm = new FermionTerm(ladderSequences);

```

For a system of  $k$  Fermions, in second quantization the action of the creation operator  $a^\dagger_i$  is given by

$$\$ \$ a^\dagger_i \langle \text{ket}\{n_1, n_2, \dots, n_k\} \rangle = (-1)^{S_i} \langle \text{ket}\{n_1, n_2, \dots, n_k\} \rangle, \$ \$$$

and

$$\$ \$ a^\dagger_i \langle \text{ket}\{n_1, n_2, \dots, n_k\} \rangle = 0, \$ \$$$

where  $S_i = \sum_{j < i} a^\dagger_j a_j$  measures the total number of Fermions that are in the state of a single particle and that have an index  $j < i$ .

A third operator is also often used in second quantized representations. This operator is known as the number

operator and is defined by

```
$$ n_i = a^\dagger a_i. $$
```

This operator counts the occupation of a given spin orbital, which is to say

```
\begin{align} n_i |0\rangle_i &= 0 \text{nonumber} \\ n_i |1\rangle_i &= |1\rangle_i. \end{align}
```

Similar to the above `FermionTerm` examples, this number operator is constructed as follows.

```
// Let us use a new method to compactly create a sequence of ladder
// operators. Note that we have omitted specifying whether the
// operators are raising or lowering. In this case, the first half
// will be raising operators, and the second half will be lowering
// operators.
var indices = new[] { 1, 1 }.ToLadderSequence();

// We now construct a `FermionTerm` representing an annihilation operator
// on the index 1 followed by the creation operator on the index 1.
var fermionTerm0 = new FermionTerm(indices);
```

A subtlety emerges though when using creation or annihilation operators in fermionic systems. We require that any valid quantum state is anti-symmetric under exchange of labels. This means that

```
$$ a^\dagger_2 a^\dagger_1 |0\rangle = -a^\dagger_1 a^\dagger_2 |0\rangle. $$
```

Such operators are said to 'anti-commute' and in general for any  $i, j$  we have that

```
$$ a^\dagger_i a^\dagger_j = -(1 - \delta_{ij}) a^\dagger_j a^\dagger_i, \quad a^\dagger_i a_j = \delta_{ij} - a_j a^\dagger_i. $$
```

Thus the following two `LadderSequence<TIndex>` instances are considered inequivalent

```
// Let us initialize an array of tuples representing the
// desired sequence of creation operators.
var indices = new[] { (RaisingLowering.u, 1), (RaisingLowering.u, 2) };

// We now construct a `LadderSequence` representing a creation operator
// on the index 1 followed by 2, then a term with the reverse ordering.
var ladderSequence = indices.ToLadderSequence();
var ladderSequenceReversed = indices.Reverse().ToLadderSequence();

// The following Boolean is `false`.
var equal = ladderSequence == ladderSequenceReversed;
```

The requirement that each of the creation operators anti-commute means that using a second quantized representation does obviate the challenges faced by the anti-symmetry of Fermions. Instead the challenge re-emerges in our definition of the creation operators.

Using the anti-commutation rules, some `LadderSequence` instances actually correspond to the same sequence of fermionic operators, sometimes up to a minus sign. For instance, consider the Hamiltonian  $a_0^\dagger a_1^\dagger a_1 a_0 = -a_1^\dagger a_0^\dagger a_0 a_1$ . This motivates us to define a canonical ordering for every `FermionTerm`. Any `FermionTerm` is automatically put into canonical order as follows.

```

// We now construct two `FermionTerms` that are equivalent with respect to
// anti-commutation up to a sign change.
var fermionTerm0 = new FermionTerm(new[] { 0, 1, 1, 0 }.ToLadderSequence());
var fermionTerm1 = new FermionTerm(new[] { 1, 0, 1, 0 }.ToLadderSequence());

// The following Boolean is `true`.
var sequenceEqual = fermionTerm0 == fermionTerm1;

// The change in sign is not compared above, but is an internally tracked
// property of `FermionTerm`.
int sign0 = fermionTerm0.Coefficient;
var sign1 = fermionTerm1.Coefficient;

// The following Boolean is `false`.
var signEqual = sign0 == sign1;

```

## Second-Quantized Fermionic Hamiltonian

It is perhaps unsurprising that the Hamiltonian in [Quantum Models for Electronic Systems](#) can be written in terms of creation and annihilation operators. In particular, if  $\psi_{j\sigma}$  are the spin orbitals that form the basis then

$$\hat{H} = \sum_{pq} h_{pq} a^\dagger_p a_q + \frac{1}{2} \sum_{pqrs} h_{pqrs} a^\dagger_p a^\dagger_q a_r a_s + h_{\text{nuc}} \quad (\text{eq:totalHam})$$

where  $h_{\text{nuc}}$  is the nuclear energy (which is a constant under the Born-Oppenheimer approximation) and

$$h_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \psi_p(x_1) \left( -\frac{\nabla^2}{2} + V(x_1) \right) \psi_q(x_1) dx_1 dx_2 \quad (\text{eq:integrals})$$

where  $V(x)$  is the mean-field potential, and

$$h_{pqrs} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \psi_p(x_1) \psi_q(x_2) \left( \frac{1}{|x_1 - x_2|} \right) \psi_r(x_2) \psi_s(x_1) dx_1 dx_2 dx_3 dx_4 \quad (\text{eq:integrals})$$

The terms  $h_{pq}$  are referred to as one-electron integrals because all such terms only involve single electrons and likewise  $h_{pqrs}$  are the two-electron integrals. They are called integrals because computing the values of these coefficients requires an integral. The one electron terms describe the kinetic energy of the individual electrons and their interactions with the electric fields of the nuclei. The two-electron integrals on the other hand describe the interactions between the electrons.

An intuition for what these terms mean can be gleaned from the creation and annihilation operators that comprise each of them. For example,  $h_{pq} a^\dagger_p a_q$  describes the electron hopping from spin orbital  $q\sigma$  to spin orbital  $p\sigma$ . Similarly, the term  $h_{pqrs} a^\dagger_p a^\dagger_q a_r a_s$  (for distinct  $p,q,r,s$ ) describes two electrons in spin orbitals  $r\sigma$  and  $s\sigma$  scattering off of each other and ending up in spin orbitals  $p\sigma$  and  $q\sigma$ . If  $r=q$  and  $p=s$  then  $h_{prrp} a^\dagger_p a^\dagger_q a_r a_p = h_{prrp} n_p n_r$  gives the energy penalty associated with the two electrons being near each other, but does not describe a dynamical process.

We may represent such Hamiltonians using the `FermionHamiltonian` class, which is essentially a list containing all the desired `FermionTerm` instances. As Hamiltonians are Hermitian by definition, we index terms using the more specialized type `HermitianFermionTerm` that also uses Hermitian symmetry when checking whether terms are equivalent.

Let us construct a few illustrative examples. Consider the Hamiltonian  $\hat{H} = a_0^\dagger a_1 + a_1^\dagger a_0$ .

```
// The code snippets in this section require the following namespaces.
// Make sure to include these at the top of your file or namespace.
using Microsoft.Quantum.Chemistry;
using Microsoft.Quantum.Chemistry.LadderOperators;
using Microsoft.Quantum.Chemistry.Fermion;
// We load this namespace for convenience methods for manipulating arrays.
using System.Linq;
```

```
// We create a `FermionHamiltonian` instance to store the fermion terms.
var hamiltonian = new FermionHamiltonian();

// We construct the terms to be added.
var fermionTerm0 = new FermionTerm(new[] { 1, 0 }.ToLadderSequence());
var fermionTerm1 = new FermionTerm(new[] { 0, 1 }.ToLadderSequence());

// These fermion terms are not equal. The following Boolean is `false`.
var sequenceEqual = fermionTerm0 == fermionTerm1;

// However, these terms are equal under Hermitian symmetry.
// We also take the opportunity to demonstrate equivalent constructors
// for hermitian fermion terms
var hermitianFermionTerm0 = new HermitianFermionTerm(fermionTerm0);
var hermitianFermionTerm1 = new HermitianFermionTerm(new[] { 0, 1 });

// These Hermitian fermion terms are equal. The following Boolean is `true`.
var hermitianSequenceEqual = hermitianFermionTerm0 == hermitianFermionTerm1;

// We add the terms to the Hamiltonian with the appropriate coefficient.
// Note that these terms are identical.
hamiltonian.Add(hermitianFermionTerm0, 1.0);
hamiltonian.Add(hermitianFermionTerm1, 1.0);
```

We may simplify this construction using the fact that Hamiltonian operators are Hermitian operators. When adding terms to the Hamiltonian using `Add`, any non-Hermitian term such as `fermionTerm0` is assumed to be paired with its Hermitian conjugate. Thus the following snippet also represents the same Hamiltonian:

```
// We create a `FermionHamiltonian` instance to store the fermion terms.
var hamiltonian = new FermionHamiltonian();

// We construct the term to be added -- note the doubled coefficient.
hamiltonian.Add(new HermitianFermionTerm(new[] { 1, 0 }), 2.0);
```

Using the anti-commutation rules, some `FermionTerm` instances in the Hamiltonian actually correspond to the same sequence of fermionic operators, sometimes up to a minus sign. For instance, consider the Hamiltonian  $\$H=a_0^\dagger a_1^\dagger a_1 a_0 - a_1^\dagger a_0^\dagger a_0 a_1 = 2a_0^\dagger a_1^\dagger a_1 a_0$ , which is a sum of terms constructed above. It may not always be clear to the user that these are equivalent terms, and so they may be added to the Hamiltonian separately. Alternatively, one may be interested in modifying already-existing terms in the Hamiltonian. In these cases, we may combine equivalent terms as follows.

```

// We create a `FermionHamiltonian` instance to store the fermion terms.
var hamiltonian = new FermionHamiltonian();

// We now create two Hermitian fermion terms that are equivalent with respect to
// anti-commutation and Hermitian symmetry.
var terms = new[] {
    (new[] { 0, 1, 1, 0 }, 1.0),
    (new[] { 1, 0, 1, 0 }, 1.0)
}.Select(o => (new HermitianFermionTerm(o.Item1.ToLadderSequence(), o.Item2.ToDoubleCoeff())));

// Now add `terms` to the Hamiltonian.
hamiltonian.AddRange(terms);

// There is only one unique term. `nTerms == 1` is `true`.
var nTerms = hamiltonian.CountTerms();

```

By combining coefficients of equivalent terms, we reduce the total number of terms in the Hamiltonian. Later on, this reduces the number of quantum gates required to simulate the Hamiltonian.

## Internal Representation

A fermionic Hamiltonian with one- and two-body interactions is represented in second-quantized notation as

$$\$ \$ H = \sum_{pq} h_{pq} a^\dagger(p) a(q) + \frac{1}{2} \sum_{pqrs} h_{pqrs} a^\dagger(p) a^\dagger(q) a(r) a(s). \$ \$$$

In this notation, there are at most  $N^2 + N^4$  coefficients. However, many of these coefficients may be collected as they correspond to the same operator. For instance, in the case where  $p, q, r, s$  are distinct indices, we may use the anti-commutation rules to show that:

$$\$ \$ a^\dagger(p) a^\dagger(q) a(r) a(s) = -a^\dagger(q) a^\dagger(p) a(r) a(s) = -a^\dagger(p) a^\dagger(q) a(s) a(r) = a^\dagger(q) a^\dagger(p) a(s) a(r). \$ \$$$

Furthermore, as  $H$  is Hermitian, every non-Hermitian fermionic operator, say  $h_{pqrs} a^\dagger(p) a^\dagger(q) a(r) a(s)$ , has a Hermitian conjugate that is also found in  $H$ . In order to uniquely index groups of terms characterized by these symmetries, we define a canonical ordering on the indices  $(i_1, \dots, i_n, j_1, \dots, j_m)$  of any sequence of  $n+m$  fermionic operators  $a^\dagger(i_1) \dots a^\dagger(i_n) a(j_1) \dots a(j_m)$  as follows:

- All creation operators  $a^\dagger(i)$  are placed before all annihilation operators  $a(j)$ .
- All creation operator indices are sorted in ascending order, that is  $i_1 < i_2 < \dots < i_n$ .
- All annihilation operator indices are sorted in descending order, that is  $j_1 > j_2 > \dots > j_m$ .
- The left-most index is less than or equal to the right-most index, that is  $i_1 \leq j_m$ .

Let us identify this set of canonically ordered indices as

$$\$ \$ (i_1, \dots, i_n, j_1, \dots, j_m) \in S_{n,m}. \$ \$$$

With this canonical ordering, the fermionic Hamiltonian may be expressed as

$$\$ \$ H = \sum_{(p,q) \in S_{1,1}} h'_{pq} \frac{a^\dagger(p) a(q) + a^\dagger(q) a(p)}{2} + \sum_{(p,q,r,s) \in S_{2,2}} h'_{pqrs} \frac{a^\dagger(p) a^\dagger(q) a(r) a(s) + a^\dagger(q) a^\dagger(r) a(s) a(p)}{2}, \$ \$$$

with suitably adapted one- and two-electron integrals  $h'_{pq}$  and  $h'_{pqrs}$ , respectively.

# Symmetries of Molecular Integrals

6/30/2021 • 3 minutes to read • [Edit Online](#)

The inherent symmetry of the Coulomb Hamiltonian, which is the Hamiltonian given in [Quantum Models for Electronic Systems](#), that describes electrons interacting electrically with each other and with the nuclei, leads to a number of symmetries that can be exploited to compress the terms in the Hamiltonian. In general if no further assumptions are made about the basis functions  $\psi_j$  then we only have that

$$h_{pqrs} = h_{qpsr} \tag{★}$$

which can be immediately seen from the integrals in [Quantum Models for Electronic Systems](#) upon noting that their values remain identical if  $p,q$  and  $r,s$  are interchanged from anti-commutation.

If we assume that the spin-orbitals are real-valued (as they are for Gaussian orbital bases) then we further have that

$$h_{pqrs} = h_{qpsr} = h_{srqp} = h_{rspq} = h_{rqps} = h_{psrq} = h_{spqr} = h_{qrsp} \tag{★}$$

Given such assumptions hold, we can use the above symmetries to reduce the data needed to store the matrix elements of the Hamiltonian by a factor of 8; although doing so makes importing data in a consistent way slightly more challenging. Fortunately the Hamiltonian simulation library has subroutines that can be used to import integral files from either [Liqui\\$|rangle\\$](#) or directly from [NWChem](#).

Molecular orbital integrals such as these (for example, the  $h_{pq}$  and  $h_{pqrs}$  terms) are represented using the `OrbitalIntegral` type, which provides a number of helpful functions to express this symmetry.

```
// The code snippets in this section require the following namespaces.  
// Make sure to include these at the top of your file or namespace.  
using Microsoft.Quantum.Chemistry.OrbitalIntegrals;
```

```
// Create a `OrbitalIntegral` instance to store a one-electron molecular  
// orbital integral data.  
var oneElectronOrbitalIndices = new[] { 0, 1 };  
var oneElectronCoefficient = 1.0;  
var oneElectronIntegral = new OrbitalIntegral(oneElectronOrbitalIndices, oneElectronCoefficient);  
  
// This enumerates all one-electron integrals with the same coefficient --  
// an array of equivalent `OrbitalIntegral` instances is generated. In this  
// case, there are two elements.  
var oneElectronIntegrals = oneElectronIntegral.EnumerateOrbitalSymmetries();  
  
// Create a `OrbitalIntegral` instance to store a two-electron molecular orbital integral data.  
var twoElectronOrbitalIndices = new[] { 0, 1, 2, 3 };  
var twoElectronCoefficient = 0.123;  
var twoElectronIntegral = new OrbitalIntegral(twoElectronOrbitalIndices, twoElectronCoefficient);  
  
// This enumerates all two-electron integrals with the same coefficient --  
// an array of equivalent `OrbitalIntegral` instances is generated. In  
// this case, there are 8 elements.  
var twoElectronIntegrals = twoElectronIntegral.EnumerateOrbitalSymmetries();
```

In addition to enumerating over all orbital integrals that are numerically identical, a list of all spin-orbital indices contained in the Hamiltonian represented by an `OrbitalIntegral` may be generated as follows.

```
// Create a `OrbitalIntegral` instance to store a two-electron molecular
// orbital integral data.
var twoElectronIntegral = new OrbitalIntegral(new[] { 0, 1, 2, 3 }, 0.123);

// This enumerates all spin-orbital indices of the `FermionTerm`s in the
// Hamiltonian represented by this integral -- this is an array of array
// of `SpinOrbital` instances.
var twoElectronSpinOrbitalIndices = twoElectronIntegral.EnumerateSpinOrbitals();
```

## Constructing Fermionic Hamiltonians from Molecular Integrals

Rather than constructing a Fermionic Hamiltonian by adding `FermionTerm`s, all terms corresponding to each orbital integral may be added automatically. For example, the following code automatically enumerates over all permutational symmetries and orders the terms in canonical order:

```
// The code snippets in this section require the following namespaces.
// Make sure to include these at the top of your file or namespace.
using Microsoft.Quantum.Chemistry;
using Microsoft.Quantum.Chemistry.OrbitalIntegrals;
using Microsoft.Quantum.Chemistry.Fermion;
// We load this namespace for convenience methods for manipulating arrays.
using System.Linq;
```

```
// Create a `OrbitalIntegral` instance to store a two-electron molecular
// orbital integral data.
var orbitalIntegral = new OrbitalIntegral(new[] { 0, 1, 2, 3 }, 0.123);

// Create an `OrbitalIntegralHamiltonian` instance to store the orbital integral
// terms.
var orbitalIntegralHamiltonian = new OrbitalIntegralHamiltonian();
orbitalIntegralHamiltonian.Add(orbitalIntegral);

// Convert the orbital integral representation to a fermion
// representation. This also requires choosing a convention for
// mapping spin orbital indices to integer indices.
var fermionHamiltonian = orbitalIntegralHamiltonian.ToFermionHamiltonian(IndexConvention.UpDown);

// Alternatively, one can add orbital integrals directly to a fermion Hamiltonian
// as follows. This automatically enumerates over all symmetries, and then
// orders the `HermitianFermionTerm` instances in canonical order. We will need to
// choose an indexing convention as well.
fermionHamiltonian.AddRange(orbitalIntegral
    .ToHermitianFermionTerms(0, IndexConvention.UpDown)
    .Select(o => (o.Item1, o.Item2.ToDoubleCoeff())));
```

# Jordan-Wigner Representation

6/30/2021 • 4 minutes to read • [Edit Online](#)

While second quantized Hamiltonians are conveniently represented in terms of  $a^\dagger$  (creation) and  $a$  (annihilation), these operations are not fundamental operations in quantum computers. As a result, if we wish it implement them on a quantum computer we need to map the operators to unitary matrices that can be implemented on a quantum computer. The Jordan–Wigner representation gives one such map. However, others such as the Bravyi–Kitaev representation also exist and have their own relative advantages and disadvantages. The main advantage of the Jordan–Wigner representation is its simplicity.

The Jordan–Wigner representation is straight forward to derive. Recall that a state  $|\ket{0}_j$  implies that spin orbital  $j$  is empty and  $|\ket{1}_j$  implies that it is occupied. This means that qubits can naturally store the occupation of a given spin orbital. We then have that  $a^\dagger_j |\ket{0}_j = |\ket{1}_j$  and  $a^\dagger_j |\ket{1}_j = 0$ . It is easy to verify that

$$\begin{aligned} a^\dagger_j &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \frac{X_j - iY_j}{2}, \quad a_j \\ &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} = \frac{X_j + iY_j}{2}, \end{aligned}$$

where  $X_j$  and  $Y_j$  are the Pauli- $X$  and  $-Y$  operators acting on qubit  $j$ .

## NOTE

In Q# the  $|\ket{0}$  state represents the  $+1$  eigenstate of the  $Z$  operator. In some areas of physics  $|\ket{0}$  represents the low-energy ground state and thus the  $-1$  eigenstate of the  $Z$  operator. Therefore some formulas might differ from popular literature.

In the chemistry library we use  $|\ket{0}$  to represent an unoccupied spin-orbital. This shows that for a single spin orbital it is easy to represent creation and annihilation operators in terms of unitary matrices that quantum computers understand. Note that while  $X$  and  $Y$  are unitary  $a^\dagger$  and  $a$  are not. We will see later that this does not pose a challenge for simulation.

One problem that remains is that while the above construction works for a single spin orbital, it fails for systems with two or more spin orbitals. Since Fermions are antisymmetric, we know that  $a^\dagger_j a^\dagger_k = -a^\dagger_k a^\dagger_j$  for any  $j$  and  $k$ . However,

$$(\frac{X_j - iY_j}{2})(\frac{X_k - iY_k}{2}) = (\frac{X_k - iY_k}{2})(\frac{X_j - iY_j}{2}).$$

In other words, the two creation operators do not anti-commute as required. This can be remedied though in a straightforward, if inelegant fashion. The fix is to note that Pauli operators naturally anti-commute. In particular,  $XZ = -ZX$  and  $YZ = -ZY$ . Thus, by interspersing  $Z$  operators into the construction of the operator, we can emulate the correct anti-commutation. The full construction is as follows:

$$\begin{aligned} a^\dagger_1 &= (\frac{X - iY}{2}) \otimes 1 \otimes 1 \otimes \dots \otimes 1, \\ a^\dagger_2 &= Z \otimes (\frac{X - iY}{2}) \otimes 1 \otimes 1 \otimes \dots \otimes 1, \\ a^\dagger_3 &= Z \otimes Z \otimes (\frac{X - iY}{2}) \otimes 1 \otimes 1 \otimes \dots \otimes 1, \\ a^\dagger_N &= Z \otimes Z \otimes Z \otimes (\frac{X - iY}{2}) \otimes 1 \otimes 1 \otimes \dots \otimes 1. \end{aligned}$$

It is also convenient to express the number operators,  $n_{j,j}$ , in terms of Pauli operators. Thankfully, the strings of  $Z$  operators (known as Jordan–Wigner strings) cancel after one makes this substitution. After carrying this out (and recalling that  $X_j Y_j = iZ_j$ ), we have

```
## n_j = a^\dagger_j a_j = \frac{(1-Z_j)}{2}. ##
```

## Constructing Hamiltonians in Jordan-Wigner Representation

Once we have invoked the Jordan-Wigner representation translating the Hamiltonian to a sum of Pauli operators is straight forward. One simply has to replace each of the  $a^\dagger$  and  $a$  operators in the Fermionic Hamiltonian with the strings of Pauli-operators given above. When one performs this substitution, there are only five classes of terms within the Hamiltonian. These five classes correspond to the different ways we can pick the  $p,q$  and  $p,q,r,s$  in the one-body and the two-body terms in the Hamiltonian. These five classes, for the case where  $p > q > r > s$  and real-valued orbitals, are

```
\begin{align} h_{pp} a_p^\dagger a_p &= \sum_p \frac{h_{pp}}{2}(1 - Z_p) \\ a_q^\dagger a_p &= \frac{h_{pq}}{2} (a_p^\dagger a_q + a_q^\dagger a_p) \\ h_{pqqp} n_p n_q &= \frac{h_{pqqp}}{4} (1 - Z_p - Z_q + Z_p Z_q) \\ &\quad H_{pqqr} = \frac{h_{pqqr}}{2} (a_p^\dagger a_r + a_r^\dagger a_p) \\ &\quad H_{pqrs} = \frac{h_{pqrs}}{8} \prod_{j=r+1}^{s-1} Z_j \\ &\quad \& \frac{h_{pqrs}}{8} \prod_{j=s+1}^{r-1} Z_j \prod_{k=q+1}^{p-1} Z_k \Big( XXXX - XXYY + XYXY - YYXX \Big) \\ &\quad \& \frac{h_{pqrs}}{8} \prod_{j=s+1}^{r-1} Z_j \prod_{k=q+1}^{p-1} Z_k \Big( YYYY + YYYX + XYXX + XXYY \Big) \end{align}
```

While generating such Hamiltonians by hand only requires applying these replacement rules, doing so would be infeasible for large molecules which can consist of millions of Hamiltonian terms. As an alternative, we can automatically construct the `JordanWignerEncoding` given a `FermionHamiltonian` representation of the Hamiltonian.

```
// Make sure to load these at the top of your file or namespace.  
using Microsoft.Quantum.Chemistry.Fermion;  
using Microsoft.Quantum.Chemistry.Paulis;
```

```
// We create an example `FermionHamiltonian` instance.  
var fermionHamiltonian = new FermionHamiltonian();  
  
// We convert this Fermion Hamiltonian to a Jordan-Wigner representation.  
var jordanWignerEncoding = fermionHamiltonian.ToPauliHamiltonian(QubitEncoding.JordanWigner);
```

Once the Hamiltonians are constructed in this form, we can use a host of quantum simulation algorithms to compile the dynamics generated by  $e^{-iHt}$  into a sequence of elementary gates (within some user definable error tolerance). We discuss the two most popular methods for quantum simulation, qubitization and Trotter-Suzuki formulas, in the algorithmic documentation. We provide implementations for both methods in the Hamiltonian simulation library.

# Simulating Hamiltonian Dynamics

6/30/2021 • 9 minutes to read • [Edit Online](#)

Once the Hamiltonian has been expressed as a sum of elementary operators the dynamics can then be compiled into fundamental gate operations using a host of well-known techniques. Three efficient approaches include Trotter–Suzuki formulas, linear combinations of unitaries, and qubitization. We explain these three approaches below and give concrete Q# examples of how to implement these methods using the Hamiltonian simulation library.

## Trotter–Suzuki Formulas

The idea behind Trotter–Suzuki formulas is simple: express the Hamiltonian as a sum of easy to simulate Hamiltonians and then approximate the total evolution as a sequence of these simpler evolutions. In particular, let  $H = \sum_{j=1}^m H_j$  be the Hamiltonian. Then,

$$\$ e^{-i\sum_{j=1}^m H_j t} = \prod_{j=1}^m e^{-iH_j t} + O(m^2 t^2), \quad \$$$

which is to say that, if  $|t| \ll 1$ , then the error in this approximation becomes negligible. Note that if  $e^{-iHt}$  were an ordinary exponential then the error in this approximation would not be  $O(m^2 t^2)$ : it would be zero. This error occurs because  $e^{-iHt}$  is an operator exponential and as a result there is an error incurred when using this formula due to the fact that the  $H_j$  terms do not, in general, commute (e.g.  $H_j H_k \neq H_k H_j$ ).

If  $t$  is large, Trotter–Suzuki formulas can still be used to simulate the dynamics accurately by breaking it up into a sequence of short time-steps. Let  $r$  be the number of steps taken in the time evolution, so each time step runs for time  $t/r$ . Then, we have that

$$\$ e^{-i\sum_{j=1}^m H_j t} = \left( \prod_{j=1}^m e^{-iH_j t/r} \right)^r + O(m^2 t^2/r), \quad \$$$

which implies that if  $r$  scales as  $m^2 t^2/\epsilon$  then the error can be made at most  $\epsilon$  for any  $\epsilon > 0$ .

More accurate approximations can be built by constructing a sequence of operator exponentials such that the error terms cancel. The simplest such formula, the second order Trotter–Suzuki formula, takes the form

$$\$ U_2(t) = \left( \prod_{j=1}^m e^{-iH_j t/2r} \right) \prod_{j=m}^1 e^{-iH_j t/2r} \right)^r = e^{-iHt} + O(m^3 t^3/r^2), \quad \$$$

the error of which can be made less than  $\epsilon$  for any  $\epsilon > 0$  by choosing  $r$  to scale as  $m^{3/2} t^{3/2}/\sqrt{\epsilon}$ .

Even higher-order formulas, specifically  $(2k)$ -th-order for  $k > 0$ , can be constructed recursively:

$$\$ U_{2k}(t) = [U_{2k-2}(s_k \sim t)]^2 U_{2k-2}([1-4s_k]t) [U_{2k-2}(s_k \sim t)]^2 = e^{-iHt} + O((m t)^{2k+1}/r^{2k}), \quad \$$$

where  $s_k = (4-4^{1/(2k-1)})^{-1}$ .

The simplest is the following fourth order ( $k=2$ ) formula, originally introduced by Suzuki:

$$\$ U_4(t) = [U_2(s_2 \sim t)]^2 U_2([1-4s_2]t) [U_2(s_2 \sim t)]^2 = e^{-iHt} + O(m^5 t^5/r^4), \quad \$$$

where  $s_2 = (4-4^{1/3})^{-1}$ . In general, arbitrarily high-order formulas can be similarly constructed; however, the costs incurred from using more complex integrators often outweigh the benefits beyond fourth order for most practical problems.

In order to make the above strategies work, we need to have a method for simulating a wide class of  $e^{-iH_j t}$ . The simplest family of Hamiltonians, and arguably most useful, that we could use here are Pauli operators. Pauli operators can be easily simulated because they can be diagonalized using Clifford operations (which are standard gates in quantum computing). Further, once they have been diagonalized, their eigenvalues can be found by computing the parity of the qubits on which they act.

For example,

$\$ e^{-iX \otimes X t} = (H \otimes H) e^{-iZ \otimes Z t} (H \otimes H)$  \$\$

where

$\$ e^{-iZ \otimes Z t} = \begin{bmatrix} e^{-it} & 0 & 0 & 0 \\ 0 & e^{it} & 0 & 0 \\ 0 & 0 & e^{-it} & 0 \\ 0 & 0 & 0 & e^{it} \end{bmatrix}$  \$\$

Here,  $e^{-iHt} |00\rangle = e^{it} |00\rangle$  and  $e^{-iHt} |01\rangle = e^{-it} |01\rangle$ , which can be seen directly as a consequence of the fact that the parity of  $|00\rangle$  is  $0$  while the parity of the bit string  $|01\rangle$  is  $1$ .

Exponentials of Pauli operators can be implemented directly in Q# using the [Exp operation](#) operation:

```
use qubits = Qubit[2];
let pauliString = [PauliX, PauliX];
let evolutionTime = 1.0;

// This applies  $e^{-iHt}$  to qubits 0 and 1.
Exp(pauliString, - evolutionTime, qubits);
```

For Fermionic Hamiltonians, the [Jordan–Wigner decomposition](#) conveniently maps the Hamiltonian into a sum of Pauli operators. This means that the above approach can easily be adapted to simulating chemistry. Rather than manually looping over all Pauli terms in the Jordan–Wigner representation, below is a simple example of how running such a simulation within the chemistry would look. Our starting point is a [Jordan–Wigner encoding](#) of the Fermionic Hamiltonian, which we convert to a format suitable for Q#.

```
// Make sure to load these namespaces at the top of your file or namespace.
using Microsoft.Quantum.Chemistry;
using Microsoft.Quantum.Chemistry.OrbitalIntegrals;
using Microsoft.Quantum.Chemistry.Fermion;
using Microsoft.Quantum.Chemistry.Paulis;
using Microsoft.Quantum.Chemistry.QSharpFormat;
```

```

// We create an instance of the `FermionHamiltonian` object class to store the terms.
var hamiltonian = new OrbitalIntegralHamiltonian(new[]
{
    new OrbitalIntegral(new[] { 0, 1, 2, 3 }, 0.123),
    new OrbitalIntegral(new[] { 0, 1 }, 0.456)
}).ToFermionHamiltonian(IndexConvention.UpDown);

// We convert this fermion Hamiltonian to a Jordan-Wigner representation.
var jordanWignerEncoding = hamiltonian.ToPauliHamiltonian(QubitEncoding.JordanWigner);

// We also need to specify an initial quantum state to invoke Q# simulation oracles,
// such as the HartreeFock state on 2 electrons.
var fermionWavefunction = hamiltonian.CreateHartreeFockState(2);

// We now convert the Jordan Wigner representation into a format consumable by Q#.
var qSharpHamiltonianData = jordanWignerEncoding.ToQSharpFormat();
var qSharpWavefunctionData = fermionWavefunction.ToQSharpFormat();
var qSharpData = Convert.ToQSharpFormat(qSharpHamiltonianData, qSharpWavefunctionData);

// Q# simulation oracles could then be invoked with the Q# data as follows.
//TrotterExample.Run(new QuantumSimulator(), qSharpData);

```

This format of the Jordan–Wigner representation that is consumable by the Q# simulation algorithms is a user-defined type `JordanWignerEncodingData`. Within Q#, this format is passed to a convenience function `TrotterStepOracle` that returns an operator approximating time-evolution using the Trotter–Suzuki integrator, in addition to other parameters required for its run.

```

operation TrotterExample (qSharpData: JordanWignerEncodingData) : Unit {
    // Choose the integrator step size
    let stepSize = 1.0;

    // Choose the order of the Trotter–Suzuki integrator.
    let integratorOrder = 4;

    // `oracle` is an operation that applies a single time-step of evolution for duration `stepSize`.
    // `rescale` is just `1.0/stepSize` -- the number of steps required to simulate unit-time evolution.
    // `nQubits` is the number of qubits that must be allocated to run the `oracle` operation.
    let (nQubits, (rescale, oracle)) = TrotterStepOracle (qSharpData, stepSize, integratorOrder);

    // Let us now apply a single time-step.
    use qubits = Qubit[nQubits];
    // Apply single step of time-evolution
    oracle(qubits);

    // Reset all qubits to the 0 state to be successfully released.
    ResetAll(qubits);
}

```

Importantly, this implementation applies some optimizations discussed in [Simulation of Electronic Structure Hamiltonians Using Quantum Computers](#) and [Improving Quantum Algorithms for Quantum Chemistry](#) to minimize the number of single-qubit rotations required, as well as reduce simulation errors.

## Qubitization

Qubitization is an alternative approach to simulation that uses ideas from quantum walks to simulate quantum dynamics. While qubitization requires more qubits than Trotter formulas, the method promises optimal scaling with the evolution time and the error tolerance. For these reasons it has become a favored method for simulating Hamiltonian dynamics in general, and for solving the electronic structure problem in particular.

At a high level, qubitization accomplishes this through the following steps. First, let  $H = \sum_j h_j H_j$  for unitary and Hermitian  $H_j$  and  $h_j \geq 0$ . By performing a pair of reflections, qubitization implements an

operator that is equivalent to

```
$$ W = e^{\pm i \cos^{-1}(H/|h|_1)}, $$
```

where  $|h|_1 = \sum_j |h_j|$ . The next step involves transforming the eigenvalues of the walk operator from  $e^{\pm i \cos^{-1}(E_k/|h|_1)}$ , where  $E_k$  are the eigenvalues of  $H$  to  $e^{\pm i E_k t}$ . This can be achieved using a variety of quantum singular value transformation methods including [quantum signal processing](#).

Alternatively, if only static quantities are desired (such as the ground state energy of the Hamiltonian) then it suffices to apply [phase estimation](#) directly to  $W$  to estimate the ground state energy from the result by taking the cosine of the result. This is significant because it allows the spectral transformation to be performed classically rather than using a quantum singular value transformation method.

On a more detailed level, the implementation of qubitization requires two subroutines that provide the interfaces for the Hamiltonian. Unlike Trotter–Suzuki methods, these subroutines are quantum not classical and their implementation will necessitate using logarithmically more qubits than would be required for a Trotter-based simulation.

The first quantum subroutine that qubitization uses is called `\operatorname{Select}` and it is promised to yield

```
$$ \operatorname{Select} \ket{j} \ket{\psi} = \ket{j} H_j \ket{\psi}, $$
```

where each  $H_j$  is assumed to be Hermitian and unitary. While this may seem to be restrictive, recall that Pauli operators are Hermitian and unitary and so applications like quantum chemistry simulation naturally fall into this framework. The `\operatorname{Select}` operation, perhaps surprisingly, is actually a reflection operation. This can be seen from the fact that  $\operatorname{Select}^2 \ket{j} \ket{\psi} = \ket{j} \ket{\psi}$  since each  $H_j$  is unitary and Hermitian and thus has eigenvalues  $\pm 1$ .

The second subroutine is called `\operatorname{Prepare}`. While the select operation provides a means to coherently access each of the Hamiltonian terms  $H_j$  the prepare subroutine gives a method for accessing the coefficients  $h_j$ ,

```
$$ \operatorname{Prepare} \ket{0} = \sum_j \sqrt{|h_j|} \ket{j}. $$
```

Then, by using a multiply controlled phase gate, we see that

```
$$ \Lambda \ket{0}^{\otimes n} = \begin{cases} -\ket{x} & \text{if } x = 0 \\ \ket{x} & \text{otherwise} \end{cases} $
```

The `\operatorname{Prepare}` operation is not used directly in qubitization, but rather is used to implement a reflection about the state that `\operatorname{Prepare}` creates

```
\begin{aligned} R &= 1 - 2\operatorname{Prepare} \ket{0}\bra{0} \operatorname{Prepare}^{-1} \\ &\quad \operatorname{Prepare} \Lambda \operatorname{Prepare}^{-1}. \end{aligned}
```

The walk operator,  $W$ , can be expressed in terms of the `\operatorname{Select}` and  $R$  operations as

```
$$ W = \operatorname{Select} R, $$
```

which again can be seen to implement an operator that is equivalent (up to an isometry) to  $e^{\pm i \cos^{-1}(H/|h|_1)}$ .

These subroutines are easy to set up in Q#. As an example, consider the simple qubit transverse-Ising Hamiltonian where  $H = X_1 + X_2 + Z_1 Z_2$ . In this case, Q# code that would implement the `\operatorname{Select}` operation is invoked by [MultiplexOperations operation](#), whereas the `\operatorname{Prepare}` operation can be implemented using [PrepareArbitraryState operation](#). An example that involves simulating the Hubbard model can be found as a [Q# sample](#).

Manually specifying these steps for arbitrary chemistry problems would require much effort, which is avoided using the chemistry library. Similarly to the Trotter–Suzuki simulation algorithm above, the

`JordanWignerEncodingData` is passed to the convenience function `QubitizationOracle` that returns the walk-operator, in addition to other parameters required for its run.

```
operation QubitizationExample(qSharpData: JordanWignerEncodingData) : Unit {
    // `oracle` is an operation that applies a single time-step of evolution for duration `stepSize`.
    // `rescale` is just `1.0/oneNorm`, where oneNorm is the sum of absolute values of all probabilities in
    state prepared by `Prepare`.
    // `nQubits` is the number of qubits that must be allocated to run the `oracle` operation.
    let (nQubits, (rescale, oracle)) = QubitizationOracle(qSharpData);

    // Let us now apply a single step of the quantum walk.
    use qubits = Qubit[nQubits];
    // Apply single step of quantum walk.
    oracle(qubits);

    // Reset all qubits to the 0 state to be successfully released.
    ResetAll(qubits);
}
```

Importantly, the implementation [QubitizationOracle function](#) is applicable to arbitrary Hamiltonians specified as a linear combination of Pauli strings. A version optimized for chemistry simulations is invoked using [OptimizedQubitizationOracle function](#). This version is optimized to minimize the number of T gates using techniques discussed in [Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity](#).

# Hartree–Fock Theory

6/30/2021 • 3 minutes to read • [Edit Online](#)

Perhaps the most important quantity in quantum chemistry simulation is the ground state, which is the minimum energy eigenvector of the Hamiltonian matrix. This is because for most molecules at room temperature quantities such as reaction rates are dominated by free energy differences between quantum states that describe the beginning and end of a step in a reaction pathway and at room temperature such intermediate state are usually ground states. While the ground state is typically too hard to learn (even with a quantum computer) because it is a distribution over an exponentially large number of configurations. Quantities such as ground state energy can be learned. For example, if  $\langle \psi | \psi \rangle$  is any pure quantum state then

$$\langle E = \langle \psi | \hat{H} | \psi \rangle \rangle$$

gives the mean energy that the system has in that state. The ground state then is the state that gives the smallest such value. As a result, choosing a state that is as close as possible to the true ground state is vitally important for estimating the energy either directly (as is done in variational eigensolvers) or through phase estimation.

Hartree–Fock theory gives a simple way to construct the initial state for quantum systems. It yields a single Slater-determinant approximation to the ground state of a quantum system. To that end, it finds a rotation within Fock-space that minimizes the ground state energy. In particular, for a system of  $N$  electrons the method performs the rotation

$$\langle \prod_{j=0}^{N-1} a^{\dagger}_j | \rangle \mapsto \prod_{j=0}^{N-1} e^{\langle u_j | a^{\dagger}_j e^{-\langle u_j |} \rangle} | \rangle \rangle$$

with an anti-Hermitian (for example,  $u = -u^{\dagger}$ ) matrix  $u = \sum_{pq} u_{pq} a^{\dagger}_p a_q$ . It should be noted that the matrix  $u$  represents the orbital rotations and  $a^{\dagger}_j$  and  $a_j$  represent creation and annihilation operators for electrons occupying Hartree–Fock molecular spin-orbitals.

The matrix  $u$  is then optimized to minimize the expected energy  $\langle \prod_{j=0}^{N-1} a^{\dagger}_j | H | \prod_{k=0}^{N-1} a_k \rangle$ . While such optimization problems may be generically hard, in practice the Hartree–Fock algorithm tends to rapidly converge to a near-optimal solution to the optimization problem, especially for closed-shell molecules in the equilibrium geometries. We may specify these states as an instance of the `FermionWavefunction` object. For instance, the state  $|a^{\dagger}_1 a^{\dagger}_2 a^{\dagger}_6\rangle$  is instantiated in the chemistry library as follows.

```
// The code snippets in this section require the following namespaces.  
// Make sure to include these at the top of your file or namespace.  
using Microsoft.Quantum.Chemistry;  
using Microsoft.Quantum.Chemistry.OrbitalIntegrals;  
using Microsoft.Quantum.Chemistry.Fermion;
```

```
// Create a list of integer indices of the creation operators  
var indices = new[] { 1, 2, 6 };  
  
// Convert the list of indices to a `FermionWavefunction` object.  
// In this case, the indices are integers, so we use the `int`  
// type specialization.  
var wavefunction = new FermionWavefunction<int>(indices);
```

It is also possible to index wave functions with `SpinOrbital` indices, and then convert these indices to integers

as follows.

```
// Create a list of spin orbital indices of the creation operators
var indices = new[] { (0, Spin.d), (1,Spin.u), (3, Spin.u) };

// Convert the list of indices to a `FermionWavefunction` object.
var wavefunctionSpinOrbital = new FermionWavefunction<SpinOrbital>(indices.ToSpinOrbitals());

// Convert a wavefunction indexed by spin orbitals to
// one indexed by integers
var wavefunctionInt = wavefunctionSpinOrbital.ToIndexing(IndexConvention.UpDown);
```

The most striking feature about Hartree–Fock theory is that it yields a quantum state that has no entanglement between the electrons. This means that it often provides a suitable qualitative description of properties of molecular systems.

The Hartree–Fock state may also be reconstructed from a `FermionHamiltonian` as follows.

```
// We initialize a fermion Hamiltonian.
var fermionHamiltonian = new FermionHamiltonian();

// Create a Hartree-Fock state from the Hamiltonian
// with, say, `4` occupied spin orbitals.
var wavefunction = fermionHamiltonian.CreateHartreeFockState(nElectrons: 4);
```

However, obtaining accurate results, especially for strongly correlated systems, necessitate quantum states that go beyond Hartree–Fock theory.

# Correlated wavefunctions

6/30/2021 • 5 minutes to read • [Edit Online](#)

For many systems, particularly those near the equilibrium geometry, Hartree–Fock theory provides a qualitative description of molecular properties through a single-determinant reference state. However, in order to achieve quantitative accuracy, one must also consider correlation effects.

In this context, it is important to distinguish between dynamic and non-dynamic correlations. Dynamical correlations arise from the tendency of electrons to stay apart, such as due to interelectronic repulsion. This effect can be modelled by considering excitations of electrons out of the reference state. Non-dynamic correlations arise when the wavefunction is dominated by two or more configurations at zeroth order, even to achieve only a qualitative description of the system. This necessitates a superposition of determinants and is an example of a multireference wavefunction.

The chemistry library provides a way to specify a zeroth order wavefunction for the multireference problem as a superposition of determinants. This approach, which we call sparse multireference wavefunctions, is effective when only a few components suffice to specify the superposition. The library also provides a method to include dynamic correlations on top of a single-determinant reference via the generalized unitary coupled-cluster ansatz. Furthermore, it also constructs quantum circuits that generate these states on a quantum computer. These states may be specified in the [Broombridge schema](#), and we also provide the functionality to manually specify these states through the chemistry library.

## Sparse multi-reference wavefunction

A multi-reference state  $\ket{\psi_{\text{MCSCF}}}$  may be specified explicitly as a linear combination of  $N$ -electron Slater determinants.

```
\begin{align} \ket{\psi_{\text{MCSCF}}} \propto \sum_{i_1 < i_2 < \dots < i_N} \lambda_{i_1, i_2, \dots, i_N} a^{\dagger}_{i_1} a^{\dagger}_{i_2} \dots a^{\dagger}_{i_N} \ket{0}. \end{align}
```

For example, the state  $0.1 a^{\dagger}_1 a^{\dagger}_2 a^{\dagger}_6 - 0.2 a^{\dagger}_2 a^{\dagger}_5 \ket{0}$  may be specified in the chemistry library as follows.

```
// Make sure to include this namespace at the top of your file or namespace.
using Microsoft.Quantum.Chemistry.Fermion;
```

```
// Create a list of tuples where the first item of each
// tuple are indices to the creation operators acting on the
// vacuum state, and the second item is the coefficient
// of that basis state.
var superposition = new[] {
    (new[] {1, 2, 6}, 0.1),
    (new[] {2, 1, 5}, -0.2);

// Create a fermion wavefunction object that represents the superposition.
var wavefunction = new FermionWavefunction<int>(superposition);
```

This explicit representation of the superposition components is effective when only a few components need to be specified. One should avoid using this representation when many components are required to accurately capture the desired state. The reason for this is the gate cost of quantum circuit that prepares this state on a quantum computer, which scales at least linearly with the number of superposition components, and at most quadratically with the one-norm of the superposition amplitudes.

# Unitary coupled-cluster wavefunction

It is also possible to specify a unitary coupled-cluster wavefunction  $|\psi_{\text{UCC}}\rangle$  using the chemistry library. In this situation, we have a single-determinant reference state, say,  $|\psi_{\text{SCF}}\rangle$ . The components of the unitary coupled-cluster wavefunction are then specified implicitly through a unitary operator acting on a reference state. This unitary operator is commonly written as  $e^{\langle T - T^\dagger \rangle}$ , where  $T^\dagger$  is the anti-Hermitian cluster operator. Thus

```
\begin{align} |\psi_{\text{UCC}}\rangle = e^{\langle T - T^\dagger \rangle} |\psi_{\text{SCF}}\rangle. \end{align}
```

It is also common to split the cluster operator  $T = T_1 + T_2 + \dots$  into parts, where each part  $T_j$  contains  $j$ -body terms. In generalized coupled-cluster theory, the one-body cluster operator (singles) is of the form

```
\begin{align} T_1 = \sum_{pq} t^{pq} a_p^\dagger a_q, \end{align}
```

and two-body cluster operator (doubles) is of the form

```
\begin{align} T_2 = \sum_{pqrs} t^{pqrs} a_p^\dagger a_q^\dagger a_r a_s. \end{align}
```

Higher-order terms (triples, quadruples, etc.) are possible, but not currently supported by the chemistry library.

For example, let  $|\psi_{\text{SCF}}\rangle = a_1^\dagger a_2^\dagger |0\rangle$ , and let  $T = 0.123 a_1^\dagger a_1 + 0.456 a_0^\dagger a_3^\dagger a_1 a_2 - 0.789 a_3^\dagger a_2 a_1 a_0$ . Then this state is instantiated in the chemistry library as follows.

```
// The code snippets in this section require the following namespaces.  
// Make sure to include these at the top of your file or namespace.  
using Microsoft.Quantum.Chemistry;  
using Microsoft.Quantum.Chemistry.OrbitalIntegrals;  
using Microsoft.Quantum.Chemistry.LadderOperators;  
using Microsoft.Quantum.Chemistry.Fermion;  
// We load this namespace for convenience methods for manipulating arrays.  
using System.Linq;
```

```
// Create a list of indices of the creation operators  
// for the single-reference state  
var reference = new[] { 1, 2 };  
  
// Create a list describing the cluster operator  
// The first half of each list of integers will be  
// associated with the creation operators, and  
// the second half with the annihilation operators.  
var clusterOperator = new[]  
{  
    (new [] {0, 1}, 0.123),  
    (new [] {0, 3, 1, 2}, 0.456),  
    (new [] {3, 2, 1, 0}, 0.789)  
};  
  
// Create a fermion wavefunction object that represents the  
// unitary coupled-cluster wavefunction. It is assumed implicitly  
// that the exponent of the unitary coupled-cluster operator  
// is the cluster operator minus its Hermitian conjugate.  
var wavefunction = new FermionWavefunction<int>(reference, clusterOperator);
```

Spin conservation may be made explicit by instead specifying `Spinorbital` indices instead of integer indices. For example, let  $|\psi_{\text{SCF}}\rangle = a_{\{0,\uparrow}\}^\dagger a_{\{1,\downarrow}\}^\dagger a_{\{2,\uparrow}\}^\dagger a_{\{3,\downarrow}\}|0\rangle$ , and let  $T = 0.123 a_{\{0,\uparrow}\}^\dagger a_{\{1,\uparrow}\}^\dagger a_{\{2,\uparrow}\}^\dagger a_{\{3,\uparrow}\}^\dagger + 0.456 a_{\{0,\uparrow}\}^\dagger a_{\{1,\uparrow}\}^\dagger a_{\{2,\uparrow}\}^\dagger a_{\{3,\downarrow}\}$ .

$\downarrow \text{downarrow} \} a_{\{1, \uparrow}\text{uparrow} \} a_{\{2, \downarrow \text{downarrow}} - 0.789 a^{\dagger} \text{dagger}_{\{3, \uparrow}\text{uparrow} \} a^{\dagger} \text{dagger}_{\{2, \uparrow}\text{uparrow} \}$   
 $a_{\{1, \uparrow}\text{uparrow} \} a_{\{0, \uparrow}\text{uparrow} \}$ \$ be spin conserving. Then this state is instantiated in the chemistry library as follows.

```
// Create a list of indices of the creation operators
// for the single-reference state
var reference = new[] { (1, Spin.u), (2, Spin.d) }.ToSpinOrbitals();

// Create a list describing the cluster operator
// The first half of each list of integers will be
// associated with the creation operators, and
// the second half with the annihilation operators.
var clusterOperator = new[]
{
    (new [] {(0, Spin.u), (1, Spin.u)}, 0.123),
    (new [] {(0, Spin.u), (3, Spin.d), (1, Spin.u), (2, Spin.d)}, 0.456),
    (new [] {(3, Spin.u), (2, Spin.u), (1, Spin.u), (0, Spin.u)}, 0.789)
}.Select(o => (o.Item1.ToSpinOrbitals(), o.Item2));

// Create a fermion wavefunction object that represents the
// unitary coupled-cluster wavefunction. It is assumed implicitly
// that the exponent of the unitary coupled-cluster operator
// is the cluster operator minus its Hermitian conjugate.
var wavefunctionSpinOrbital = new FermionWavefunction<SpinOrbital>(reference, clusterOperator);

// Convert the wavefunction indexed by spin-orbitals to one indexed
// by integers
var wavefunctionInteger = wavefunctionSpinOrbital.ToIndexing(IndexConvention.UpDown);
```

We also provide a convenience function that enumerates over all spin-converting cluster operators that annihilate only occupied spin-orbitals and excite to only unoccupied spin-orbitals.

```
// Create a list of indices of the creation operators
// for the single-reference state
var reference = new[] { (1, Spin.u), (2, Spin.d) }.ToSpinOrbitals();

// Generate all spin-converting excitations from spin-orbitals
// occupied by the reference state to virtual orbitals.
var generatedExcitations = reference.CreateAllUCCSDSingletExcitations(nOrbitals: 3).Excitations;

// This is the list of expected spin-conserving excitations
var expectedExcitations = new[]
{
    new []{ (0, Spin.u), (1, Spin.u)},
    new []{ (2, Spin.u), (1, Spin.u)},
    new []{ (0, Spin.d), (2, Spin.d)},
    new []{ (1, Spin.d), (2, Spin.d)},
    new []{ (0, Spin.u), (0, Spin.d), (2, Spin.d), (1, Spin.u)},
    new []{ (0, Spin.u), (1, Spin.d), (2, Spin.d), (1, Spin.u)},
    new []{ (0, Spin.d), (2, Spin.u), (2, Spin.d), (1, Spin.u)},
    new []{ (1, Spin.d), (2, Spin.u), (2, Spin.d), (1, Spin.u)}
}.Select(o => new IndexOrderedSequence<SpinOrbital>(o.ToLadderSequence()));

// The following two assertions are true, and verify that the generated
// excitations exactly match the expected excitations.
var bool0 = generatedExcitations.Keys.All(expectedExcitations.Contains);
var bool1 = generatedExcitations.Count() == expectedExcitations.Count();
```

# Quantum chemistry examples

3/5/2021 • 2 minutes to read • [Edit Online](#)

In the quantum chemistry concepts, we manually constructed example fermion Hamiltonians. We now combine the chemistry simulation algorithms outlined in [Simulating Hamiltonian dynamics](#) with [quantum phase estimation](#) in the canon library. This combination allows us to obtain estimates of energy levels in the represented molecule, which is one of the key applications of quantum chemistry on a quantum computer.

Instead of specifying terms of the Hamiltonian one-by-one, we also work through some examples that allow us to perform quantum chemistry experiments at scale. We begin with examples that load a chemistry Hamiltonian encoded in the [Broombridge schema](#).

For molecules that are too large to simulate on the [full state simulator](#), interesting science can still be performed. For instance, the resource costs of performing large chemistry simulations may still be evaluated by targeting the [trace simulator](#).

Let us now illustrate interesting applications of the chemistry simulation library through a few of the provided samples.

# Obtaining energy level estimates

6/30/2021 • 4 minutes to read • [Edit Online](#)

Estimating the values of energy levels is one of the principal applications of quantum chemistry. This article outlines how you can perform this for the canonical example of molecular hydrogen. The sample referenced in this section is [MolecularHydrogen](#) in the chemistry samples repository. A more visual example that plots the output is the [MolecularHydrogenGUI](#) demo.

## Estimating the energy values of molecular hydrogen

The first step is to construct the Hamiltonian representing molecular hydrogen. Although you can construct this using the NWChem tool, for brevity, this sample adds the Hamiltonian terms manually.

```
// The code snippets in this section require the following namespaces.  
// Make sure to include these at the top of your file or namespace.  
using Microsoft.Quantum.Chemistry.OrbitalIntegrals;  
using Microsoft.Quantum.Chemistry.Fermion;  
using Microsoft.Quantum.Chemistry.Paulis;  
using Microsoft.Quantum.Chemistry.QSharpFormat;  
using Microsoft.Quantum.Simulation.Simulators;
```

```
// These orbital integrals are represented using the OrbitalIntegral  
// data structure.  
var energyOffset = 0.713776188; // This is the coulomb repulsion  
var nElectrons = 2; // Molecular hydrogen has two electrons  
var orbitalIntegrals = new OrbitalIntegral[]  
{  
    new OrbitalIntegral(new[] { 0,0 }, -1.252477495),  
    new OrbitalIntegral(new[] { 1,1 }, -0.475934275),  
    new OrbitalIntegral(new[] { 0,0,0,0 }, 0.674493166),  
    new OrbitalIntegral(new[] { 0,1,0,1 }, 0.181287518),  
    new OrbitalIntegral(new[] { 0,1,1,0 }, 0.663472101),  
    new OrbitalIntegral(new[] { 1,1,1,1 }, 0.697398010),  
    // This line adds the identity term.  
    new OrbitalIntegral(new int[] { }, energyOffset)  
};  
  
// Initialize a fermion Hamiltonian data structure and add terms to it.  
var fermionHamiltonian = new OrbitalIntegralHamiltonian(orbitalIntegrals).ToFermionHamiltonian();
```

Simulating the Hamiltonian requires converting the fermion operators to qubit operators. This conversion is performed through the Jordan-Wigner encoding as follows:

```

// The Jordan-Wigner encoding converts the fermion Hamiltonian,
// expressed in terms of fermionic operators, to a qubit Hamiltonian,
// expressed in terms of Pauli matrices. This is an essential step
// for simulating our constructed Hamiltonians on a qubit quantum
// computer.
var jordanWignerEncoding = fermionHamiltonian.ToPauliHamiltonian(QubitEncoding.JordanWigner);

// You also need to create an input quantum state to this Hamiltonian.
// Use the Hartree-Fock state.
var fermionWavefunction = fermionHamiltonian.CreateHartreeFockState(nElectrons);

// This Jordan-Wigner data structure also contains a representation
// of the Hamiltonian and wavefunction made for consumption by the Q# operations.
var qSharpHamiltonianData = jordanWignerEncoding.ToQSharpFormat();
var qSharpWavefunctionData = fermionWavefunction.ToQSharpFormat();
var qSharpData = Convert.ToQSharpFormat(qSharpHamiltonianData, qSharpWavefunctionData);

```

Next, pass `qSharpData`, which represents the Hamiltonian, to the `TrotterStepOracle` function (available in the [Microsoft.Quantum.Chemistry.JordanWigner](#) namespace). `TrotterStepOracle` returns a quantum operation that approximates the real-time evolution of the Hamiltonian. For more information, see [Simulating Hamiltonian dynamics](#).

```

operation RunTrotterStep(qSharpData: JordanWignerEncodingData) : Unit {
    // Choose the integrator step size
    let stepSize = 1.0;

    // Choose the order of the Trotter-Suzuki integrator.
    let integratorOrder = 4;

    // `oracle` is an operation that applies a single time-step of evolution for duration `stepSize`.
    // `rescale` is just `1.0/stepSize` -- the number of steps required to simulate unit-time evolution.
    // `nQubits` is the number of qubits that must be allocated to run the `oracle` operation.
    let (nQubits, (rescale, oracle)) = TrotterStepOracle (qSharpData, stepSize, integratorOrder);
}

```

At this point, you can use the standard library's [phase estimation algorithms](#) to learn the ground state energy using the previous simulation. This requires preparing a good approximation to the quantum ground state. Suggestions for such approximations are provided in the [Broombridge](#) schema. However, absent these suggestions, the default approach adds a number of `hamiltonian.NElectrons` electrons to greedily minimize the diagonal one-electron term energies. The phase estimation functions and operations are provided in DocFX notation in the [Microsoft.Quantum.Characterization](#) namespace, as well as the [Microsoft.Quantum.Simulation](#) namespace. Make sure to open these in your Q# file when using the `GetEnergyByTrotterization` code shown here.

The following snippet shows how the real-time evolution output by the chemistry simulation library integrates with quantum phase estimation.

```

open Microsoft.Quantum.Intrinsic;
open Microsoft.Quantum.Chemistry.JordanWigner;
open Microsoft.Quantum.Characterization;
open Microsoft.Quantum.Simulation;

operation GetEnergyByTrotterization (
    qSharpData : JordanWignerEncodingData,
    nBitsPrecision : Int,
    trotterStepSize : Double,
    trotterOrder : Int) : (Double, Double) {

    // The data describing the Hamiltonian for all these steps is contained in
    // `qSharpData`
    let (nSpinOrbitals, fermionTermData, statePrepData, energyOffset) = qSharpData!;

    // Using a Product formula, also known as `Trotterization`, to
    // simulate the Hamiltonian.
    let (nQubits, (rescaleFactor, oracle)) =
        TrotterStepOracle(qSharpData, trotterStepSize, trotterOrder);

    // The operation that creates the trial state is defined here.
    // By default, greedy filling of spin-orbitals is used.
    let statePrep = PrepareTrialState(statePrepData, _);

    // Using the Robust Phase Estimation algorithm
    // of Kimmel, Low and Yoder.
    let phaseEstAlgorithm = RobustPhaseEstimation(nBitsPrecision, _,_);

    // This runs the quantum algorithm and returns a phase estimate.
    let estPhase = EstimateEnergy(nQubits, statePrep, oracle, phaseEstAlgorithm);

    // Now, obtain the energy estimate by rescaling the phase estimate
    // with the trotterStepSize. We also add the constant energy offset
    // to the estimated energy.
    let estEnergy = estPhase * rescaleFactor + energyOffset;

    // Return both the estimated phase and the estimated energy.
    return (estPhase, estEnergy);
}

```

You can now invoke the Q# code from the host program. The following C# code creates a full-state simulator and runs `GetEnergyByTrotterization` to obtain the ground state energy.

```

using (var qsim = new QuantumSimulator())
{
    // Specify the bits of precision desired in the phase estimation algorithm
    var bits = 7;

    // Specify the step size of the simulated time evolution. The step size needs to
    // be small enough to avoid aliasing of phases, and also to control the
    // error of simulation.
    var trotterStep = 0.4;

    // Choose the Trotter integrator order
    Int64 trotterOrder = 1;

    // As the quantum algorithm is probabilistic, run a few trials.

    // This may be compared to true value of
    Console.WriteLine("Exact molecular hydrogen ground state energy: -1.137260278.\n");
    Console.WriteLine("----- Performing quantum energy estimation by Trotter simulation algorithm");
    for (int i = 0; i < 5; i++)
    {
        // EstimateEnergyByTrotterization
        var (phaseEst, energyEst) = GetEnergyByTrotterization.Run(qsim, qSharpData, bits, trotterStep,
trotterOrder).Result;
        Console.WriteLine("Estimated molecular hydrogen ground state energy: {0}", energyEst);
    }
}

```

The operation returns two parameters:

- `energyEst` is the estimate of the ground state energy and should be close to `-1.137` on average.
- `phaseEst` is the raw phase returned by the phase estimation algorithm. This useful for diagnosing aliasing when it occurs due to a `trotterStep` value that is too large.

# Loading a Hamiltonian from file

6/30/2021 • 2 minutes to read • [Edit Online](#)

Previously, we constructed Hamiltonians by adding individual terms to it. While this is fine for small examples, quantum chemistry at scale require Hamiltonians with millions or billions of terms. Such Hamiltonians, generated by chemistry packages such as NWChem, are too large to import by hand. In this sample, we illustrate how a `FermionHamiltonian` instance may be automatically generated from a molecule represented by the [Broombridge schema](#). For reference, one may inspect the provided [LithiumHydrideGUI sample](#), or the [RunSimulation sample](#). Limited support is also available for importing from the format consumed by [LIQUIl>](#).

Let us consider the example of the Nitrogen molecule, which is provided in the [IntegralData/YAML/N2](#) folder of the samples repository. The method for loading the `Broombridge` schema is straightforward.

```
// The code snippets in this section require the following namespaces.  
// Make sure to include these at the top of your file or namespace.  
using Microsoft.Quantum.Chemistry;  
using Microsoft.Quantum.Chemistry.Broombridge;  
using Microsoft.Quantum.Chemistry.OrbitalIntegrals;  
using Microsoft.Quantum.Chemistry.Fermion;  
using Microsoft.Quantum.Chemistry.Paulis;  
using Microsoft.Quantum.Chemistry.QSharpFormat;  
using System.Linq;
```

```
// This is the name of the file we want to load  
var filename = @"n2_1_00Re_sto3g.nw.out.yaml";  
// This is the directory containing the file  
var root = @"IntegralData\YAML\N2";  
// This creates a stream that can be passed to the deserializer  
using var textReader = System.IO.File.OpenText($"{root}\{filename}");  
  
// This deserializes a Broombridge file, given its filename.  
var broombridge = BroombridgeSerializer.Deserialize(textReader);  
  
// Note that the deserializer returns a list of `ElectronicStructureProblem` instances,  
// as the file might describe multiple Hamiltonians. In this example, there is  
// only one Hamiltonian. So we use `.First()`, which selects the first element of the list.  
var problem = broombridge.First();  
  
// This extracts the `OrbitalIntegralHamiltonian` from Broombridge format,  
// then converts it to a fermion Hamiltonian, then to a Jordan-Wigner  
// representation.  
var orbitalIntegralHamiltonian = problem.OrbitalIntegralHamiltonian;  
var fermionHamiltonian = orbitalIntegralHamiltonian.ToFermionHamiltonian(IndexConvention.UpDown);  
var jordanWignerEncoding = fermionHamiltonian.ToPauliHamiltonian(QubitEncoding.JordanWigner);
```

The Broombridge schema also contains suggestions for the initial state to be prepared. The labels, for example, "`|G0"` or "`|E1"`, for these states may be seen by inspecting the file. In order to prepare these initial states, the `qSharpData` consumed by the Q# quantum algorithms is obtained similar to the [previous section](#), but with an additional parameter selecting the desired initial state. For instance,

```

// The desired initial state, assuming that a description of it is present in the
// Broombridge schema.
var state = "|E1>";
var wavefunction = problem.InitialStates[state].ToIndexing(IndexConvention.UpDown);

// This creates the qSharpData consumable by the Q# chemistry library algorithms.
var qSharpHamiltonianData = jordanWignerEncoding.ToQSharpFormat();
var qSharpWavefunctionData = wavefunction.ToQSharpFormat();
var qSharpData = Convert.ToQSharpFormat(qSharpHamiltonianData, qSharpWavefunctionData);

```

We may also load a Hamiltonian from the LIQUI|> format, using a very similar syntax.

```

// This is the name of the file we want to load
var filename = @"fe2s2_sto3g.dat"; // This is Ferredoxin.
// This is the directory containing the file
var root = @"IntegralData\Liquid";
// This creates a stream that can be passed to the deserializer
using var textReader = System.IO.File.OpenText($"{root}\{filename}");

// Deserialize the LiQuiD format.
var problem = LiQuidSerializer.Deserialize(textReader).First();

// This extracts the `OrbitalIntegralHamiltonian` from Broombridge format,
// then converts it to a fermion Hamiltonian, then to a Jordan-Wigner
// representation.
var orbitalIntegralHamiltonian = problem.OrbitalIntegralHamiltonian;
var fermionHamiltonian = orbitalIntegralHamiltonian.ToFermionHamiltonian(IndexConvention.UpDown);
var jordanWignerEncoding = fermionHamiltonian.ToPauliHamiltonian(QubitEncoding.JordanWigner);

// This is a data structure representing the Jordan-Wigner encoding
// of the Hamiltonian that we may pass to a Q# algorithm.
var qSharpHamiltonianData = jordanWignerEncoding.ToQSharpFormat();

```

By following this process from any instance of Broombridge, or any intermediate step, quantum algorithms such as quantum phase estimation may be run on the specified electronic structure problem.

# Obtaining resource counts

6/30/2021 • 3 minutes to read • [Edit Online](#)

The cost of simulating  $n$  qubits on classical computers scales exponentially with  $n$ . This greatly limits the size of a quantum chemistry simulation we may perform with the full-state simulator. For large instances of chemistry, we may nevertheless obtain useful information. Here, we examine how resource costs, such as the number of T-gates or CNOT gates, for simulating chemistry may be obtained in an automated fashion using the [trace simulator](#). Such information informs us of when quantum computers might be large enough to run these quantum chemistry algorithms. For reference, see the provided [GetGateCount](#) sample.

Let us assume that we already have a `FermionHamiltonian` instance, say, loaded from the Broombridge schema as discussed in the [loading-from-file](#) example.

```
// The code snippets in this section require the following namespaces.  
// Make sure to include these at the top of your file or namespace.  
using Microsoft.Quantum.Chemistry;  
using Microsoft.Quantum.Chemistry.Broombridge;  
using Microsoft.Quantum.Chemistry.OrbitalIntegrals;  
using Microsoft.Quantum.Chemistry.Fermion;  
using Microsoft.Quantum.Chemistry.Paulis;  
using Microsoft.Quantum.Chemistry.QSharpFormat;  
using Microsoft.Quantum.Simulation.Simulators.QCTraceSimulators;  
using System.Linq;
```

```
// Filename of Hamiltonian to be loaded.  
var filename = @"...";  
// This creates a stream that can be passed to the deserializer  
using var textReader = System.IO.File.OpenText(filename);  
  
// This deserializes Broombridge.  
var problem = BroombridgeSerializer.Deserialize(textReader).First();  
  
// This extracts the `OrbitalIntegralHamiltonian` from Broombridge format,  
// then converts it to a fermion Hamiltonian, then to a Jordan-Wigner  
// representation.  
var orbitalIntegralHamiltonian = problem.OrbitalIntegralHamiltonian;  
var fermionHamiltonian = orbitalIntegralHamiltonian.ToFermionHamiltonian(IndexConvention.UpDown);  
var jordanWignerEncoding = fermionHamiltonian.ToPauliHamiltonian(QubitEncoding.JordanWigner);  
  
// The desired initial state, assuming that a description of it is present in the  
// Broombridge schema.  
var state = "...";  
var wavefunction = problem.InitialStates[state].ToIndexing(IndexConvention.UpDown);  
  
// This is a data structure representing the Jordan-Wigner encoding  
// of the Hamiltonian that we may pass to a Q# algorithm.  
var qSharpHamiltonianData = jordanWignerEncoding.ToQSharpFormat();  
var qSharpWavefunctionData = wavefunction.ToQSharpFormat();  
var qSharpData = Convert.ToQSharpFormat(qSharpHamiltonianData, qSharpWavefunctionData);
```

The syntax for obtaining resource estimates is almost identical to running the algorithm on the full-state simulator. We simply choose a different target machine. For the purposes of resource estimates, it suffices to evaluate the cost of a single Trotter step, or a quantum walk created by the Qubitization technique. The boilerplate for invoking these algorithms is as follows.

```

open Microsoft.Quantum.Intrinsic;
open Microsoft.Quantum.Chemistry.JordanWigner;

/// This allocates qubits and applies a single Trotter step.
operation RunTrotterStep (qSharpData: JordanWignerEncodingData) : Unit {

    // The data describing the Hamiltonian for all these steps is contained in
    // `qSharpData`
    // We use a Product formula, also known as `Trotterization` to
    // simulate the Hamiltonian.
    // The integrator step size does not affect the gate cost of a single step.
    let trotterStepSize = 1.0;

    // Order of integrator
    let trotterOrder = 1;
    let (nQubits, (rescaleFactor, oracle)) = TrotterStepOracle(qSharpData, trotterStepSize, trotterOrder);

    // We now allocate qubits and run a single step.
    use qubits = Qubit[nQubits];
    oracle(qubits);
    ResetAll(qubits);
}

/// This allocates qubits and applies a single qubitization step.
operation RunQubitizationStep (qSharpData: JordanWignerEncodingData) : Double {

    // The data describing the Hamiltonian for all these steps is contained in
    // `qSharpData`
    let (nQubits, (l1Norm, oracle)) = QubitizationOracle(qSharpData);

    // We now allocate qubits and run a single step.
    use qubits = Qubit[nQubits] {
        oracle(qubits);
        ResetAll(qubits);
    }

    return l1Norm;
}

```

We now configure the trace simulator to track the resources we are interested in. In this case, we count primitive quantum operations by setting the `usePrimitiveOperationsCounter` flag to `true`. A technical detail `throwOnUnconstraintMeasurement` is set to `false` to avoid exceptions in cases where the Q# code does not correctly assert of probability of measurement outcomes, if any are performed.

```

private static QCTraceSimulator CreateAndConfigureTraceSim()
{
    // Create and configure Trace Simulator
    var config = new QCTraceSimulatorConfiguration()
    {
        UsePrimitiveOperationsCounter = true,
        ThrowOnUnconstrainedMeasurement = false
    };

    return new QCTraceSimulator(config);
}

```

We now run the quantum algorithm from the driver program as follows.

```
// Instantiate a trace simulator instance
QCTraceSimulator sim = CreateAndConfigureTraceSim();

// Run the quantum algorithm on the trace simulator.
RunQubitizationStep.Run(sim, qSharpData);

// Print all resource counts to file.
var gateStats = sim.ToCSV();
foreach (var x in gateStats)
{
    System.IO.File.WriteAllLines($"QubitizationGateCountEstimates.{x.Key}.csv", new string[] { x.Value });
}
```

# End-to-end with NWChem

6/1/2021 • 5 minutes to read • [Edit Online](#)

In this article, you will walk through an example of getting gate counts for quantum chemistry simulation, starting from an [NWChem](#) input deck. Before proceeding with this example, make sure that you've installed Docker, following the [installation and validation guide](#).

For more information:

- [Structure of NWChem input decks](#)
  - [Input deck commands for use with the Quantum Development Kit](#)
- [Installing the chemistry library and dependencies](#)
- [Resource counting](#)

## NOTE

This example requires Windows PowerShell Core to run. Download PowerShell Core for Windows, macOS, or Linux at <https://github.com/PowerShell/PowerShell>.

## Importing required PowerShell modules

If you haven't already done so, clone the [Microsoft/Quantum repository](#), which contains samples and utilities for working with the Quantum Development Kit:

```
git clone https://github.com/Microsoft/Quantum
```

Once you've cloned `Microsoft/Quantum`, perform `cd` into the `utilities/` folder and import the PowerShell module for working with Docker and NWChem:

```
cd utilities  
Import-Module InvokeNWChem.psm1
```

## NOTE

By default, Windows prevents the running of any scripts or modules as a security measure. To allow modules such as `Invoke-NWChem.psm1` to run on Windows, you may need to change the policy. To do so, run the `Set-ExecutionPolicy` command:

```
Set-ExecutionPolicy RemoteSigned -Scope Process
```

The policy will revert when you exit PowerShell. If you would like to save the policy, use a different value for `-Scope`:

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

You should now have the `Convert-NWChemToBroombridge` command available:

```
Get-Command -Module InvokeNWChem
```

Next, we'll import the `Get-GateCount` command provided with the `GetGateCount` sample. For full details, see the [instructions provided with the sample](#). Next, run the following, substituting `<runtime>` with either `win10-x64`, `osx-x64`, or `linux-x64`, depending on your operating system:

```
cd ..\Chemistry\GetGateCount  
dotnet publish --self-contained -r <runtime>  
Import-Module .\bin\Debug\netcoreapp2.1\<runtime>\publish\get-gatecount.dll
```

You should now have the `Get-GateCount` command available:

```
Get-Command Get-GateCount
```

## Input decks

The NWChem package takes a text file called an *input deck* which specifies a quantum chemistry problem to be solved, along with other parameters such as memory allocation settings. For this example, we'll use one of the pre-made input decks that comes with NWChem. First, clone the [nwchemgit/nwchem repository](#):

### NOTE

Since this is a very large repository, we can do a shallow clone to save some bandwidth and disk space, using the `--depth 1` argument. This is optional, however. Cloning will work just fine without `--depth 1`.

```
git clone https://github.com/nwchemgit/nwchem --depth 1
```

The `nwchemgit/nwchem` repository comes with a variety of input decks intended for use with the Quantum Development Kit, listed under the [QA/chem\\_library\\_tests](#) folder. For this example, we'll use the `H4` input deck:

```
cd nwchem/QA/chem_library_tests/H4  
Get-Content h4_sto6g_0.000.nw
```

The molecule in question is a system of 4 hydrogen atoms that are arranged in a certain geometry that depends on one angle, the parameter `alpha` as indicated in the name `h4_sto6g_alpha.nw` of the deck. H4 is a known [molecular benchmark](#) for computational chemistry since the 1970s. The parameter `sto6g` is indicative that the deck implements a representation with respect to a Slater-type orbital, specifically, a representation with respect to an [STO-nG basis set](#) with 6 Gaussian basis functions. This input deck furthermore contains several instructions to the NWChem Tensor Contraction Engine (TCE) that direct NWChem to produce the information needed for interoperating with the Quantum Development Kit:

```
...  
set tce:print_integrals T  
set tce:qorb 18  
set tce:qela 9  
set tce:qelb 9
```

## Producing and consuming Broombridge output from NWChem

You now have everything you need to produce and consume Broombridge documents. To run NWChem and produce a Broombridge document for the `h4_sto6g_0.000.nw` input deck, run `Convert-NWChemToBroombridge`:

#### NOTE

The first time you run this command, Docker will download the `nwchemorg/nwchem-qc` image for you. This may take a little while, depending on your connection speed, possibly providing an opportunity to get a cup of coffee.

```
Convert-NWChemToBroombridge h4_sto6g_0.000.nw
```

This will produce a Broombridge document called `h4_sto6g_0.000.yaml` that you can use with `Get-GateCount`:

```
Get-GateCount -Format YAML h4_sto6g_0.000.yaml
```

You should now see console output which contains resource estimation such as T-count, rotations count, CNOT count, etc. for various quantum simulation methods:

```
IntegralDataPath : C:\Users\martinro\REPOS\nwchem\qa\chem_library_tests\h4\h4_sto6g_0.000.yaml
HamiltonianName : hamiltonian_0
SpinOrbitals : 8
Method : Trotter
TCount : 0
RotationsCount : 92
CNOTCount : 520
ElapsedMilliseconds : 327

IntegralDataPath : C:\Users\martinro\REPOS\nwchem\qa\chem_library_tests\h4\h4_sto6g_0.000.yaml
HamiltonianName : hamiltonian_0
SpinOrbitals : 8
Method : Qubitization
TCount : 438
RotationsCount : 516
CNOTCount : 2150
ElapsedMilliseconds : 528

IntegralDataPath : C:\Users\martinro\REPOS\nwchem\qa\chem_library_tests\h4\h4_sto6g_0.000.yaml
HamiltonianName : hamiltonian_0
SpinOrbitals : 8
Method : Optimized Qubitization
TCount : 3540
RotationsCount : 18
CNOTCount : 7932
ElapsedMilliseconds : 721
```

There are many things to go do from here:

- Try out different predefined input decks, for example, by varying the parameter `alpha` in `h4_sto6g_alpha.nw`,
- Try modifying the decks by editing the NWChem decks directly, for example, exploring `STO-nG` models for various choices of n,
- Try other predefined NWChem input decks that are available at `nwchem/qa/chem_library_tests`,
- Try out a suite of predefined Broombridge YAML benchmarks that were generated from NWChem and are available as part of the [Microsoft/Quantum repository](#). These benchmarks include:
  - small molecules such as molecular hydrogen (H<sub>2</sub>), Beryllium (Be), Lithium hydride (LiH),
  - larger molecules such as ozone (O<sub>3</sub>), beta-carotene, cytosine, and many more.
- Try out the graphical front-end [EML Arrows](#) that features an interface to the Microsoft Quantum Development Kit.

# Producing Broombridge output from EMSL Arrows

To get started with web-based front end EMSL Arrows, navigate a browser [here](#).

## NOTE

Running EMSL Arrows in a web browser requires JavaScript to be enabled. Please refer to these [instructions](#) on how to enable JavaScript in your browser.

First, enter a molecule in the query box **Enter an esmiles, esmiles reaction, or other Arrows input**, then push the "Run Arrows" button.

You can enter many molecules by their colloquial name, such as "caffeine" instead of "1,3,7-Trimethylxanthine".

Next, click **theory{qsharp\_chem}**. This will populate the query box further with an instruction that will tell the run to export output in the Broombridge YAML format.

Now, click **Run Arrows**. Depending on the size of the input, this might take a while. Or, in case the particular model has already been computed before, it can be done extremely fast as it will only amount to a lookup in a database. In either case, you will be taken to a new page that contains a plethora of information about the particular run of NWChem against the deck specified by your input.

You can download and save the Broombridge YAML file from the section that starts with the following header:

```
+=====+  
||      Molecular Calculation      ||  
+=====+  
  
Id      = 48443  
  
NWOutput = Link to NWChem Output (download)  
  
Datafiles:  
qsharp_chem.yaml-2018-10-23-14:37:42 (download)  
...  
+
```

Click **download**, which saves a local copy with a unique file name, for example *qsharp\_chem48443.yaml* (the particular name will be different for each run). You can then further process this file as above, for example, with

```
Get-GateCount -Format YAML qsharp_chem48443.yaml
```

to get resource counts.

You might enjoy the 3D molecule builder that can be accessed from the **Arrows Entry - 3D Builder** tab on the EMSL Arrows start page. Clicking the **JSMol** 3D picture of the shown molecule will let you allow to edit it. You can move atoms around, drag atoms apart so that their inter-molecular distances change, add/remove atoms, etc. For each of these choices, once you added **theory{qsharp\_chem}** in the query box, you can then generate an instance of the Broombridge YAML schema and further explore it using the Quantum chemistry library.

# Broombridge Quantum Chemistry Schema

3/5/2021 • 2 minutes to read • [Edit Online](#)

Powerful computational chemistry software such as [NWChem](#) allows you to model a wide range of real-world chemistry problems. In order to access NWChem molecular models with the Microsoft Quantum Chemistry library, you use a [YAML](#)-based schema named **Broombridge**. The name was chosen in reference to a [landmark](#) which in some circles is celebrated as a birthplace of Hamiltonians.

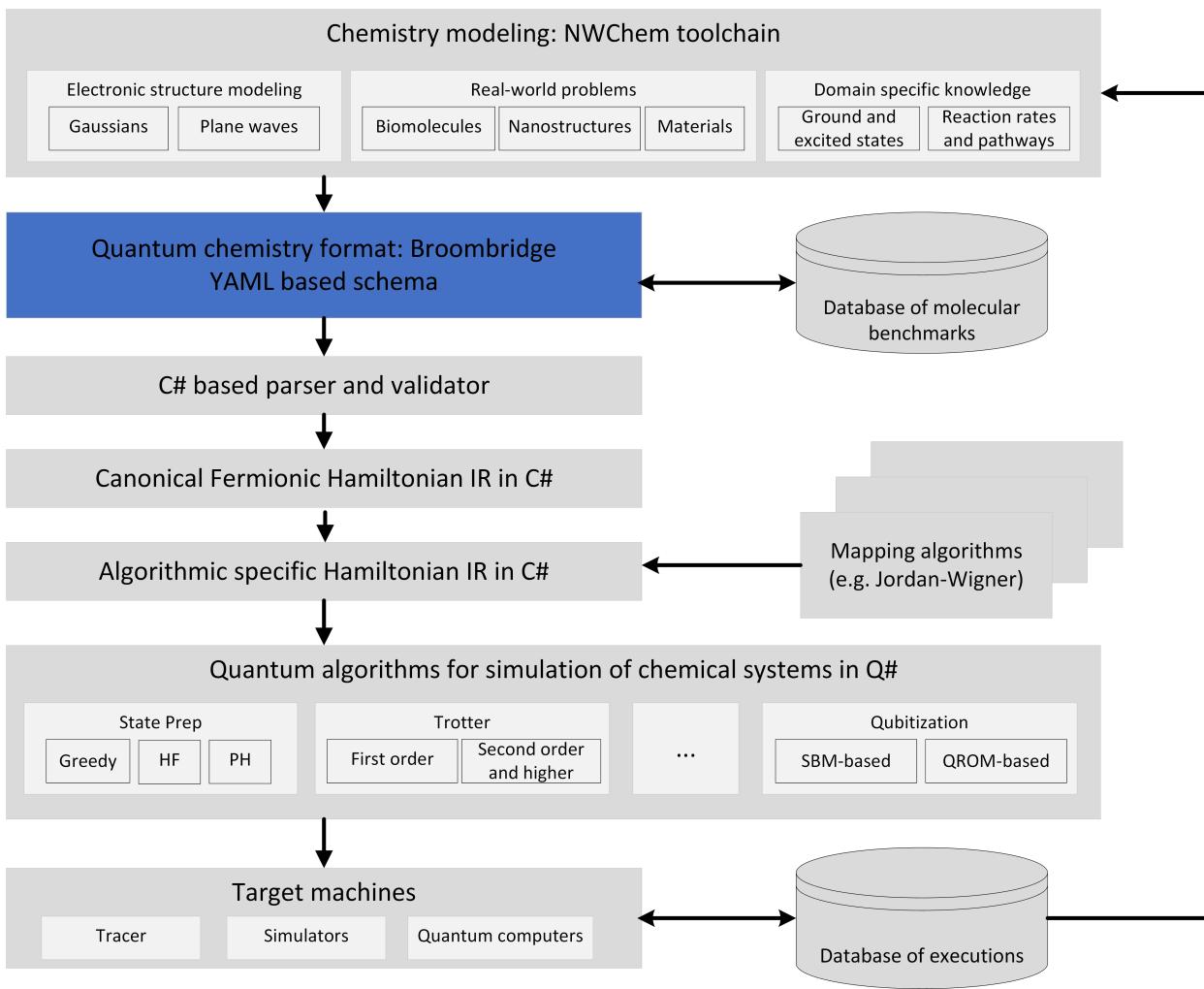
[NWChem](#) is an Open Source project licensed under the permissive Educational Community License (ECL) 2.0 license. The [Broombridge Quantum Chemistry Schema](#)) is an Open Source schema that includes a [definition](#) following [RFC 2119](#) and a [validator script](#) licensed under the MIT license.

Being YAML-based, Broombridge is a structured, human-readable and human-editable way of representing electronic structure problems. In particular, the following data can be represented:

- Fermionic Hamiltonians can be represented using one- and two-electron integrals.
- Ground and excited states can be presented using creation sequences.
- Upper and lower bounds of energy levels can be specified.

Data can be generated from NWChem using various methods, such as using a full installation of NWChem to run chemistry decks (for example the ones provided in the [NWChem library](#) that output Broombridge as part of the run), or a docker image of NWChem which can also be used to generate Broombridge from chemistry decks. To get started with computational chemistry quickly without having to install any chemistry software, you can use the visual interface to NWChem provided by [EMSL Arrows](#).

At a high level, the interplay between NWChem and the Microsoft Quantum Development Kit can be visualized as follows:



The blue shaded box represents the Broombridge schema, the various grey shaded boxes represent other internal data representations that were chosen to represent and process quantum algorithms for computational chemistry based on real-world chemistry problems.

The [Integral/YAML](#) folder in the Quantum Development Kit Samples repository contains multiple chemical representations defined using the Broombridge schema.

# Broombridge Specification v0.2

3/9/2021 • 10 minutes to read • [Edit Online](#)

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](#).

Any sidebar with the headings "NOTE," "INFORMATION," or "WARNING" is informative.

## Introduction

This section is informative.

Broombridge documents are intended to communicate instances of simulation problems in quantum chemistry for processing using quantum simulation and programming toolchains.

## Serialization

This section is normative.

A Broombridge document MUST be serialized as a [YAML 1.2 document](#) representing a JSON object as described in [RFC 4627](#) section 2.2. The object serialized to YAML MUST have a property `"$schema"` whose value is ["https://raw.githubusercontent.com/Microsoft/Quantum/master/Chemistry/Schema/qchem-0.2.schema.json"](https://raw.githubusercontent.com/Microsoft/Quantum/master/Chemistry/Schema/qchem-0.2.schema.json), and MUST be valid according to the JSON Schema draft-06 specifications [1, 2].

For the remainder of this specification, "the Broombridge object" will refer to the JSON object deserialized from a Broombridge YAML document.

Unless otherwise explicitly noted, objects MUST NOT have additional properties beyond those specified explicitly in this document.

## Additional Definitions

This section is normative.

### Quantity Objects

This section is normative.

A *quantity object* is a JSON object, and MUST have a property `units` whose value is one of the allowed values listed in Table 1.

A quantity object is a *simple quantity object* if it has a single property `value` in addition to its `units` property. The value of the `value` property MUST be a number.

A quantity object is a *bounded quantity object* if it has the properties `lower` and `upper` in addition to its `units` property. The values of the `lower` and `upper` properties MUST be numbers. A bounded quantity object MAY have a property `value` whose value is a number.

A quantity object is a *sparse array quantity object* if it has the property `format` and a property `values` in addition to its `units` property. The value of `format` MUST be the string `sparse`. The value of the `values` property MUST be an array. Each element of `values` MUST be an array representing the indices and value of the sparse array quantity.

The indices for each element of a sparse array quantity object MUST be unique across the entire sparse array

quantity object. If an index is present with a value of `0`, a parser MUST treat the sparse array quantity object identically to a sparse array quantity object in which that index is not present at all.

A quantity object MUST either be

- a simple quantity object,
- a bounded quantity object, or
- a sparse array quantity object.

## Examples

This section is informative.

A simple quantity representing  $1.9844146837$ :  
 $\text{Hartree}$ :

```
coulomb_repulsion:  
  value: 1.9844146837  
  units: hartree
```

A bounded quantity representing a uniform distribution over the interval  $[-7, -6]$ :  
 $\text{Hartree}$ :

```
fci_energy:  
  upper: -6  
  lower: -7  
  units: hartree
```

The following is a sparse array quantity with an element `[1, 2]` equal to `hello` and an element `[3, 4]` equal to `world`:

```
sparse_example:  
  format: sparse  
  units: hartree  
  values:  
    - [1, 2, "hello"]  
    - [3, 4, "world"]
```

## Format Section

This section is normative.

The Broombridge object MUST have a property `format` whose value is a JSON object with one property called `version`. The `version` property MUST have the value `"0.2"`.

## Example

This section is informative.

```
format:                      # required  
  version: "0.2"            # must match exactly
```

## Problem Description Section

This section is normative.

The Broombridge object MUST have a property `problem_description` whose value is a JSON array. Each item in the value of the `problem_description` property MUST be a JSON object describing one set of integrals, as described in the remainder of this section. In the remainder of this section, the term "problem description

object" will refer to an item in the value of the `problem_description` property of the Broombridge object.

Each problem description object MUST have a property `metadata` whose value is a JSON object. The value of `metadata` MAY be the empty JSON object (that is, `{}`), or MAY contain additional properties defined by the implementor.

## Hamiltonian Section

### Overview

This section is informative.

The `hamiltonian` property of each problem description object describes the Hamiltonian for a particular quantum chemistry problem by listing out its one- and two-body terms as sparse arrays of real numbers. The Hamiltonian operators described by each problem description object take the form

```
$$ H = \sum_{\{i,j\}} \sum_{\{\sigma\}} \in \{\uparrow, \downarrow\} h_{\{ij\}} a^{\dagger}_{\{i,\sigma\}} a_{\{j,\sigma\}} + \frac{1}{2} \sum_{\{i,j,k,l\}} \sum_{\{\sigma,\rho\}} \in \{\uparrow, \downarrow\} h_{\{ijkl\}} a^{\dagger}_{\{i,\sigma\}} a^{\dagger}_{\{k,\rho\}} a_{\{l,\rho\}} a_{\{j,\sigma\}}, $$
```

here  $h_{\{ijkl\}} = \langle ij|kl \rangle$  in Mulliken convention.

For clarity, the one-electron term is

```
$$ h_{\{ij\}} = \int \mathrm{d}x \psi^*_i(x) \left( \frac{1}{2} \nabla^2 + \sum_A \frac{Z_A}{|x-x_A|} \right) \psi_j(x), $$
```

and the two-electron term is

```
$$ h_{\{ijkl\}} = \int \mathrm{d}x^2 \psi^*_i(x_1) \psi_j(x_1) \frac{1}{|x_1 - x_2|} \psi_k^*(x_2) \psi_l(x_2). $$
```

As noted in our description of the `basis_set` property of each element of the `integral_sets` property, we further explicitly assume that the basis functions used are real-valued. This allows us to use the following symmetries between the terms to compress the representation of the Hamiltonian.

```
$$ h_{\{ijkl\}} = h_{\{ijlk\}} = h_{\{jikl\}} = h_{\{jilk\}} = h_{\{klji\}} = h_{\{kljj\}} = h_{\{lkji\}} = h_{\{llkj\}}. $$
```

#### NOTE

The term  $h_{\{ijkl\}} = \langle ij|kl \rangle$  follows Mulliken index convention, also known as chemists' notation. The representation used by the .NET and Q# data models follows Dirac or physicists' notation, where

```
$$ h_{\{ijkl\}} = \int \mathrm{d}x^2 \psi_i^*(x_1) \psi_j(x_1) \psi_k^*(x_2) \psi_l(x_2). $$
```

Broombridge schema currently only supports Mulliken indexing, such that the deserializer converts between the two conventions when loading data.

### Contents

This section is normative.

Each problem description object MUST have a property `hamiltonian` whose value is a JSON object. The value of the `hamiltonian` property is known as a Hamiltonian object, and MUST have the properties `one_electron_integrals` and `two_electron_integrals` as described in the remainder of this section.

Each problem description object MUST have a property `coulomb_repulsion` whose value is a simple quantity object. Each problem description object MUST have a property `energy_offset` whose value is a simple quantity object.

## NOTE

The values of `coulomb_repulsion` and `energy_offset` added together capture the identity term of the Hamiltonian.

### One-Electron Integrals Object

This section is normative.

The `one_electron_integrals` property of the Hamiltonian object MUST be a sparse array quantity whose indices are two integers and whose values are numbers. Every term MUST have indices `[i, j]` where `i >= j`.

## NOTE

This reflects the symmetry that  $h_{ij} = h_{ji}$  which is a consequence of the fact that the Hamiltonian is Hermitian.

### Example

This section is informative.

The following sparse array quantity represents the Hamiltonian  $H = \left( -5.0 (a^{\dagger}_{1\uparrow} a_{1\uparrow} + a^{\dagger}_{1\downarrow} a_{1\downarrow}) + 0.17 (a^{\dagger}_{2\uparrow} a_{2\uparrow} + a^{\dagger}_{2\downarrow} a_{2\downarrow}) \right) \text{Ha}$ .

```
one_electron_integrals:      # required
    units: hartree          # required
    format: sparse           # required
    values:                  # required
        # i j f(i,j)
        - [1, 1, -5.0]
        - [2, 1,  0.17]
```

## NOTE

Broombridge uses 1-based indexing.

### Two-Electron Integrals Object

This section is normative.

The `two_electron_integrals` property of the Hamiltonian object MUST be a sparse array quantity with one additional property called `index_convention`. Each element of the value of `two_electron_integrals` MUST have four indices.

Each `two_electron_integrals` property MUST have a `index_convention` property. The value of the `index_convention` property MUST be one of the allowed values listed in Table 1. If the value of `index_convention` is `mulliken`, then for each element of the `two_electron_integrals` sparse array quantity, a parser loading a Broombridge document MUST instantiate a Hamiltonian term equal to the two-electron operator  $h_{ijkl} a^{\dagger}_i a^{\dagger}_j a_k a_l$ , where  $i, j, k, l$  MUST be integers of value at least 1, and where  $h_{ijkl}$  is the element `[i, j, k, l]` of the sparse array quantity.

### Symmetries

This section is normative.

If the `index_convention` property of a `two_electron_integrals` object is equal to `mulliken`, then if an element with indices `[i, j, k, l]` is present, the following indices MUST NOT be present unless they are equal to `[i, j, k, l]`:

- `[i, j, l, k]`

- [j, i, k, l]
- [j, i, l, k]
- [k, l, i, j]
- [k, l, j, i]
- [l, k, j, i]

#### NOTE

Because the `index_convention` property is a sparse quantity object, no indices may be repeated on different elements. In particular, if an element with indices `[i, j, k, l]` is present, no other element may have those indices.

#### Example

This section is informative.

The following object specifies the Hamiltonian

```
$$ H = \frac{1}{12} \sum_{\{\sigma, \rho\}} \langle \uparrow\downarrow | \Biggr( 1.6 a^{\dagger} \sigma_1 \rho_1
a^{\dagger} \sigma_1 \rho_1 a_1 \sigma_1 - 0.1 a^{\dagger} \sigma_6 \rho_1 a_1 \rho_1 a_3 \sigma_1
a_2 \sigma_1 - 0.1 a^{\dagger} \sigma_6 \rho_1 a^{\dagger} \rho_1 a_2 \rho_1 a_3 \sigma_1 - 0.1
a^{\dagger} \sigma_1 \rho_1 a^{\dagger} \sigma_6 \rho_1 a_3 \rho_1 a_2 \sigma_1 - 0.1 a^{\dagger} \sigma_1 \rho_1 a_2 \rho_1 a_6 \sigma_1
a_1 \sigma_1 - 0.1 a^{\dagger} \sigma_3 \rho_1 a^{\dagger} \rho_2 a_1 \rho_1 a_6 \sigma_1 - 0.1
a^{\dagger} \sigma_2 \rho_1 a^{\dagger} \sigma_3 \rho_1 a_6 \rho_1 a_1 \sigma_1 - 0.1 a^{\dagger} \sigma_2 \rho_1 a_3 \rho_1 a_6 \sigma_1
a_1 \rho_1 a_6 \sigma_1 \Biggr) \text{Bigr}\langle \uparrow\downarrow |
```

```
two_electron_integrals:
  index_convention: mulliken
  units: hartree
  format: sparse
  values:
    - [1, 1, 1, 1, 1.6]
    - [6, 1, 3, 2, -0.1]
```

## Initial State Section

This section is normative.

The `initial_state_suggestion` object whose value is a JSON array specifies initial quantum states of interest to the specified Hamiltonian. Each item in the value of the `initial_state_suggestion` property MUST be a JSON object describing one quantum state, as described in the remainder of this section. In the remainder of this section, the term "state object" will refer to an item in the value of the `initial_state_suggestion` property of the Broombridge object.

#### State object

This section is normative.

Each state object MUST have a `label` property containing a string. Each state object MUST have a `method` property. The value of the `method` property MUST be one of the allowed values listed in Table 3. Each state object MAY have a property `energy` whose value MUST be a simple quantity object.

If the value of the `method` property is `sparse_multi_configurational`, the state object MUST have a `superposition` property containing an array of basis states and their unnormalized amplitudes.

For example, the initial states  $\ket{G0} = \ket{G1} = \ket{G2} = (a^{\dagger} \sigma_1 \uparrow a^{\dagger} \sigma_2 \uparrow a^{\dagger} \sigma_6 \downarrow) \ket{0}$   $\ket{E} = \frac{1}{\sqrt{0.1 + 0.2}} (a^{\dagger} \sigma_1 \uparrow a^{\dagger} \sigma_2 \uparrow a^{\dagger} \sigma_6 \downarrow)$

$(a^{\dagger})_1 \uparrow a^{\dagger}_3 \downarrow + \sqrt{0.1^2 + 0.2^2} |0\rangle$ , \$\$ where  $|\text{ket}{E}\rangle$  has energy 0.987 Hartree, are represented by

```

initial_stateSuggestions: # optional. If not provided, spin-orbitals will be filled to minimize one-body
diagonal term energies.
- label: "|G0>"
  method: sparse_multi_configurational
  superposition:
    - [1.0, "(1a)+", "(2a)+", "(2b)+", "|vacuum>"]
- label: "|G1>"
  method: sparse_multi_configurational
  superposition:
    - [-1.0, "(2a)+", "(1a)+", "(2b)+", "|vacuum>"]
- label: "|G2>"
  method: sparse_multi_configurational
  superposition:
    - [1.0, "(3a)", "(1a)+", "(2a)+", "(3a)+", "(2b)+", "|vacuum>"]
- label: "|E>"
  energy: {units: hartree, value: 0.987}
  method: sparse_multi_configurational
  superposition:
    - [0.1, "(1a)+", "(2a)+", "(2b)+", "|vacuum>"]
    - [0.2, "(1a)+", "(3a)+", "(2b)+", "|vacuum>"]

```

If the value of the `method` property is `unitary_coupled_cluster`, the state object MUST have a `cluster_operator` property whose value is a JSON object. The JSON object MUST have a `reference_state` property whose value is a basis state. The JSON object MAY have a `one_body_amplitudes` property whose value is an array of one-body cluster operators and their amplitudes. The JSON object MAY have a `two_body_amplitudes` property whose value is an array of two-body cluster operators and their amplitudes. containing an array of basis states and their unnormalized amplitudes.

For example, the state  $\langle \text{ket}{\text{reference}} | = (a^{\dagger})_1 \uparrow a^{\dagger}_2 \downarrow + \sqrt{0.1^2 + 0.2^2} |0\rangle$ , \$\$

$\langle \text{ket}{\text{UCCSD}} | = e^{T-T^{\dagger}} | \text{ket}{\text{reference}} |$ , \$\$

$T = 0.1 a^{\dagger}_3 \uparrow a_2 \downarrow + 0.2 a^{\dagger}_2 \uparrow a_3 \downarrow - 0.3 a^{\dagger}_1 \uparrow a^{\dagger}_3 \downarrow$  is represented by

```

initial_stateSuggestions: # optional. If not provided, spin-orbitals will be filled to minimize one-body
diagonal term energies.
- label: "UCCSD"
  method: unitary_coupled_cluster
  cluster_operator: # Initial state that cluster operator is applied to.
    reference_state:
      [1.0, "(1a)+", "(2a)+", "(2b)+", '|vacuum>']
  one_body_amplitudes: # A one-body cluster term is  $t^{q,p} a^{\dagger,p} a_q$ 
    - [0.1, "(3a)+", "(2b)"]
    - [-0.2, "(2a)+", "(2b)"]
  two_body_amplitudes: # A two-body unitary cluster term is  $t^{rs,qp} a^{\dagger,p} a^{\dagger,q} a_r a_s$ 
    - [-0.3, "(1a)+", "(3b)+", "(3a)", "(2b)"]

```

## Basis Set Object

This section is normative.

Each problem description object MAY have a `basis_set` property. If present, the value of the `basis_set` property MUST be an object with two properties, `type` and `name`.

The basis functions identified by the value of the `basis_set` property MUST be real-valued.

## NOTE

The assumption that all basis functions are real-valued may be relaxed in future versions of this specification.

# Tables and Lists

## Table 1. Allowed Physical Units

This section is normative.

Any string specifying a unit MUST be one of the following:

- `hartree`
- `ev`

Parsers and producers MUST treat the following simple quantity objects as equivalent:

```
- {"units": "hartree", "value": 1}
- {"units": "ev", "value": 27.2113831301723}
```

## Table 2. Allowed Index Conventions

This section is normative.

Any string specifying an index convention MUST be one of the following:

- `mulliken`

This section is informative.

Additional index conventions may be introduced in future versions of this specification.

## Interpretation of Index Conventions

This section is informative.

## Table 3. Allowed State methods

This section is normative.

Any string specifying a state method MUST be one of the following:

- `sparse_multi_configurational`
- `unitary_coupled_cluster`

This section is informative.

Additional state methods may be introduced in future versions of this specification.

# Broombridge Specification v0.1

3/5/2021 • 9 minutes to read • [Edit Online](#)

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](#).

Any sidebar with the headings "NOTE," "INFORMATION," or "WARNING" is informative.

## Introduction

This section is informative.

Broombridge documents are intended to communicate instances of simulation problems in quantum chemistry for processing using quantum simulation and programming toolchains.

## Serialization

This section is normative.

A Broombridge document MUST be serialized as a [YAML 1.2 document](#) representing a JSON object as described in [RFC 4627](#) section 2.2. The object serialized to YAML MUST have a property `"$schema"` whose value is ["https://raw.githubusercontent.com/Microsoft/Quantum/master/Chemistry/Schema/qchem-0.1.schema.json"](https://raw.githubusercontent.com/Microsoft/Quantum/master/Chemistry/Schema/qchem-0.1.schema.json), and MUST be valid according to the JSON Schema draft-06 specifications [1, 2].

For the remainder of this specification, "the Broombridge object" will refer to the JSON object deserialized from a Broombridge YAML document.

Unless otherwise explicitly noted, objects MUST NOT have additional properties beyond those specified explicitly in this document.

## Additional Definitions

This section is normative.

### Quantity Objects

This section is normative.

A *quantity object* is a JSON object, and MUST have a property `units` whose value is one of the allowed values listed in Table 1.

A quantity object is a *simple quantity object* if it has a single property `value` in addition to its `units` property. The value of the `value` property MUST be a number.

A quantity object is a *bounded quantity object* if it has the properties `lower` and `upper` in addition to its `units` property. The values of the `lower` and `upper` properties MUST be numbers. A bounded quantity object MAY have a property `value` whose value is a number.

A quantity object is a *sparse array quantity object* if it has the property `format` and a property `values` in addition to its `units` property. The value of `format` MUST be the string `sparse`. The value of the `values` property MUST be an array. Each element of `values` MUST be an array representing the indices and value of the sparse array quantity.

The indices for each element of a sparse array quantity object MUST be unique across the entire sparse array

quantity object. If an index is present with a value of `0`, a parser MUST treat the sparse array quantity object identically to a sparse array quantity object in which that index is not present at all.

A quantity object MUST either be

- a simple quantity object,
- a bounded quantity object, or
- a sparse array quantity object.

## Examples

This section is informative.

A simple quantity representing  $1.9844146837 \text{ Hartree}$ :

```
coulomb_repulsion:  
  value: 1.9844146837  
  units: hartree
```

A bounded quantity representing a uniform distribution over the interval  $[-7, -6] \text{ Hartree}$ :

```
fci_energy:  
  upper: -6  
  lower: -7  
  units: hartree
```

The following is a sparse array quantity with an element `[1, 2]` equal to `hello` and an element `[3, 4]` equal to `world`:

```
sparse_example:  
  format: sparse  
  units: hartree  
  values:  
    - [1, 2, "hello"]  
    - [3, 4, "world"]
```

## Format Section

This section is normative.

The Broombridge object MUST have a property `format` whose value is a JSON object with one property called `version`. The `version` property MUST have the value `"0.1"`.

## Example

This section is informative.

```
format:          # required  
  version: "0.1"  # must match exactly
```

## Integral Sets Section

This section is normative.

The Broombridge object MUST have a property `integral_sets` whose value is a JSON array. Each item in the value of the `integral_sets` property MUST be a JSON object describing one set of integrals, as described in the remainder of this section. In the remainder of this section, the term "integral set object" will refer to an item in

the value of the `integral_sets` property of the Broombridge object.

Each integral set object MUST have a property `metadata` whose value is a JSON object. The value of `metadata` MAY be the empty JSON object (that is, `{}`), or MAY contain additional properties defined by the implementor.

## Hamiltonian Section

### Overview

This section is informative.

The `hamiltonian` property of each integral set object describes the Hamiltonian for a particular quantum chemistry problem by listing out its one- and two-body terms as sparse arrays of real numbers. The Hamiltonian operators described by each integral set object take the form

$$\$ H = \sum_{ij} \sum_{\{\sigma\}} h_{ij} a^{\dagger}_{i\sigma} a_{j\sigma} + \frac{1}{2} \sum_{ijkl} \sum_{\{\sigma\rho\}} h_{ijkl} a^{\dagger}_{i\sigma} a^{\dagger}_{j\sigma} a_{k\rho} a_{l\rho}, \$$$

here  $h_{ijkl} = \langle ij|kl \rangle$  in Mulliken convention.

For clarity, the one-electron term is

$$\$ h_{ij} = \int d\mathbf{x} \psi_i^*(\mathbf{x}) \left( \frac{1}{2} \nabla^2 + \sum_A \frac{Z_A}{|\mathbf{x} - \mathbf{x}_A|} \right) \psi_j(\mathbf{x}), \$$$

and the two-electron term is

$$\$ h_{ijkl} = \int d\mathbf{x}^2 \psi_i^*(\mathbf{x}_1) \psi_j(\mathbf{x}_1) \frac{1}{|\mathbf{x}_1 - \mathbf{x}_2|} \psi_k^*(\mathbf{x}_2) \psi_l(\mathbf{x}_2). \$$$

As noted in our description of the `basis_set` property of each element of the `integral_sets` property, we further explicitly assume that the basis functions used are real-valued. This allows us to use the following symmetries between the terms to compress the representation of the Hamiltonian.

$$\$ h_{ijkl} = h_{ijlk} = h_{jikl} = h_{jilk} = h_{klji} = h_{klji} = h_{lkij} = h_{lkji}. \$$$

### Contents

This section is normative.

Each integral set MUST have a property `hamiltonian` whose value is a JSON object. The value of the `hamiltonian` property is known as a Hamiltonian object, and MUST have the properties `one_electron_integrals` and `two_electron_integrals` as described in the remainder of this section. A Hamiltonian object MAY also have a property `particle_hole_representation`. If present, the value of `particle_hole_representation` MUST follow the format described in the remainder of this section.

#### One-Electron Integrals Object

This section is normative.

The `one_electron_integrals` property of the Hamiltonian object MUST be a sparse array quantity whose indices are two integers and whose values are numbers. Every term MUST have indices `[i, j]` where `i >= j`.

[NOTE] This reflects the symmetry that  $h_{ij} = h_{ji}$  which is a consequence of the fact that the Hamiltonian is Hermitian.

#### Example

This section is informative.

The following sparse array quantity represents the Hamiltonian  $\$ H = -5.0 (a^{\dagger}_{1\uparrow} a_{1\uparrow} + a^{\dagger}_{1\downarrow} a_{1\downarrow}) + 0.17 (a^{\dagger}_{2\uparrow} a_{2\uparrow} + a^{\dagger}_{2\downarrow} a_{2\downarrow}) + a^{\dagger}_{1\downarrow} a_{2\uparrow} + a^{\dagger}_{2\uparrow} a_{1\downarrow}) \$$

```

one_electron_integrals:      # required
  units: hartree            # required
  format: sparse             # required
  values:                   # required
    # i j f(i,j)
    - [1, 1, -5.0]
    - [2, 1,  0.17]

```

### NOTE

Broombridge uses 1-based indexing.

#### Two-Electron Integrals Object

This section is normative.

The `two_electron_integrals` property of the Hamiltonian object MUST be a sparse array quantity with one additional property called `index_convention`. Each element of the value of `two_electron_integrals` MUST have four indices.

Each `two_electron_integrals` property MUST have a `index_convention` property. The value of the `index_convention` property MUST be one of the allowed values listed in Table 1. If the value of `index_convention` is `mulliken`, then for each element of the `two_electron_integrals` sparse array quantity, a parser loading a Broombridge document MUST instantiate a Hamiltonian term equal to the two-electron operator  $\$h_{i,j,k,l} a^{\dagger}_i a^{\dagger}_j a_k a_l$ , where  $i$ ,  $j$ ,  $k$ , and  $l$  MUST be integers in the inclusive range from 1 to the number of electrons specified by the `n_electrons` property of the integral set object, and where  $\$h_{i,j,k,l}$  is the element `[i, j, k, l, h(i, j, k, l)]` of the sparse array quantity.

#### Symmetries

This section is normative.

If the `index_convention` property of a `two_electron_integrals` object is equal to `mulliken`, then if an element with indices `[i, j, k, l]` is present, the following indices MUST NOT be present unless they are equal to `[i, j, k, l]`:

- `[i, j, l, k]`
- `[j, i, k, l]`
- `[j, i, l, k]`
- `[k, l, i, j]`
- `[k, l, j, i]`
- `[l, k, j, i]`

### NOTE

Because the `index_convention` property is a sparse quantity object, no indices may be repeated on different elements. In particular, if an element with indices `[i, j, k, l]` is present, no other element may have those indices.

#### Example

This section is informative.

The following object specifies the Hamiltonian

$$\begin{aligned}
 \$\$ H = & \frac{1}{2} \sum_{\sigma} \rho \in (\uparrow \downarrow) \Bigg( 1.6 a^{\dagger}_{1,\sigma} \\
 & a^{\dagger}_{1,\rho} a_{1,\sigma} - 0.1 a^{\dagger}_{6,\sigma} a^{\dagger}_{1,\rho} a_{3,\rho} \\
 & a_{2,\sigma} - 0.1 a^{\dagger}_{6,\sigma} a^{\dagger}_{1,\rho} a_{2,\rho} a_{3,\sigma} - 0.1 \\
 & a^{\dagger}_{1,\sigma} a^{\dagger}_{6,\rho} a_{3,\rho} a_{2,\sigma} - 0.1 a^{\dagger}_{1,\sigma}
 \end{aligned}$$

```

a^{\dagger}_{6,\rho} a_{2,\rho} a_{3,\sigma} $$$ - 0.1 a^{\dagger}_{3,\sigma} a^{\dagger}_{2,\rho} a_{6,\rho}
a_{1,\sigma} - 0.1 a^{\dagger}_{3,\sigma} a^{\dagger}_{2,\rho} a_{1,\rho} a_{6,\sigma} - 0.1
a^{\dagger}_{2,\sigma} a^{\dagger}_{3,\rho} a_{6,\rho} a_{1,\sigma} - 0.1 a^{\dagger}_{2,\sigma}
a^{\dagger}_{3,\rho} a_{1,\rho} a_{6,\sigma}\Biggr)\text{textrm{Ha}}. $$
```

```

two_electron_integrals:
  index_convention: mulliken
  units: hartree
  format: sparse
  values:
    - [1, 1, 1, 1, 1.6]
    - [6, 1, 3, 2, -0.1]
```

#### Particle–Hole Representation Object

This section is normative.

The particle–hole representation object specifies that the integrals stored are defined with respect to particle hole representation wherein the creation and annihilation operators describe excitations away from the reference state used, such as a Hartree–Fock state. The object is OPTIONAL. If the object is not specified then the Hamiltonian is to be interpreted as not given in particle-hole representation. If present, the value of `particle_hole_representation` MUST be a sparse array quantity object whose indices are four integers, and whose values are a number and a string. The string portion of the value of each element MUST contain only the characters `'+'` and `'-'` which specifies whether a given factor in the term is a creation or annihilation operator in the particle–hole representation. For example `"-+++"` corresponds to a hole being created at site `$i$` and particles being created at sites `$j,k$` and `$l$`.

#### NOTE

As the value of the `particle_hole_representation` is a sparse array quantity object, the `unit` and `format` properties must be specified. Acceptable units include are listed in Table 1. The `format` property is required, and indicates whether the Hamiltonian coefficients are specified as a dense or sparse array. In the current version, only sparse arrays are supported, with interpretation that all unspecified elements are `$0$`, but future versions may add support for additional values of the `format` property.

## Initial State Section

The `initial_state_suggestion` object specifies initial quantum states of interest to the specified Hamiltonian. This object must be an array of JSON `state` objects.

#### State object

Each states represents a superposition of occupied orbitals. Each state object MUST have a `label` property containing a string. Each state object MUST have a `superposition` property containing an array of basis states and their unnormalized amplitudes.

For example, the initial states  $\ket{G0} = \ket{G1} = \ket{G2} =$   
 $(a^{\dagger}_{1,\uparrow} a^{\dagger}_{2,\uparrow} a^{\dagger}_{3,\uparrow}) \ket{0}$   $\ket{E} = \frac{0.1}{\sqrt{0.1^2 + 0.2^2}} \ket{0}$  are represented by

```

initial_stateSuggestions: # optional. If not provided, spin-orbitals will be filled to minimize one-body
diagonal term energies.
  - state:
    label: "|G0>"
    superposition:
      - [1.0, "(1a)+", "(2a)+", "(2b)+", "|vacuum>"]
  - state:
    label: "|G1>"
    superposition:
      - [-1.0, "(2a)+", "(1a)+", "(2b)+", "|vacuum>"]
  - state:
    label: "|G2>"
    superposition:
      - [1.0, "(3a)", "(1a)+", "(2a)+", "(3a)+", "(2b)+", "|vacuum>"]
  - state:
    label: "|E>"
    superposition:
      - [0.1, "(1a)+", "(2a)+", "(2b)+", "|vacuum>"]
      - [0.2, "(1a)+", "(3a)+", "(2b)+", "|vacuum>"]

```

### Basis Set Object

This section is normative.

Each integral set object MAY have a `basis_set` property. If present, the value of the `basis_set` property MUST be an object with two properties, `type` and `name`.

The basis functions identified by the value of the `basis_set` property MUST be real-valued.

#### NOTE

The assumption that all basis functions are real-valued may be relaxed in future versions of this specification.

## Tables and Lists

### Table 1. Allowed Physical Units

This section is normative.

Any string specifying a unit MUST be one of the following:

- `hartree`
- `ev`

Parsers and producers MUST treat the following simple quantity objects as equivalent:

```

- {"units": "hartree", "value": 1}
- {"units": "ev", "value": 27.2113831301723}

```

### Table 2. Allowed Index Conventions

This section is normative.

Any string specifying an index convention MUST be one of the following:

- `mulliken`

This section is informative.

Additional index conventions may be introduced in future versions of this specification.

#### Interpretation of Index Conventions

This section is informative.

# Introduction to the Quantum Machine Learning Library

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Quantum Machine Learning Library is an API, written in Q#, that gives you the ability to run hybrid quantum/classical machine learning experiments. The library gives you the ability to:

- Load your own data to classify with quantum simulators
- Use samples and tutorials to get introduced to the field of quantum machine learning

You can expect low performance compared to current classical machine learning frameworks (remember that everything is running on top of the simulation of a quantum device that is already computationally expensive).

The purpose of this documentation is:

- A concise introduction to machine learning tools (written in Q#) for hybrid quantum/classical learning.
- Introduce machine learning concepts and specifically their realization in quantum circuit centric classifiers (also known as quantum sequential classifiers).
- Provide a set of tutorials on the basics to start using the tools provided by the library.
- Discuss the training and validation methods for such classifiers and how they translate into specific Q# operations provided by the library.

The model implemented in this library is based on the quantum-classical training scheme presented in [Circuit-centric quantum classifiers](#)

# Introduction to Quantum Machine Learning

3/5/2021 • 5 minutes to read • [Edit Online](#)

## Framework and goals

Quantum encoding and processing of information is a powerful alternative to classical machine learning Quantum classifiers. In particular, it allows us to encode data in quantum registers that are concise relative to the number of features, systematically employing quantum entanglement as computational resource and employing quantum measurement for class inference. Circuit centric quantum classifier is a relatively simple quantum solution that combines data encoding with a rapidly entangling/disentangling quantum circuit followed by measurement to infer class labels of data samples. The goal is to ensure classical characterization and storage of subject circuits, as well as hybrid quantum/classical training of the circuit parameters even for extremely large feature spaces.

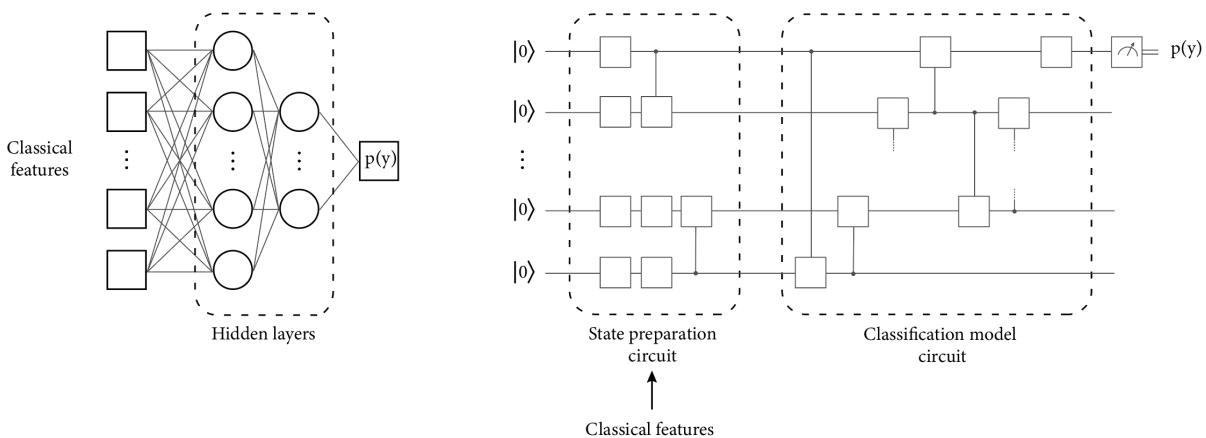
## Classifier architecture

Classification is a supervised machine learning task, where the goal is to infer class labels  $\{y_1, y_2, \dots, y_d\}$  of certain data samples. The "training data set" is a collection of samples  $\mathcal{D} = \{(x, y)\}$  with known pre-assigned labels. Here  $x$  is a data sample and  $y$  is its known label called "training label". Somewhat similar to traditional methods, quantum classification consists of three steps:

- data encoding
- preparation of a classifier state
- measurement Due to the probabilistic nature of the measurement, these three steps must be repeated multiple times. Both the encoding and the computing of the classifier state are done by means of *quantum circuits*. While the encoding circuit is usually data-driven and parameter-free, the classifier circuit contains a sufficient set of learnable parameters.

In the proposed solution the classifier circuit is composed of single-qubit rotations and two-qubit controlled rotations. The learnable parameters here are the rotation angles. The rotation and controlled rotation gates are known to be *universal* for quantum computation, which means that any unitary weight matrix can be decomposed into a long enough circuit consisting of such gates.

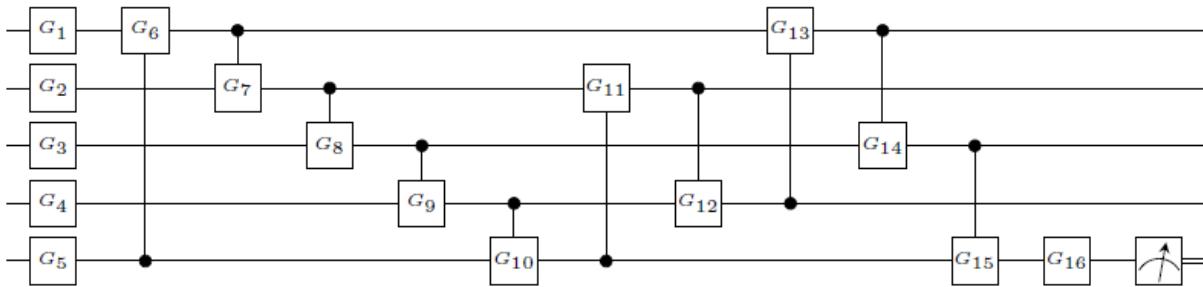
In the proposed version, only one circuit followed by a single frequency estimation is supported. Thus, the solution is a quantum analog of a support vector machine with a low-degree polynomial kernel.



A simple quantum classifier design can be compared to a traditional support vector machine (SVM) solution. The inference for a data sample  $x$  in case of SVM is done using an optimal kernel form  $\sum \alpha_j k(x_j, x)$  where  $k$  is a certain kernel function.

By contrast, a quantum classifier uses the predictor  $p(y | x, U(\theta)) = \text{Tr}(U(\theta)x|M|U(\theta)x)$ , which is similar in spirit but technically quite different. Thus, when a straightforward amplitude encoding is used,  $p(y | x, U(\theta))$  is a quadratic form in the amplitudes of  $x$ , but the coefficients of this form are no longer learned independently; they are instead aggregated from the matrix elements of the circuit  $U(\theta)$ , which typically has significantly fewer learnable parameters  $\theta$  than the dimension of the vector  $x$ . The polynomial degree of  $p(y | x, U(\theta))$  in the original features can be increased to  $2^l$  by using a quantum product encoding on  $l$  copies of  $x$ .

Our architecture explores relatively shallow circuits, which therefore must be *rapidly entangling* in order to capture all the correlations between the data features at all ranges. An example of the most useful rapidly entangling circuit component is shown on figure below. Even though a circuit with this geometry consists of only  $3n+1$  gates, the unitary weight matrix that it computes ensures significant cross-talk between  $2^n$  features.



The circuit in the above example consists of 6 single-qubit gates ( $G_1 \dots G_5$ ;  $G_{16}$ ) and 10 two-qubits gates ( $G_6 \dots G_{15}$ ). Assuming that each of the gates is defined with one learnable parameter we have 16 learnable parameters, while the dimension of the 5-qubit Hilbert space is 32. Such circuit geometry can be easily generalized to any  $n$ -qubit register, when  $n$  is odd, yielding circuits with  $3n+1$  parameters for  $2^n$ -dimensional feature space.

## Classifier training as a supervised learning task

Training of a classifier model involves finding optimal values of its operational parameters, such that they maximize the average likelihood of inferring the correct training labels across the training samples. Here, we concern ourselves with two level classification only, for example, the case of  $d=2$  and only two classes with the labels  $y_1, y_2$ .

### NOTE

A principled way of generalizing our methods to arbitrary number of classes is to replace qubits with qudits, which are quantum units with  $d$  basis states, and the two-way measurement with  $d$ -way measurement.

### Likelihood as the training goal

Given a learnable quantum circuit  $U(\theta)$ , where  $\theta$  is a vector of parameters, and denoting the final measurement by  $M$ , the average likelihood of the correct label inference is  $\frac{1}{d} \left( \sum_{y=1}^d P(M=y | U(\theta), x) + \sum_{(x,y) \in D} P(M=y | U(\theta), x) \right)$  where  $P(M=y | z)$  is the probability of measuring  $y$  in quantum state  $z$ . Here, it suffices to understand that the likelihood function  $L(\theta)$  is smooth in  $\theta$  and its derivative in any  $\theta_j$  can be computed by essentially the same quantum protocol as used for computing the likelihood function itself. This allows for optimizing the

$\mathcal{L}(\theta)$  by gradient descent.

### Classifier bias and training score

Given some intermediate (or final) values of the parameters in  $\theta$ , we need to identify a single real value  $b$  known as *classifier bias* to do the inference. The label inference rule works as follows:

- A sample  $x$  is assigned label  $y_2$  if and only if  $P(M=y_2|U(\theta)x) + b > 0.5$  (RULE1) (otherwise it is assigned label  $y_1$ )

Clearly  $b$  must be in the interval  $(-0.5, +0.5)$  to be meaningful.

A training case  $(x, y) \in \mathcal{D}$  is considered a *misclassification* given the bias  $b$  if the label inferred for  $x$  as per RULE1 is actually different from  $y$ . The overall number of misclassifications is the *training score* of the classifier given the bias  $b$ . The *optimal classifier bias*  $b$  minimizes the training score. It is easy to see that, given the precomputed probability estimates  $\{ P(M=y_2|U(\theta)x) | (x, y) \in \mathcal{D} \}$ , the optimal classifier bias can be found by binary search in interval  $(-0.5, +0.5)$  by making at most  $\log_2(|\mathcal{D}|)$  steps.

### Reference

This information should be enough to start playing with the code. However, if you want to learn more about this model, please read the original proposal: '[Circuit-centric quantum classifiers](#)', *Maria Schuld, Alex Bocharov, Krysta Svore and Nathan Wiebe*

In addition to the code sample you will see in the next steps, you can also start exploring quantum classification in [this tutorial](#)

# Basic classification: Classify data with the QDK

5/27/2021 • 8 minutes to read • [Edit Online](#)

In this guide, you will learn how to run a quantum sequential classifier written in Q# using the Quantum Machine Learning library of the QDK. To do that, we will train a simple sequential model using a classifier structure defined in Q#. The model is trained on a half-moon dataset with training and validation data that you can find in the [code samples](#). We will create our Q# project using either a Python or a C# program to load data and call Q# operations from.

## Prerequisites

- The Microsoft [Quantum Development Kit](#).
- Create a Q# project for either a [Python host program](#) or a [C# host program](#).
- To add the [Microsoft.Quantum.MachineLearning](#) package to your Q# project, run the following command from the root of your project folder:

```
dotnet add package Microsoft.Quantum.MachineLearning
```

## Q# classifier code

We start by creating a file called `Training.qs` and adding the following code to it:

```
namespace Microsoft.Quantum.Samples {
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Arrays;
    open Microsoft.Quantum.MachineLearning;
    open Microsoft.Quantum.Math;

    function WithProductKernel(scale : Double, sample : Double[]) : Double[] {
        return sample + [scale * Fold(TimesD, 1.0, sample)];
    }

    function Preprocessed(samples : Double[][][]) : Double[][][] {
        let scale = 1.0;

        return Mapped(
            WithProductKernel(scale, _),
            samples
        );
    }

    function DefaultSchedule(samples : Double[][][]) : SamplingSchedule {
        return SamplingSchedule([
            0..Length(samples) - 1
        ]);
    }

    function ClassifierStructure() : ControlledRotation[] {
        return [
            ControlledRotation((0, new Int[0]), PauliX, 4),
            ControlledRotation((0, new Int[0]), PauliZ, 5),
            ControlledRotation((1, new Int[0]), PauliX, 6),
            ControlledRotation((1, new Int[0]), PauliZ, 7),
            ControlledRotation((0, [1]), PauliX, 0),
            ControlledRotation((0, [1]), PauliZ, 1),
            ControlledRotation((1, [1]), PauliX, 2),
            ControlledRotation((1, [1]), PauliZ, 3)
        ];
    }
}
```

```

        ControlledRotation((1, [0]), PauliX, 1),
        ControlledRotation((1, new Int[0]), PauliZ, 2),
        ControlledRotation((1, new Int[0]), PauliX, 3)
    ];
}

operation TrainHalfMoonModel(
    trainingVectors : Double[][][],
    trainingLabels : Int[],
    initialParameters : Double[][]
) : (Double[], Double) {
    let samples = Mapped(
        LabeledSample,
        Zipped(Preprocessed(trainingVectors), trainingLabels)
    );
    Message("Ready to train.");
    let (optimizedModel, nMisses) = TrainSequentialClassifier(
        Mapped(
            SequentialModel(ClassifierStructure(), _, 0.0),
            initialParameters
        ),
        samples,
        DefaultTrainingOptions()
        w/ LearningRate <- 0.1
        w/ MinibatchSize <- 15
        w/ Tolerance <- 0.005
        w/ NMeasurements <- 10000
        w/ MaxEpochs <- 16
        w/ VerboseMessage <- Message,
        DefaultSchedule(trainingVectors),
        DefaultSchedule(trainingVectors)
    );
    Message($"Training complete, found optimal parameters: {optimizedModel::Parameters}");
    return (optimizedModel::Parameters, optimizedModel::Bias);
}

operation ValidateHalfMoonModel(
    validationVectors : Double[][][],
    validationLabels : Int[],
    parameters : Double[],
    bias : Double
) : Double {
    let samples = Mapped(
        LabeledSample,
        Zipped(Preprocessed(validationVectors), validationLabels)
    );
    let tolerance = 0.005;
    let nMeasurements = 10000;
    let results = ValidateSequentialClassifier(
        SequentialModel(ClassifierStructure(), parameters, bias),
        samples,
        tolerance,
        nMeasurements,
        DefaultSchedule(validationVectors)
    );
    return IntAsDouble(results::NMisclassifications) / IntAsDouble(Length(samples));
}

operation ClassifyHalfMoonModel(
    samples : Double[][][],
    parameters : Double[],
    bias : Double,
    tolerance : Double,
    nMeasurements : Int
)
: Int[] {
    let model = Default<SequentialModel>()
    w/ Structure <- ClassifierStructure()
    // Parameters <- parameters
}

```

```

    w/ Parameters <- parameters
    w/ Bias <- bias;
    let features = Preprocessed(samples);
    let probabilities = EstimateClassificationProbabilities(
        tolerance, model,
        features, nMeasurements
    );
    return InferredLabels(model::Bias, probabilities);
}

}

```

The most important functions and operations defined in the code above are:

- `ClassifierStructure() : ControlledRotation[]` : in this function we set the structure of our circuit model by adding the layers of the controlled gates we consider. This step is analogous to the declaration of layers of neurons in a sequential deep learning model.
- `TrainHalfMoonModel() : (Double[], Double)` : this operation is the core part of the code and defines the training. Here we load the samples from the dataset included in the library, we set the hyper parameters and the initial parameters for the training and we start the training by calling the operation `TrainSequentialClassifier` included in the library. It outputs the parameters and the bias that determine the classifier.
- `ValidateHalfMoonModel(parameters : Double[], bias : Double) : Int` : this operation defines the validation process to evaluate the model. Here we load the samples for validation, the number of measurements per sample and the tolerance. It outputs the number of misclassifications on the chosen batch of samples for validation.

## Host program

Next, in the same folder we create a host program. Your host program consists of three parts:

- Load the dataset `data.json` and choose a set of classifier parameters where you want to start your training iterations for your model.
- Run training to determine the parameters and bias of the model.
- After training, validate the model to determine its accuracy.
  - [Python with Visual Studio Code or the Command Line](#)
  - [C# with Visual Studio Code or the Command Line](#)
  - [C# with Visual Studio 2019](#)

To run your the Q# classifier from Python, save the following code as `host.py`. Remember that you also need the Q# file `Training.qs` that is explained above in this tutorial.

```

import json

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.cm as cmx
plt.style.use('ggplot')

import qsharp

from Microsoft.Quantum.Samples import (
    TrainHalfMoonModel, ValidateHalfMoonModel, ClassifyHalfMoonModel
)

```

```

if __name__ == "__main__":
    with open('data.json') as f:
        data = json.load(f)
    parameter_starting_points = [
        [0.060057, 3.00522, 2.03083, 0.63527, 1.03771, 1.27881, 4.10186, 5.34396],
        [0.586514, 3.371623, 0.860791, 2.92517, 1.14616, 2.99776, 2.26505, 5.62137],
        [1.69704, 1.13912, 2.3595, 4.037552, 1.63698, 1.27549, 0.328671, 0.302282],
        [5.21662, 6.04363, 0.224184, 1.53913, 1.64524, 4.79508, 1.49742, 1.545]
    ]

    (parameters, bias) = TrainHalfMoonModel.simulate(
        trainingVectors=data['TrainingData']['Features'],
        trainingLabels=data['TrainingData']['Labels'],
        initialParameters=parameter_starting_points
    )

    miss_rate = ValidateHalfMoonModel.simulate(
        validationVectors=data['ValidationData']['Features'],
        validationLabels=data['ValidationData']['Labels'],
        parameters=parameters, bias=bias
    )

    print(f"Miss rate: {miss_rate:0.2%}")

    # Classify the validation so that we can plot it.
    actual_labels = data['ValidationData']['Labels']
    classified_labels = ClassifyHalfMoonModel.simulate(
        samples=data['ValidationData']['Features'],
        parameters=parameters, bias=bias,
        tolerance=0.005, nMeasurements=10_000
    )

# To plot samples, it's helpful to have colors for each.
# We'll plot four cases:
# - actually 0, classified as 0
# - actually 0, classified as 1
# - actually 1, classified as 1
# - actually 1, classified as 0
cases = [
    (0, 0),
    (0, 1),
    (1, 1),
    (1, 0)
]
# We can use these cases to define markers and colormaps for plotting.
markers = [
    '.' if actual == classified else 'x'
    for (actual, classified) in cases
]
colormap = cmx.ScalarMappable(colors.Normalize(vmin=0, vmax=len(cases) - 1))
colors = [colormap.to_rgba(idx_case) for (idx_case, case) in enumerate(cases)]

# It's also really helpful to have the samples as a NumPy array so that we
# can find masks for each of the four cases.
samples = np.array(data['ValidationData']['Features'])

# Finally, we loop over the cases above and plot the samples that match
# each.
for (idx_case, ((actual, classified), marker, color)) in enumerate(zip(cases, markers, colors)):
    mask = np.logical_and(
        np.equal(actual_labels, actual),
        np.equal(classified_labels, classified)
    )
    if not np.any(mask):
        continue
    plt.scatter(
        samples[mask, 0],
        samples[mask, 1],

```

```
c=[color],  
label=f"Was {actual}, classified {classified}",  
marker=marker  
)  
plt.legend()  
plt.show()
```

You can then run your Python host program from the command line:

```
$ python host.py
```

```
Preparing Q# environment...  
[...]  
Observed X.XX% misclassifications.
```

## Next steps

First, you can play with the code and try to change some parameters to see how it affects the training. Then, in the next tutorial, [Design your own classifier](#), you will learn how to define the structure of the classifier.

# Design your own classifier

5/27/2021 • 2 minutes to read • [Edit Online](#)

In this guide you will learn the basic concepts behind the design of circuit models for the quantum circuit centric classifier.

As in classical deep learning, there is no general rule for choosing a specific architecture. Depending on the problem some architectures will perform better than others. But, there are some concepts that might be useful when designing the circuit:

- A large number of parameters implies a more flexible model, which may be suitable to draw complicated classification boundaries but which may also be more susceptible to overfitting.
- Entangling gates between qubits are essential to capture the correlations between the quantum features.

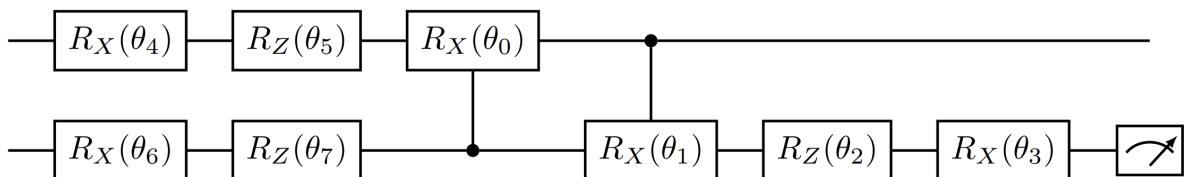
## How to build a classifier with Q#

To build a classifier we are going to concatenate parametrized controlled rotations in our circuit model. To do it we can use the type `ControlledRotation` defined in the Quantum Machine Learning library. This type accepts four arguments that determine: the index of the target qubit, the array of indices of the control qubits, the axis of rotation, and index of the associated parameter in the array of parameters defining the model.

Let's see an example of a classifier. In the [half-moons sample](#), we can find the following classifier defined in the file `Training.qs`.

```
function ClassifierStructure() : ControlledRotation[] {
    return [
        ControlledRotation((0, new Int[0]), Paulix, 4),
        ControlledRotation((0, new Int[0]), Pauliz, 5),
        ControlledRotation((1, new Int[0]), Paulix, 6),
        ControlledRotation((1, new Int[0]), Pauliz, 7),
        ControlledRotation((0, [1]), Paulix, 0),
        ControlledRotation((1, [0]), Paulix, 1),
        ControlledRotation((1, new Int[0]), Pauliz, 2),
        ControlledRotation((1, new Int[0]), Paulix, 3)
    ];
}
```

What we are defining here is a function that returns an array of `ControlledRotation` elements, that together with an array of parameters and a bias will define our `SequentialModel`. This type is fundamental in the Quantum Machine Learning library and defines the classifier. The circuit defined in the function above is part of a classifier in which each sample of the dataset contains two features. Therefore we only need two qubits. The graphical representation of the circuit is:

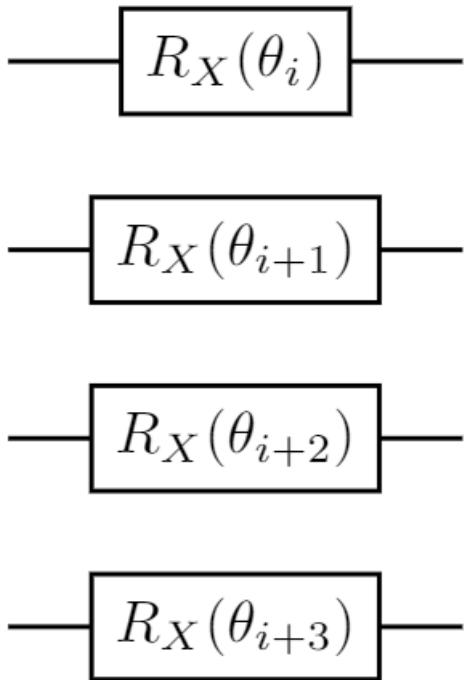


Note that by default the operations of the Quantum Machine Learning library measure the last qubit of the register to estimate the classification probabilities. You should keep in mind this fact when designing your

circuit.

## Use the library functions to write layers of gates

Suppose we have a dataset with 784 features per instance, for example, images of  $28 \times 28$  pixels like the MNIST dataset. In this case, the width of the circuit becomes large enough so that writing by hand each individual gate becomes a possible but impractical task. This is why the Quantum Machine Learning library provides a set of tools to automatically generate layers of parametrized rotations. For instance, the function `LocalRotationsLayer` returns an array of uncontrolled single-qubit rotations along a given axis, with one rotation for each qubit in the register, each parametrized by a different model parameter. For example, `LocalRotationsLayer(4, x)` returns the following set of gates:



We recommend you explore the [API reference of the Quantum Machine Learning library](#) to discover all the tools available to streamline the circuit design.

## Next steps

Try different structures in the examples provided by the samples. Do you see any changes in the performance of the model? In the next tutorial, [Load your own datasets](#), you will learn how to load datasets to try and explore new architectures of classifiers.

# Load and classify your own datasets

5/27/2021 • 2 minutes to read • [Edit Online](#)

In this short tutorial, we are going to learn how to load your own dataset to train a classifier model with the Quantum Development Kit (QDK).

We highly recommend the use of a standardized serialization format such as [JSON files](#) to store your data. Such formats offer high compatibility with different frameworks like Python and the .NET ecosystem. In particular, we recommend using our template for loading the data, so that you can copy-paste the code directly from the samples.

## Template for loading your datasets

Suppose we have a training dataset \$(x, y)\$ of size \$N=2\$ where each instance \$x\_i\$ of \$x\$ has three features: \$x\_{i1}\$, \$x\_{i2}\$ and \$x\_{i3}\$. The validation dataset has the same structure. These datasets can be represented by a `data.json` file similar to the following:

```
{
    "TrainingData": {
        "Features": [
            [
                x_11,
                x_12,
                x_13
            ],
            [
                x_21,
                x_22,
                x_23
            ]
        ],
        "Labels": [
            y_1,
            y_2
        ]
    },
    "ValidationData": {
        "Features": [
            [
                xv_11,
                xv_12,
                xv_13
            ],
            [
                xv_21,
                xv_22,
                xv_23
            ]
        ],
        "Labels": [
            yv_1,
            yv_2
        ]
    }
}
```

### Example using the template

Suppose we have a small dataset with the heights and weights of different cats and dogs. This dataset is very small to train a model but will be enough to show the process of loading a dataset.

HEIGHT (M)	WEIGHT (KG)	ANIMAL
0.54	30	Dog
0.30	8	Cat
0.91	44	Dog
0.86	31	Dog
0.32	5	Cat
0.25	4	Cat

The process is:

- First we need to separate the dataset into training and validation. In this case we can just take the first three samples for training and the rest of samples for validation. In general it is a good practice to sample randomly the training and validation dataset to avoid unwanted biases in the training data.
- Secondly, we need to assign a numeric label to each class. Note that, for the moment, the QML library only admits binary classification problems. So we will assign the label 0 to the class `Dog` and the number 1 to the class `Cat`.
- Finally, we fill the template using the data from our dataset. Note that for big datasets you should build a small script to automatically generate the template from your specific dataset. This script will depend on the original format of your dataset.

For our dataset the `data.json` file is:

```
{  
    "TrainingData": {  
        "Features": [  
            [  
                0.54,  
                30  
            ],  
            [  
                0.30,  
                8  
            ],  
            [  
                0.91,  
                44  
            ]  
        ],  
        "Labels": [  
            0,  
            1,  
            0  
        ]  
    },  
    "ValidationData": {  
        "Features": [  
            [  
                0.86,  
                31  
            ],  
            [  
                0.32,  
                5  
            ],  
            [  
                0.25,  
                4  
            ]  
        ],  
        "Labels": [  
            0,  
            1,  
            1  
        ]  
    }  
}
```

## Loading the data

Once you have your data serialized as a JSON file, you can load it in using JSON libraries provided with your host language of choice.

- [Python](#)
- [C#](#)

Python provides the [built-in json package](#) for working with JSON-serialized data:

```
import json  
  
data = json.load(f)  
parameter_starting_points = [  
    [0.060057, 3.00522, 2.03083, 0.63527, 1.03771, 1.27881, 4.10186, 5.34396],
```

## Next steps

Now you are ready to start running your own experiments with your own datasets. Try different classifiers and dataset and contribute to the community sharing your results!

# Quantum Machine Learning glossary

3/5/2021 • 3 minutes to read • [Edit Online](#)

Training of a circuit-centric quantum classifier is a process with many moving parts that require the same (or slightly larger) amount of calibration by trial and error as training of traditional classifiers. Here we define the main concepts and ingredients of this training process.

## Training/testing schedules

In the context of classifier training a *schedule* describes a subset of data samples in an overall training or testing set. A schedule is usually defined as a collection of sample indices.

## Parameter/bias scores

Given a candidate parameter vector and a classifier bias, their *validation score* is measured relative to a chosen validation schedule S and is expressed by a number of misclassifications counted over all the samples in the schedule S.

## Hyperparameters

The model training process is governed by certain pre-set values called *hyperparameters*.

### Learning rate

It is one of the key hyperparameters. It defines how much current stochastic gradient estimate impacts the parameter update. The size of parameter update delta is proportional to the learning rate. Smaller learning rate values lead to slower parameter evolution and slower convergence, but excessively large values of LR may break the convergence altogether as the gradient descent never commits to a particular local minimum. While learning rate is adaptively adjusted by the training algorithm to some extent, selecting a good initial value for it is important. A usual default initial value for learning rate is 0.1. Selecting the best value of learning rate is a fine art (see, for example, section 4.3 of Goodfellow et al., "Deep learning", MIT Press, 2017).

### Minibatch size

Defines how many data samples is used for a single estimation of stochastic gradient. Larger values of minibatch size generally lead to more robust and more monotonic convergence but can potentially slow down the training process, as the cost of any one gradient estimation is proportional to the minimatch size. A usual default value for the minibatch size is 10.

### Training epochs, tolerance, gridlocks

"Epoch" means one complete pass through the scheduled training data. The maximum number of epochs per a training thread (see below) should be capped. The training thread is defined to terminate (with the best known candidate parameters) when the maximum number of epochs has been run. However such training would terminate earlier when misclassification rate on validation schedule falls below a chosen tolerance. Suppose, for example, that misclassification tolerance is 0.01 (1%); if on validation set of 2000 samples we are seeing fewer than 20 misclassifications, then the tolerance level has been achieved. A training thread also terminates prematurely if the validation score of the candidate model has not shown any improvement over several consecutive epochs (a gridlock). The logic for the gridlock termination is currently hardcoded.

### Measurements count

Estimating the training/validation scores and the components of the stochastic gradient on a quantum device amounts to estimating quantum state overlaps that requires multiple measurements of the appropriate

observables. The number of measurements should scale as  $O(1/\epsilon^2)$  where  $\epsilon$  is the desired estimation error. As a rule of thumb, the initial measurements count could be approximately  $1/\text{tolerance}^2$  (see definition of tolerance in the previous paragraph). One would need to revise the measurement count upward if the gradient descent appears to be too erratic and convergence too hard to achieve.

## Training threads

The likelihood function which is the training utility for the classifier is very seldom convex, meaning that it usually has a multitude of local optima in the parameter space that may differ significantly by quality. Since the SGD process can converge to only one specific optimum, it is important to explore multiple starting parameter vectors. Common practice in machine learning is to initialize such starting vectors randomly. The Q# training API accepts an arbitrary array of such starting vectors but the underlying code explores them sequentially. On a multicore computer or in fact on any parallel computing architecture it is advisable to perform several calls to Q# training API in parallel with different parameter initializations across the calls.

## How to modify the hyperparameters

In the QML library, the best way to modify the hyperparameters is by overriding the default values of the UDT `TrainingOptions`. To do this we call it with the function `DefaultTrainingOptions` and apply the operator `w/` to override the default values. For example, to use 100,000 measurements and a learning rate of 0.01:

```
let options = DefaultTrainingOptions()
w/ LearningRate <- 0.01
w/ NMeasurements <- 100000;
```

# Introduction to the Quantum Numerics Library

3/5/2021 • 2 minutes to read • [Edit Online](#)

Many quantum algorithms rely on [oracles](#) that evaluate mathematical functions on a superposition of inputs. The main component of Shor's algorithm, for example, evaluates  $f(x) = a^x \operatorname{mod} N$  for a fixed  $a$ , the number to factor  $N$ , and  $x$  a  $2n$ -qubit integer in a uniform superposition over all  $2n$ -bit strings.

To run Shor's algorithm on an actual quantum computer, this function has to be written in terms of the native operations of the target machine. Using the binary representation of  $x$  with  $x_i$  denoting the  $i$ -th bit counting from the least-significant bit,  $f(x)$  can be written as  $f(x) = a^{\sum_{i=0}^{2n-1} x_i 2^i} \operatorname{mod} N$ . In turn, this can be written as a product (mod  $N$ ) of terms  $a^{2^i x_i} = (a^{2^i})^{x_i}$ . The function  $f(x)$  can thus be implemented using a sequence of  $2n$  (modular) multiplications by  $a^{2^i}$  conditional on  $x_i$  being nonzero. The constants  $a^{2^i}$  can be precomputed and reduced modulo  $N$  before running the algorithm.

This sequence of controlled modular multiplications can be simplified further: Each multiplication can be performed using a sequence of  $n$  controlled modular additions; and each modular addition can be built from a regular addition and a comparator.

Given that so many steps are necessary to arrive at an actual implementation, it would be extremely helpful to have such functionality available from the start. This is why the Quantum Development Kit provides support for a wide range of numerics functionality.

## Functionality

Besides the integer arithmetic mentioned thus far, the numerics library provides

- (Un)signed integer functionality (multiply, square, division with remainder, inversion, ...) with one or two quantum integer numbers as input
- Fixed-point functionality (add / subtract, multiply, square,  $1/x$ , polynomial evaluation) with one or two quantum fixed-point numbers as input

## Getting started

[Learn about using the numerics library](#)

# Using the Numerics library

3/5/2021 • 2 minutes to read • [Edit Online](#)

## Overview

The Numerics library consists of three components

1. **Basic integer arithmetic** with integer adders and comparators
2. **High-level integer functionality** that is built on top of the basic functionality; it includes multiplication, division, inversion, etc. for signed and unsigned integers.
3. **Fixed-point arithmetic functionality** with fixed-point initialization, addition, multiplication, reciprocal, polynomial evaluation, and measurement.

All of these components can be accessed using a single `open` statement:

```
open Microsoft.Quantum.Arithmetic;
```

## Types

The numerics library supports the following types

1. `LittleEndian`: A qubit array `qArr : Qubit[]` that represents an integer where `qArr[0]` denotes the least significant bit.
2. `SignedLittleEndian`: Same as `LittleEndian` except that it represents a signed integer stored in two's complement.
3. `FixedPoint`: Represents a real number consisting of a qubit array `qArr2 : Qubit[]` and a binary point position `pos`, which counts the number of binary digits to the left of the binary point. `qArr2` is stored in the same way as `SignedLittleEndian`.

## Operations

For each of the three types above, a variety of operations is available:

1. `LittleEndian`
  - Addition
  - Comparison
  - Multiplication
  - Squaring
  - Division (with remainder)
2. `SignedLittleEndian`
  - Addition
  - Comparison
  - Inversion modulo 2's complement
  - Multiplication
  - Squaring
3. `FixedPoint`

- Preparation / initialization to a classical values
- Addition (classical constant or other quantum fixed-point)
- Comparison
- Multiplication
- Squaring
- Polynomial evaluation with specialization for even and odd functions
- Reciprocal ( $1/x$ )
- Measurement (classical Double)

For more information and detailed documentation for each of these operations, see the Q# library reference docs at [docs.microsoft.com](https://docs.microsoft.com).

## Sample: Integer addition

As a basic example, consider the operation  $\ket{x}\ket{y} \mapsto \ket{x}\ket{x+y}$  that is, an operation that takes an  $n$ -qubit integer  $x$  and an  $n$ - or  $(n+1)$ -qubit register  $y$  as input, the latter of which it maps to the sum  $(x+y)$ . Note that the sum is computed modulo  $2^n$  if  $y$  is stored in an  $n$ -bit register.

Using the Quantum Development Kit, this operation can be applied as follows:

```
operation TestMyAddition(xValue : Int, yValue : Int, n : Int) : Unit {
    use (xQubits, yQubits) = (Qubit[n], Qubit[n]);
    let x = LittleEndian(xQubits); // define bit order
    let y = LittleEndian(yQubits);

    ApplyXorInPlace(xValue, x); // initialize values
    ApplyXorInPlace(yValue, y);

    AddI(x, y); // perform addition x+y into y

    // ... (use the result)
}
```

## Sample: Evaluating smooth functions

To evaluate smooth functions such as  $\sin(x)$  on a quantum computer, where  $x$  is a quantum `FixedPoint` number, the Quantum Development Kit numerics library provides the operations `EvaluatePolynomialFxP` and `Evaluate[Even/Odd]PolynomialFxP`.

The first, `EvaluatePolynomialFxP`, allows to evaluate a polynomial of the form  $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_dx^d$  where  $d$  denotes the *degree*. To do so, all that is needed are the polynomial coefficients `[a0, ..., ad]` (of type `Double[]`), the input `x : FixedPoint` and the output `y : FixedPoint` (initially zero):

```
EvaluatePolynomialFxP([1.0, 2.0], x, y);
```

The result,  $P(x)=1+2x$ , will be stored in `yFxP`.

The second, `EvaluateEvenPolynomialFxP`, and the third, `EvaluateOddPolynomialFxP`, are specializations for the cases of even and odd functions, respectively. That is, for an even/odd function  $f(x)$  and  $P_{\{even\}}(x)=a_0 + a_1 x^2 + a_2 x^4 + \dots + a_d x^{2d}$ ,  $f(x)$  is approximated well by  $P_{\{even\}}(x)$  or  $P_{\{odd\}}(x) := x \cdot P_{\{even\}}(x)$ , respectively. In Q#, these two cases can be handled as follows:

```
EvaluateEvenPolynomialFxP([1.0, 2.0], x, y);
```

which evaluates  $P_{\text{even}}(x) = 1 + 2x^2$ , and

```
EvaluateOddPolynomialFxP([1.0, 2.0], x, y);
```

which evaluates  $P_{\text{odd}}(x) = x + 2x^3$ .

## More samples

You can find more samples in the [main samples repository](#).

To get started, clone the repo and open the `Numerics` subfolder:

```
git clone https://github.com/Microsoft/Quantum.git
cd Quantum/samples/numerics
```

Then, `cd` into one of the sample folders and run the sample via

```
dotnet run
```

# Release Notes

6/30/2021 • 32 minutes to read • [Edit Online](#)

This article contains information on each Quantum Development Kit release.

For installation instructions, please refer to the [install guide](#).

For update instructions, please refer to the [update guide](#).

## QDK Version 0.18.2106.148911

*Release date: June 25th, 2021*

- You can now [configure](#) how many solutions you want returned from a solver run.
- A new NuGet package [Microsoft.Quantum.AutoSubstitution](#), which when added to a Q# project, allows you to annotate operations with the `SubstitutableOnTarget(AltOp, Sim)` attribute. It will then call `AltOp` instead of the annotated operation, whenever it is executed using `Sim`.
- Integration with Azure-Identity provides more mechanisms to [authenticate](#) with Azure.
- The .NET [Microsoft.Azure.Management.Quantum](#) now returns the Restricted Access URL so you can to know more/apply for a restricted access SKU.
- Preview support for noisy simulation in open systems and stabilizer representations [qsharp-runtime#714](#). See [here](#) for documentation on preview simulators.
- Using [quantum-viz.js](#) as the engine to render the output from the jupyter notebook %trace magic.

## QDK Version 0.17.2105.144881

*Release date: June 1st, 2021*

- Reverted a change in the `azure-quantum` Python client that could create authentication issues for some users (refer to issues [#66](#), [#67](#)).

## QDK Version 0.17.2105.143879

*Release date: May 26th, 2021*

- Added a new function to the `azure-quantum` Python client to support the translation of binary optimization terms from `npz` to Azure Quantum. See full details in [QDK Python](#).
- Published [QIR oracle generation sample](#). This program allows turns classical Q# functions on Boolean inputs into quantum implementations of that functions in terms of Q# operations at the level of QIR. This allows, for example, to implement quantum algorithms that are used by many quantum algorithms readily as classical functions.
- Fixed a bug that prevents QIR generation from being enabled in the iqsharp-base Docker image. See details [here](#).
- Implemented new special functions (e.g.: factorial and log-gamma) in Microsoft.Quantum.Math ([microsoft/QuantumLibraries#448](#)). Thanks to @TheMagicNacho for the contribution ([microsoft/QuantumLibraries#440](#))!
- C# Client: Changed input data format type to "v2" for Quantum Computing.
- Released Az CLI quantum extension version 0.5.0: Adapted to 'az' tool version 2.23.0, adding user agent information on calls to Azure Quantum Service.

## Azure Quantum service update

- Added PA (population annealing) and SSMC (sub-stochastic Monte Carlo) solvers along with preview access via a specialized SKU available to a subset of customers.
- Added support for new regions: Japan East, Japan West, UK South, UK West
- Set Provider in Failed state if provisioning fails. Previously it would be stuck in Launching/Updating state.
- Added help button in portal to direct user to support forum.
- Rendered provider cost in localized currency from Azure Marketplace.
- Added feedback button in portal to gather user feedback.
- Added quickstart guide in portal in overview blade.

## Version 0.16.2105.140472

*Release date: May 10th, 2021*

- Fixed dependency error in IQSharp on System.Text.Json when submitting jobs to Azure Quantum. See full details in issue [iqsharp#435](#).
- Resolved issue affecting joint measurements of multi-qubit states on some combinations of Pauli basis resulting in incorrect values. For details, please refer to issue [qsharp-runtime#680](#).

## Version 0.16.2104.138035

*Release date: April 27th, 2021*

- Improved Q# type inference based on the Hindley-Milner type inference algorithm.
- Added support for NumPy types in coefficient definitions for problems in QIO Python SDK.
- Updated control-plane swagger file to [support restricted access SKUs](#).
- Added new `StreamingProblem` class in QIO Python SDK. It supports the same interface for adding terms to a problem definition as the `Problem` class. However, once terms are added to the problem they are queued to be uploaded by a background thread and are not kept in memory for future reference.
- Restored the packages size of Microsoft.Quantum.Sdk and Microsoft.Quantum.Compiler back to normal. (See related note in 0.15.2103.133969)
- Improved compiler performance.
- Released Az CLI quantum extension version 0.4.0: Exposed URL for restricted access SKUs. Fixed regression on offerings commands dependent on Azure Marketplace APIs.

## Version 0.15.2103.133969

*Release date: March 30th, 2021*

- Released QIR emission as experimental feature (<https://github.com/microsoft/qsharp-compiler/tree/main/src/QsCompiler/QirGeneration#qir-emission---preview-feature>). The inclusion of the necessary LLVM packages, and in particular LlvmLibs, causes an increase in package size of the Microsoft.Quantum.Sdk and the Microsoft.Quantum.Compiler, and correspondingly to longer download times the first time the new versions are used. We are working on reducing that again in the future.
- Loosen restriction on AllowAtMostNCallsCA operation (<https://github.com/microsoft/QuantumLibraries/pull/431>).
- Added missing APIs for Math Library (<https://github.com/microsoft/QuantumLibraries/issues/413>).
- Removed `GetQubitsAvailableToBorrow` operation and `GetQubitsAvailableToUse` operation (<https://github.com/microsoft/QuantumLibraries/issues/418>).
- Fixed Q# Language Server fails during initialization in Visual Studio due to JsonReaderException (<https://github.com/microsoft/qsharp-compiler/issues/885>).

- Added support for multiple entry points.
- Released Az CLI quantum extension version 0.3.0: Updated command 'az quantum workspace create' to require an explicit list of Quantum providers and remove a default. Fixed issue with incorrect location parameter during job submission.

## Version 0.15.2102.129448

*Release date: February 25th, 2021*

- Improved IQ# debug user experience by adding a horizontal scrollbar to scroll both execution path and basis state visualizations.
- New functions to represent the group product and group inverse on the single-qubit Clifford group, to quickly define common single-qubit Clifford operators, and to apply single-qubit Clifford operators as operations. For more information, see issue [#409](#).
- Addressing security issue in the Microsoft Quantum Development Kit for Visual Studio Code extension. For details, refer to [CVE-2021-27082](#).
- Released Az CLI quantum extension version 0.2.0: Added parameter '--provider-sku-list' to 'az quantum workspace create' to allow specification of Quantum providers. Added command group 'az quantum offerings' with 'list', 'accept-terms' and 'show-terms'.

## Version 0.15.2102.128318

*Release date: February 12th, 2021*

- Fix "'npm' is not recognized as an internal or external command" error during creation of Q# projects with Visual Studio Code extension. See issue [#848](#).

## Version 0.15.2101.126940

*Release date: January 29th, 2021*

- Added project templates to Q# compiler for executables targeting IonQ and Honeywell providers
- Update IQ# kernel syntax highlighting to include changes to Q# syntax introduced in version [0.15.2101125897](#)
- Bugfix to support passing arrays as input arguments to Q# programs submitted to Azure Quantum via `%azure.execute`, see issue [#401](#)
- Fix "Permission denied" error encountered using `az` inside of `iqsharp-base` Docker images, see issue [#404](#)
- Released Az CLI quantum extension version 0.1.0: Provided command line tool for workspace management and quantum computing job submission.

## Version 0.15.2101125897

*Release date: January 26th, 2021*

- Simplified qubit allocation, providing more convenient syntax for allocating qubits, [see details in Q# language repository](#).
- Created QDK-Python repository that includes `azure-quantum`, the Python client for submitting quantum-inspired optimization jobs to the Azure Quantum service, as well as `qdk`, including `qdk.chemistry`, a Python-based convenience layer for the Q# chemistry library that includes molecular visualization and functionality to generate input files for several chemistry packages such as NWChem, Psi4 and OpenMolcas.
- Parentheses are now optional for operation and function types and `if`, `elif`, `while` and `until` statements. Parentheses for `for`, `use` and `borrow` statements have been deprecated.
- Improved width estimates for optimal depth, [see details](#).

- Apply unitary operation provided as explicit matrix using `ApplyUnitary` ([QuantumLibraries#391](#), external contribution by Dmytro Fedoriaka)
- Fixed <https://github.com/microsoft/iqsharp/issues/387> by mitigating performance impact on IQ# kernel startup.

## Version 0.14.2011120240

*Release date: November 25th, 2020*

- Improved compiler performance due to faster reference loading.
- Added an [ANTLR grammar for Q#](#) to the Q# language specification.
- Updated the `Microsoft.Quantum.Preparation` namespace to be more consistent with style guide and API design principles, and to support purified mixed states with additional data (see [proposal](#), [review notes](#) and PRs [#212](#), [#322](#), [#375](#), [#376](#)).
- Parentheses around repeated call expressions are now optional: `(Foo(x))(y)` may be written as `Foo(x)(y)`.
- Users of the Visual Studio or Visual Studio Code extensions who have installed .NET 5 or Visual Studio 16.8 may be prompted to install .NET Core 3.1 to continue to work with the extensions.

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#), [IQ#](#) and [Katas](#).

## Version 0.13.20111004

*Release date: November 10th, 2020*

This release disables IntelliSense features for Q# files in Visual Studio and Visual Studio Code when a project file is not present. This resolves an issue where IntelliSense features may stop working after adding a new Q# file to a project (see [qsharp-compiler#720](#)).

## Version 0.13.20102604

*Release date: October 27th, 2020*

This release contains the following:

- Resource estimation now emits simultaneously achievable depth and width estimates in addition to the qubit count. See [here](#) for details.

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#), [IQ#](#) and [Katas](#).

## Version 0.12.20100504

*Release date: October 5th, 2020*

This release fixes a bug affecting load of Q# notebooks (see [iqsharp#331](#)).

## Version 0.12.20092803

*Release date: September 29th, 2020*

This release contains the following:

- Announcement and draft specification of [Quantum Intermediate Representation \(QIR\)](#) intended as a common format across different front- and back-ends. See also our [blog post](#) on QIR.
- Launch of our new [Q# language repo](#) containing also the full [Q# documentation](#).
- Performance improvements for QuantumSimulator for programs involving a large number of qubits: better application of gate fusion decisions; improved parallelization on Linux system; added intelligent scheduling

of gate execution; bug fixes.

- IntelliSense features are now supported for Q# files in Visual Studio and Visual Studio Code even without a project file.
- Various Q#/Python interoperability improvements and bug fixes, including better support for NumPy data types.
- Improvements to the Microsoft.Quantum.Arrays namespace (see [microsoft/QuantumLibraries#313](#)).
- Added a new [Repeat-Until-Success sample](#) that uses only two qubits.

Since the last release, the default branch in each of our open source repositories has been renamed to `main`.

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#), [IQ#](#) and [Katas](#).

## Version 0.12.20082513

*Release date: August 25th, 2020*

This release contains the following:

- New [Microsoft.Quantum.Random namespace](#), providing a more convenient way to sample random values from within Q# programs. ([QuantumLibraries#311](#), [qsharp-runtime#328](#))
- Improved [Microsoft.Quantum.Diagnostics namespace](#) with new `DumpOperation` operation, and new operations for restricting qubit allocation and oracle calls. ([QuantumLibraries#302](#))
- New `%project` magic command in IQ# and `qsharp.projects` API in Python to support references to Q# projects outside the current workspace folder. See [iqsharp#277](#) for the current limitations of this feature.
- Support for automatically loading `.csproj` files for IQ#/Python hosts, which allows external project or package references to be loaded at initialization time. See the guide for using [Q# with Python and Jupyter Notebooks](#) for more details.
- Added ErrorCorrection.Syndrome sample.
- Added tunable coupling to SimpleIsing.
- Updated HiddenShift sample.
- Added sample for solving Sudoku with Grover's algorithm (external contribution)
- General bug fixes.

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#), [IQ#](#) and [Katas](#).

## Version 0.12.20072031

*Release date: July 21st, 2020*

This release contains the following:

- Opened namespaces in Q# notebooks are now available when running all future cells. This allows, for example, namespaces to be opened once in a cell at the top of the notebook, rather than needing to open relevant namespaces in each code cell. A new `%1sopen` magic command displays the list of currently-opened namespaces.

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#), [IQ#](#) and [Katas](#).

## Version 0.12.20070124

*Release date: July 2nd, 2020*

This release contains the following:

- New `qdk-chem` tool for converting legacy electronic structure problem serialization formats (for example,

FCIDUMP) to Broombridge

- New functions and operations in the `Microsoft.Quantum.Synthesis` namespace for coherently applying classical oracles using transformation- and decomposition-based synthesis algorithms.
- IQ# now allows arguments to the `%simulate`, `%estimate`, and other magic commands. See the [%simulate magic command reference](#) for more details.
- New phase display options in IQ#. See the [%config magic command reference](#) for more details.
- IQ# and the `qsharp` Python package are now provided via conda packages (`qsharp` and `iqsharp`) to simplify local installation of Q# Jupyter and Python functionality to a conda environment. See the [Q# Jupyter Notebooks](#) and [Q# with Python](#) installation guides for more details.
- When using the simulator, qubits no longer need to be in the  $|0\rangle$  state upon release, but can be automatically reset if they were measured immediately before releasing.
- Updates to make it easier for IQ# users to consume library packages with different QDK versions, requiring only major & minor version numbers match rather than the exact same version
- Removed deprecated `Microsoft.Quantum.Primitive.*` namespace
- Moved operations:
  - `Microsoft.Quantum.Intrinsic.Assert` is now `Microsoft.Quantum.Diagnostics.AssertMeasurement`
  - `Microsoft.Quantum.Intrinsic.AssertProb` is now `Microsoft.Quantum.Diagnostics.AssertMeasurementProbability`
- Bug fixes

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#), [IQ#](#) and [Katas](#).

## Version 0.11.2006.403

*Release date: June 4th, 2020*

This release fixes a bug affecting compilation of Q# projects.

## Version 0.11.2006.207

*Release date: June 3rd, 2020*

This release contains the following:

- Q# notebooks and Python host programs will no longer fail when a Q# entry point is present
- Updates to [Standard library](#) to use access modifiers
- Compiler now allows plug-in of rewrite steps between built-in rewrite steps
- Several deprecated functions and operations have been removed following the schedule described in our [API principles](#). Q# programs and libraries that build without warnings in version 0.11.2004.2825 will continue to work unmodified.

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#), [IQ#](#) and [Katas](#).

### NOTE

This version contains a bug affecting compilation of Q# projects. We recommend upgrading to a newer release.

## Version 0.11.2004.2825

*Release date: April 30th, 2020*

This release contains the following:

- New support for Q# applications, which no longer require a C# or Python host file. For more information on getting started with Q# applications, see [here](#).
- Updated quantum random number generator quickstart to no longer require a C# or Python host file. See the updated [Quickstart](#)
- Performance improvements to IQ# Docker images

**NOTE**

Q# applications using the new `@EntryPoint()` attribute currently cannot be called from Python or .NET host programs. See the [Python](#) and [.NET interoperability](#) guides for more information.

## Version 0.11.2003.3107

*Release date: March 31, 2020*

This release contains minor bugfixes for version 0.11.2003.2506.

## Version 0.11.2003.2506

*Release date: March 26th, 2020*

This release contains the following:

- New support for access modifiers in Q#
- Updated to .NET Core SDK 3.1

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#) and [Katas](#).

## Version 0.10.2002.2610

*Release date: February 27th, 2020*

This release contains the following:

- New Quantum Machine Learning library, for more information go to our [QML docs page](#)
- IQ# bug fixes, resulting in up to a 10-20x performance increase when loading NuGet packages

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#) and [Katas](#).

## Version 0.10.2001.2831

*Release date: January 29th, 2020*

This release contains the following:

- New Microsoft.Quantum.SDK NuGet package which will replace Microsoft.Quantum.Development.Kit NuGet package when creating new projects. Microsoft.Quantum.Development.Kit NuGet package will continue to be supported for existing projects.
- Support for Q# compiler extensions, enabled by the new Microsoft.Quantum.SDK NuGet packge, for more information see the [documentation on Github](#), the [compiler extensions sample](#) and the [Q# Dev Blog](#)
- Added support for .NET Core 3.1, it is highly recommended to have version 3.1.100 installed since building with older .NET Core SDK versions may cause issues
- New compiler transformations available under Microsoft.Quantum.QsCompiler.Experimental
- New functionality to expose output state vectors as HTML in IQ#
- Added support for EstimateFrequencyA to Microsoft.Quantum.Characterization for Hadamard and SWAP

tests

- AmplitudeAmplification namespace now uses Q# style guide

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#) and [Katas](#).

## Version 0.10.1912.0501

*Release date: December 5th, 2019*

This release contains the following:

- New Test attribute for Q# unit testing, see updated API documentation [here](#) and updated testing & debugging guide [here](#)
- Added stack trace in the case of a Q# program run error
- Support for breakpoints in Visual Studio Code due to an update in the [OmniSharp C# Visual Studio Code extension](#)

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#) and [Katas](#).

## Version 0.10.1911.1607

*Release date: November 17th, 2019*

This release contains the following:

- Performance fix for [Quantum Katas](#) and Jupyter notebooks

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#) and [Katas](#).

## Version 0.10.1911.307

*Release date: November 1st, 2019*

This release contains the following:

- Updates to Visual Studio Code & Visual Studio extensions to deploy language server as a self-contained executable file, eliminating the .NET Core SDK version dependency
- Migration to .NET Core 3.0
- Breaking change to Microsoft.Quantum.Simulation.Core.IOperationFactory with introduction of new `Fail` method. It affects only custom simulators that do not extend SimulatorBase. For more details, [view the pull request on GitHub](#).
- New support for Deprecated attributes

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#) and [Katas](#).

## Version 0.9.1909.3002

*Release date: September 30th, 2019*

This release contains the following:

- New support for Q# code completion in Visual Studio 2019 (versions 16.3 & later) & Visual Studio Code
- New [Quantum Kata](#) for quantum adders

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#) and [Katas](#).

## Version 0.9 (*PackageReference 0.9.1908.2902*)

*Release date: August 29th, 2019*

This release contains the following:

- New support for [conjugation statements](#) in Q#
- New code actions in the compiler, such as: "replace with", "add documentation", and simple array item update
- Added install template and new project commands to Visual Studio Code extension
- Added new variants of ApplyIf combinator such as [Microsoft.Quantum.Canon.ApplyIfOne](#)
- Additional [Quantum Katas](#) converted to Jupyter Notebooks
- Visual Studio Extension now requires Visual Studio 2019

See the full list of closed PRs for [libraries](#), [compiler](#), [runtime](#), [samples](#) and [Katas](#).

The changes are summarized here as well as instructions for upgrading your existing programs. Read more about these changes on the [Q# dev blog](#).

## Version 0.8 (*PackageReference 0.8.1907.1701*)

*Release date: July 12, 2019*

This release contains the following:

- New indexing for slicing arrays, [see the language reference](#) for more information.
- Added Dockerfile hosted on the [Microsoft Container Registry](#), see the [IQ# repository](#) for more information
- Breaking change for [the trace simulator](#), update to configuration settings, name changes; see the [.NET API Browser](#) for the updated names.

See the full list of closed PRs for [libraries](#) and [samples](#).

## Version 0.7 (*PackageReference 0.7.1905.3109*)

*Release date: May 31, 2019*

This release contains the following:

- additions to the Q# language,
- updates to the chemistry library,
- a new numerics library.

See the full list of closed PRs for [libraries](#) and [samples](#).

The changes are summarized here as well as instructions for upgrading your existing programs. Read more about these changes on the [Q# dev blog](#).

### **Q# language syntax**

This release adds new Q# language syntax:

- Add named items for [user-defined types](#).
- User-defined type constructors can now be used as functions.
- Add support for [copy-and-update](#) and [apply-and-reassign](#) in user-defined types.
- Fixup-block for [repeat-until-success](#) loops is now optional.
- We now support while loops in functions (not in operations).

### **Library**

This release adds a numerics library: Learn more about how to [use the new numerics library](#) and try out the [new samples](#). [PR #102](#).

This release reorganizes extends and updates the chemistry library:

- Improves modularity of components, extensibility, general code cleanup. [PR #58](#).
- Add support for [multi-reference wavefunctions](#), both sparse multi-reference wavefunctions and unitary coupled cluster. [PR #110](#).
- (Thank you!) [1QBit](#) contributor (@valentinS4t1qbit): Energy evaluation using variational ansatz. [PR #120](#).
- Updating [Broombridge](#) schema to new [version 0.2](#), adding unitary coupled cluster specification. [Issue #65](#).
- Adding Python interoperability to chemistry library functions. Try out this [sample](#). [Issue #53](#) [PR #110](#).

## Version 0.6.1905

*Release date: May 3, 2019*

This release contains the following:

- makes changes to the Q# language,
- restructures the Quantum Development Kit libraries,
- adds new samples, and
- fixes bugs. Several closed PRs for [libraries](#) and [samples](#).

The changes are summarized here as well as instructions for upgrading your existing programs. You can read more about these changes on [devblogs.microsoft.com/qsharp](https://devblogs.microsoft.com/qsharp).

### **Q# language syntax**

This release adds new Q# language syntax:

- Add a [shorthand way to express specializations of quantum operations](#) (control and adjoints) with `+ [ ]` operators. The old syntax is deprecated. Programs that use the old syntax (for example, `[ ] : adjoint`) will continue to work, but a compile time warning will be generated.
- Add a new ternary operator for [copy-and-update](#), `w/ [ ] <- [ ]`, can be used to express array creation as a modification of an existing array.
- Add the common [apply-and-reassign statement](#), for example, `[ ] += [ ]`, `[ ] w/= [ ]`.
- Add a way to specify a short name for namespaces in [open directives](#).

With this release, we no longer allow an array element to be specified on the left side of a set statement. This is because that syntax implies that arrays are mutable when in fact, the result of the operation has always been the creation of a new array with the modification. Instead, a compiler error will be generated with a suggestion to use the new copy-and-update operator, `w/ [ ]`, to accomplish the same result.

### **Library restructuring**

This release reorganizes the libraries to enable their growth in a consistent way:

- Renames the Microsoft.Quantum.Primitive namespace to Microsoft.Quantum.Intrinsic. These operations are implemented by the target machine. The Microsoft.Quantum.Primitive namespace is deprecated. A runtime warning will advise when programs call operations and functions using deprecated names.
- Renames the Microsoft.Quantum.Canon package to Microsoft.Quantum.Standard. This package contains namespaces that are common to most Q# programs. This includes:
  - Microsoft.Quantum.Canon for common operations
  - Microsoft.Quantum.Arithmetic for general purpose arithmetic operations
  - Microsoft.Quantum.Preparation for operations used to prepare qubit state
  - Microsoft.Quantum.Simulation for simulation functionality

With this change, programs that include a single "open" statement for the namespace Microsoft.Quatum.Canon

may encounter build errors if the program references operations that were moved to the other three new namespaces. Adding the additional open statements for the three new namespaces is a straightforward way to resolve this issue.

- Several namespaces have been deprecated as the operations within have been reorganized to other namespaces. Programs that use these namespaces will continue to work, and a compile time warning will denote the namespace where the operation is defined.
- The Microsoft.Quantum.Arithmetic namespace has been normalized to use the [LittleEndian user defined type](#) user-defined type. Use the function [BigEndianAsLittleEndian](#) when needed to convert to little endian.
- The names of several callables (functions and operations) have been changed to conform to the [Q# Style Guide](#). The old callable names are deprecated. Programs that use the old callables will continue to work with a compile time warning.

## New Samples

We added a [sample of using Q# with F# driver](#).

**Thank you!** to the following contributor to our open code base at <http://github.com/Microsoft/Quantum>. These contributions add significantly to the rich samples of Q# code:

- Mathias Soeken (@msoeken): Oracle function synthesis. [PR #135](#).

## Migrating existing projects to 0.6.1905.

See the [install guide](#) to update the QDK.

If you have existing Q# projects from version 0.5 of the Quantum Development Kit, the following are the steps to migrate those projects to the newest version.

1. Projects need to be upgraded in order. If you have a solution with multiple projects, update each project in the order they are referenced.
2. From a command prompt, Run `dotnet clean` to remove all existing binaries and intermediate files.
3. In a text editor, edit the .csproj file to change the version of all the "Microsoft.Quantum" `PackageReference` to version 0.6.1904, and change the "Microsoft.Quantum.Canon" package name to "Microsoft.Quantum.Standard", for example:

```
<PackageReference Include="Microsoft.Quantum.Standard" Version="0.6.1905.301" />
<PackageReference Include="Microsoft.Quantum.Development.Kit" Version="0.6.1905.301" />
```

4. From the command prompt, run this command: `dotnet msbuild`
5. After running this, you might still need to manually address errors due to changes listed above. In many cases, these errors will also be reported by IntelliSense in Visual Studio or Visual Studio Code.
  - Open the root folder of the project or the containing solution in Visual Studio 2019 or Visual Studio Code.
  - After opening a .qs file in the editor, you should see the output of the Q# language extension in the output window.
  - After the project has loaded successfully (indicated in the output window) open each file and manually to address all remaining issues.

#### **NOTE**

- For the 0.6 release, the language server included with the Quantum Development Kit does not support multiple workspaces.
- In order to work with a project in Visual Studio Code, open the root folder containing the project itself and all referenced projects.
- In order to work with a solution in Visual Studio, all projects contained in the solution need to be in the same folder as the solution or in one of its subfolders.
- References between projects migrated to 0.6 and higher and projects using older package versions are **not** supported.

## Version 0.5.1904

*Release date: April 15, 2019*

This release contains bug fixes.

## Version 0.5.1903

*Release date: March 27, 2019*

This release contains the following:

- Adds support for Jupyter Notebook, which offers a great way to learn about Q#. [Check out new Jupyter Notebook samples and learn how to write your own Notebooks.](#)
- Adds integer adder arithmetic to the Quantum Canon library. See also a Jupyter Notebook that [describes how to use the new integer adders](#).
- Bug fix for DumpRegister issue reported by the community ([#148](#)).
- Added ability to return from within a [using- and borrowing-statement](#).
- Revamped [getting started guide](#).

## Version 0.5.1902

*Release date: February 27, 2019*

This release contains the following:

- Adds support for a cross-platform Python host. The `qsharp` package for Python makes it easy to simulate Q# operations and functions from within Python. Learn more about [Python interoperability](#).
- The Visual Studio and Visual Studio Code extensions now support renaming of symbols (for example, functions and operations).
- The Visual Studio extension can now be installed on Visual Studio 2019.

## Version 0.4.1901

*Release date: January 30, 2019*

This release contains the following:

- adds support for a new primitive type, `BigInt`, which represents a signed integer of arbitrary size. Learn more about [BigInt](#).
- adds new Toffoli simulator, a special purpose fast simulator that can simulate X, CNOT and multi-controlled X

quantum operations with very large numbers of qubits. Learn more about [Toffoli simulator](#).

- adds a simple resource estimator that estimates the resources required to run a given instance of a Q# operation on a quantum computer. Learn more about the [Resource Estimator](#).

## Version 0.3.1811.2802

*Release date: November 28, 2018*

Even though our VS Code extension was not using it, it was flagged and removed from the marketplace during the [extensions purge](#) related to the `event-stream` NPM package. This version removes all runtime dependencies that could make the extension trigger any red flags.

If you had previously installed the extension you will need to install it again by visiting the [Microsoft Quantum Development Kit for Visual Studio Code](#) extension on the Visual Studio Marketplace and press Install. We are sorry about the inconvenience.

## Version 0.3.1811.1511

*Release date: November 20, 2018*

This release fixes a bug that prevented some users to successfully load the Visual Studio extension.

## Version 0.3.1811.203

*Release date: November 2, 2018*

This release includes a few bug fixes, including:

- Invoking `DumpMachine` could change the state of the simulator under certain situations.
- Removed compilation warnings when building projects using a version of .NET Core previous to 2.1.403.
- Clean up of documentation, specially the tooltips shown during mouse hover in VS Code or Visual Studio.

## Version 0.3.1810.2508

*Release date: October 29, 2018*

This release includes new language features and an improved developer experience:

- This release includes a language server for Q#, as well as the client integrations for Visual Studio and Visual Studio Code. This enables a new set of IntelliSense features along with live feedback on typing in form of squiggly underlinings of errors and warnings.
- This update greatly improves diagnostic messages in general, with easy navigation to and precise ranges for diagnostics and additional details in the displayed hover information.
- The Q# language has been extended in ways that unifies the ways developers can do common operations and new enhancements to the language features to powerfully express quantum computation. There are a handful of breaking changes to the Q# language with this release.

This release also includes a new quantum chemistry library:

- The chemistry library contains new Hamiltonian simulation features, including:
  - Trotter–Suzuki integrators of arbitrary even order for improved simulation accuracy.
  - Qubitization simulation technique with chemistry-specific optimizations for reducing \$T\$-gate complexity.
- A new open source schema, called Broombridge Schema (in reference to a [landmark](#) celebrated as a birthplace of Hamiltonians), is introduced for importing representations of molecules and simulating them.

- Multiple chemical representations defined using the Broombridge Schema are provided. These models were generated by [NWChem](#), an open source high-performance computational chemistry tool.
- Tutorials and Samples describe how to use the chemistry library and the Broombridge data models to:
  - Construct simple Hamiltonians using the chemistry library
  - Visualize ground and excited energies of Lithium Hydride using phase estimation.
  - Perform resource estimates of quantum chemistry simulation.
  - Estimate energy levels of molecules represented by the Broombridge schema.
- Documentation describes how to use NWChem to generate additional chemical models for quantum simulation with Q#.

Learn more about the [Quantum Development Kit chemistry library](#).

With the new chemistry library, we are separating out the libraries into a new GitHub repo, [Microsoft/QuantumLibraries](#). The samples remain in the repo [Microsoft/Quantum](#). We welcome contributions to both!

This release includes bug fixes and features for issues reported by the community.

### Community Contributions

Thank you! to the following contributors to our open code base at <http://github.com/Microsoft/Quantum>.

These contributions add significantly to the rich samples of Q# code:

- Rolf Huisman ([@RolfHuisman](#)): Improved the experience for QASM/Q# developers by creating a QASM to Q# translator. [PR #58](#).
- Andrew Helwer ([@ahelwer](#)): Contributed a sample implementing the CHSH Game, a quantum game related to non-locality. [PR #84](#).

Thank you also to Rohit Gupta ([@guptarohit](#),[PR #90](#)), Tanaka Takayoshi ([@tanaka-takayoshi](#),[PR #289](#)), and Lee James O'Riordan ([@mlxd](#),[PR #96](#)) for their work improving the content for all of us through documentation, spelling and typo corrections!

## Version 0.2.1809.701

*Release date: September 10, 2018*

This release includes bug fixes for issues reported by the community.

## Version 0.2.1806.3001

*Release date: June 30, 2018*

This releases is just a quick fix for [issue #48 reported on GitHub](#) (Q# compilation fails if user name contains a blank space). Follow same update instructions as [0.2.1806.1503](#) with the corresponding new version ([0.2.1806.3001-preview](#)).

## Version 0.2.1806.1503

*Release date: June 22, 2018*

This release includes several community contributions as well as an improved debugging experience and improved performance. Specifically:

- Performance improvements on both small and large simulations for the QuantumSimulator target machine.
- Improved debugging functionality.
- Community contributions in bug fixes, new helper functions, operations and new samples.

## Performance improvements

This update includes significant performance improvements for simulation of large and small numbers of qubits for all the target machines. This improvement is easily visible with the H<sub>2</sub> simulation that is a standard sample in the Quantum Development Kit.

## Improved debugging functionality

This update adds new debugging functionality:

- Added two new operations, @"microsoft.quantum.extensions.diagnostics.dumpmachine" and @"microsoft.quantum.extensions.diagnostics.dumpregister" that output wave function information about the target quantum machine at a point in time.
- In Visual Studio, the probability of measuring a \$|1\rangle\$ on a single qubit is now automatically shown in the debugging window for the QuantumSimulator target machine.
- In Visual Studio, improved the display of variable properties in the **Autos** and **Locals** debug windows.

Learn more about [Testing and Debugging](#).

## Community Contributions

The Q# coder community is growing and we are thrilled to see the first user contributed libraries and samples that were submitted to our open code base at <http://github.com/Microsoft/quantum>. A big **Thank you!** to the following contributors:

- Mathias Soeken ([@msoeken](#)): contributed a sample defining a transformation based logic synthesis method that constructs Toffoli networks to implement a given permutation. The code is written entirely in Q# functions and operations. [PR #41](#)
- RolfHuisman ([@RolfHuisman](#)): Microsoft MVP Rolf Huisman contributed a sample that generates flat QASM code from Q# code for a restricted class of programs that do not have classical control flow and restricted quantum operations. [PR #59](#)
- Sarah Kasier ([@crazy4pi314](#)): helped to improve our code base by submitting a library function for controlled operations. [PR #53](#)
- Jessica Lemieux ([@Lemj3111](#)): fixed @"microsoft.quantum.canon.quantumphaseestimation" and created new unit tests. [PR #54](#)
- Tama McGlinn ([@TamaMcGlinn](#)): cleaned the Teleportation sample by making sure the QuantumSimulator instance is disposed. [PR #20](#)

Additionally, a big **Thank You!** to these Microsoft Software Engineers from the Commercial Engineering Services team contributors who made valuable changes to our documentation during their Hackathon. Their changes vastly improved the clarity and onboarding experience for all of us:

- Sascha Corti
- Mihaela Curmei
- John Donnelly
- Kirill Logachev
- Jan Pospisil
- Anita Ramanan
- Frances Tibble
- Alessandro Vozza

## Update existing projects

This release is fully backwards compatible. Just update the nuget pakages in your projects to version [0.2.1806.1503-preview](#) and do a **full rebuild** to make sure all intermediate files are regenerated.

From Visual Studio, follow the normal instructions on how to [update a package](#).

To update project templates for the command line, run the following command:

```
dotnet new -i "Microsoft.Quantum.ProjectTemplates::0.2.1806.1503-preview"
```

After running this command, any new projects created using `dotnet new <project-type> -lang Q#` will automatically use this version of the Quantum Development Kit.

To update an existing project to use the newest version, run the following command from within the directory for each project:

```
dotnet add package Microsoft.Quantum.Development.Kit -v "0.2.1806.1503-preview"  
dotnet add package Microsoft.Quantum.Canon -v "0.2.1806.1503-preview"
```

If an existing project also uses XUnit integration for unit testing, then a similar command can be used to update that package as well:

```
dotnet add package Microsoft.Quantum.Xunit -v "0.2.1806.1503-preview"
```

Depending on the version of XUnit that your test project uses, you may also need to update XUnit to 2.3.1:

```
dotnet add package xunit -v "2.3.1"
```

After the update, make sure you remove all temporary files generated by the previous version by doing:

```
dotnet clean
```

## Known Issues

No additional known issues to report.

## Version 0.2.1802.2202

*Release date: February 26, 2018*

This release brings support for development on more platforms, language interoperability, and performance enhancements. Specifically:

- Support for macOS- and Linux-based development.
- .NET Core compatibility, including support for Visual Studio Code across platforms.
- A full Open Source license for the Quantum Development Kit Libraries.
- Improved simulator performance on projects requiring 20 or more qubits.
- Interoperability with the Python language (preview release available on Windows).

## .NET Editions

The .NET platform is available through two different editions, the .NET Framework that is provided with Windows, and the open-source .NET Core that is available on Windows, macOS and Linux. With this release, most parts of the Quantum Development Kit are provided as libraries for .NET Standard, the set of classes common to both Framework and Core. These libraries are therefore compatible with recent versions of either .NET Framework or .NET Core.

Thus, to help ensure that projects written using the Quantum Development Kit are as portable as possible, we recommend that library projects written using the Quantum Development Kit target .NET Standard, while console applications target .NET Core. Since previous releases of the Quantum Development Kit only supported

.NET Framework, you may need to migrate your existing projects; see below for details on how to do this.

## Project Migration

Projects created using previous versions of Quantum Development Kit will still work, as long as you don't update the NuGet packages used in them. To migrate existing code to the new version, perform the following steps:

1. Create a new .NET Core project using the right type of Q# project template (Application, Library or Test Project).
2. Copy existing .qs and .cs/.fs files from the old project to the new project (using Add > Existing Item). Do not copy the AssemblyInfo.cs file.
3. Build and run the new project.

Please note that the operation RandomWalkPhaseEstimation from the namespace Microsoft.Quantum.Canon was moved into the namespace Microsoft.Research.Quantum.RandomWalkPhaseEstimation in the [Microsoft/Quantum-NC](#) repository.

## Known Issues

- The `--filter` option to `dotnet test` does not work correctly for tests written in Q#. As a result, individual unit tests cannot be run in Visual Studio Code; we recommend using `dotnet test` at the command prompt to re-run all tests.

## Version 0.1.1801.1707

*Release date: January 18, 2018*

This release fixes some issues reported by the community. Namely:

- The simulator now works with early non-AVX-enabled CPUs.
- Regional decimal settings will not cause the Q# parser to fail.
- `SignD` primitive operation now returns `Int` rather than `Double`.

## Version 0.1.1712.901

*Release date: December 11, 2017*

## Known Issues

### Hardware and Software Requirements

- The simulator included with the Quantum Development Kit requires a 64-bit installation of Microsoft Windows to run.
- Microsoft's quantum simulator, installed with the Quantum Development Kit, utilizes Advanced Vector Extensions (AVX), and requires an AVX-enabled CPU. Intel processors shipped in Q1 2011 (Sandy Bridge) or later support AVX. We are evaluating support for earlier CPUs and may announce details at a later time.

### Project Creation

- When creating a solution (.sln) that will use Q#, the solution must be one directory higher than each project (.csproj) in the solution. When creating a new solution, this can be accomplished by making sure that the "Create directory for solution" checkbox on the "New Project" dialog box is checked. If this is not done, the Quantum Development Kit NuGet packages will need to be installed manually.

### Q#

- Intellisense does not display proper errors for Q# code. Make sure that you are displaying Build errors in the Visual Studio Error List to see correct Q# errors. Also note that Q# errors will not show up until after you've done a build.
- Using a mutable array in a partial application may lead to unexpected behavior.

- Binding an immutable array to a mutable array (let `a = b`, where `b` is a mutable array) may lead to unexpected behavior.
- Profiling, code coverage and other VS plugins may not always count Q# lines and blocks accurately.
- The Q# compiler does not validate interpolated strings. It is possible to create C# compilation errors by misspelling variable names or using expressions in Q# interpolated strings.

#### **Simulation**

- The Quantum Simulator uses OpenMP to parallelize the linear algebra required. By default OpenMP uses all available hardware threads, which means that programs with small numbers of qubits will often run slowly because the coordination required will dwarf the actual work. This can be fixed by setting the environment variable `OMP_NUM_THREADS` to a small number. As a very rough rule of thumb, 1 thread is good for up to about 4 qubits, and then an additional thread per qubit is good, although this is highly dependent on your algorithm.

#### **Debugging**

- F11 (step in) doesn't work in Q# code.
- Code highlighting in Q# code at a breakpoint or single-step pause is sometimes inaccurate. The correct line will be highlighted, but sometimes the highlight will start and end at incorrect columns on the line.

#### **Testing**

- Tests must be run in 64-bit mode. If your tests are failing with a `BadImageFormatException`, go to the Test menu and select Test Settings > Default Processor Architecture > X64.
- Some tests take a long time (possibly as much as 5 minutes depending on your computer) to run. This is normal, as some use over twenty qubits; our largest test currently runs on 23 qubits.

#### **Samples**

- On some machines, some small samples may run slowly unless the environment variable `OMP_NUM_THREADS` is set to "1". See also the release note under "Simulation".

#### **Libraries**

- There is an implicit assumption that the qubits passed to an operation in different arguments are all distinct. For instance, all of the library operations (and all of the simulators) assume that the two qubits passed to a controlled NOT are different qubits. Violating this assumption may lead to unpredictable unexpected. It is possible to test for this using the quantum computer tracer simulator.
- The `Microsoft.Quantum.Bind` function may not act as expected in all cases.
- In the `Microsoft.Quantum.Extensions.Math` namespace, the `SignD` function returns a `Double` rather than an `Int`, although the underlying `System.Math.Sign` function always returns an integer. It is safe to compare the result against 1.0, -1.0, and 0.0, since these doubles all have exact binary representations.

# Quantum computing glossary

5/27/2021 • 6 minutes to read • [Edit Online](#)

## Adjoint

The complex conjugate transpose of an [operation](#). For operations that implement a [unitary](#) operator, the adjoint is the inverse of the operation and is indicated by a dagger symbol. For example, if the operation `U` represents the unitary operator `U`, then `Adjoint U` represents `U^\dagger`. For more information, see [Functor application](#).

## Auxiliary qubit

A [qubit](#) that serves as temporary memory for a quantum computer and is allocated and de-allocated as needed. Sometimes referred to as an [ancilla](#). For more information, see [Multiple qubits](#).

## Bell state

One of four specific maximally [entangled quantum states](#) of two qubits. The four states are defined  $\lvert \beta_{ij} \rangle = (\mathbb{I} \otimes X^i Z^j) (\lvert 00 \rangle + \lvert 11 \rangle) / \sqrt{2}$ . A Bell state is also known as an [EPR pair](#).

## Bloch sphere

A graphical representation of a single-[qubit quantum state](#) as a point in a three-dimensional unit sphere. For more information, see [Visualizing Qubits and Transformations using the Bloch Sphere](#).

## Callable

An [operation](#) or [function](#) in the [Q# language](#). For more information, see [Q# programs](#)

## Clifford group

The set of operations that occupy the octants of the [Bloch sphere](#) and effect permutations of the [Pauli operators](#). These include the operations `X$, Y$, Z$, H$` and `S$`.

## Controlled

A quantum [operation](#) that takes one or more [qubits](#) as enablers for the target operation. For more information, see [Functor application](#).

## Dirac Notation

A symbolic shorthand that simplifies the representation of [quantum states](#), also called *bra-ket* notation. The *bra* portion represents a row vector, for example  $\langle A | = \begin{bmatrix} A_{11} & A_{12} \end{bmatrix}$  and the *ket* portion represents a column vector,  $\langle B | = \begin{bmatrix} B_{11} \\ B_{21} \end{bmatrix}$ . For more information, see [Dirac Notation](#).

## Eigenvalue

The factor by which the magnitude of an [eigenvector](#) of a given transformation is changed by the application of

the transformation. Given a square matrix  $M$  and an eigenvector  $v$ , then  $Mv = cv$ , where  $c$  is the eigenvalue and can be a complex number of any argument. For more information, see [Advanced matrix concepts](#).

## Eigenvector

A vector whose direction is unchanged by a given transformation and whose magnitude is changed by a factor corresponding to that vector's [eigenvalue](#). Given a square matrix  $M$  and an eigenvalue  $c$ , then  $Mv = cv$ , where  $v$  is an eigenvector of the matrix and can be a complex number of any argument. For more information, see [Advanced matrix concepts](#).

## Entanglement

Quantum particles, such as [qubits](#), can be connected or *entangled* such that they cannot be described independently from each other. Their measurement results are correlated even when they are separated infinitely far away. Entanglement is essential to [measuring](#) the [state](#) of a qubit. For more information, see [Advanced matrix concepts](#).

## EPR pair

One of four specific maximally entangled [quantum states](#) of two [qubits](#). The four states are defined  $|\beta_{ij}\rangle = (\mathbb{1} \otimes X^i Z^j) (|00\rangle + |11\rangle) / \sqrt{2}$ . An EPR pair is also known as a [Bell state](#)

## Evolution

How a [quantum state](#) changes over time. For more information, see [Matrix exponentials](#).

## Function

A type of subroutine in the Q# language that is purely deterministic. While functions are used within quantum algorithms, they cannot act on [qubits](#) or call [operations](#). For more information, see [Q# programs](#)

## Gate

A legacy term for certain intrinsic quantum [operations](#), based on the concept of classical logic gates. A [quantum circuit](#) is a network of gates, based on the similar concept of classical logic circuits.

## Global phase

When two [states](#) are identical up to a multiple of a complex number  $e^{i\phi}$ , they are said to differ up to a global phase. Unlike local phases, global phases cannot be observed through any [measurement](#). For more information, see [The Qubit](#).

## Hadamard

The Hadamard operation (also referred to as the Hadamard gate or transform) acts on a single [qubit](#) and puts it in an even [superposition](#) of  $|\ket{0}$  or  $|\ket{1}$  if the qubit is initially in the  $|\ket{0}$  state. In Q#, this operation is applied by the pre-defined [`H`](#) operation.

## Immutable

A variable whose value cannot be changed. An immutable variable in Q# is created using the [`let`](#) keyword. To declare variables that *can* be changed, use the [`mutable`](#) keyword to declare and the [`set`](#) keyword to modify the

value.

## Measurement

A manipulation of a [qubit](#) (or set of qubits) that yields the result of an observation, in effect obtaining a classical bit. For more information, see [The Qubit](#).

## Mutable

A variable whose value may be changed after it is created. A mutable variable in Q# is declared using the `mutable` keyword and modified using the `set` keyword. Variables created with the `let` keyword are [immutable](#) and their value cannot be changed.

## Namespace

A label for a collection of related names (for example, [operations](#), [functions](#), and [user-defined types](#)). For instance the namespace [Microsoft.Quantum.Preparation](#) labels all of the symbols defined in the standard library that help with preparing initial states.

## Operation

The basic unit of quantum computation in Q#. It is roughly equivalent to a function in C, C++ or Python, or a static method in C# or Java. For more information, see [Q# programs](#).

## Oracle

A subroutine that provides data-dependent information to a quantum algorithm at runtime. Typically, the goal is to provide a [superposition](#) of outputs corresponding to inputs that are in superposition. For more information, see [Oracles](#).

## Partial application

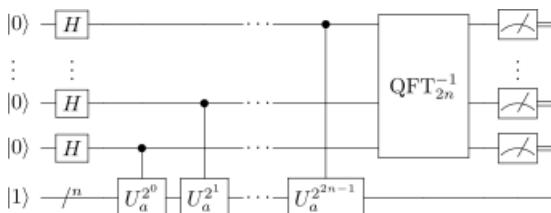
Calling a [function](#) or [operation](#) without all the required inputs. This returns a new [callable](#) that only needs the missing parameters (indicated by an underscore) to be supplied during a future application. For more information, see [Partial application](#).

## Pauli operators

A set of three  $2 \times 2$  unitary matrices known as the `x`, `y` and `z` quantum operations. The identity matrix, `$I$`, is often included in the set as well.  $\$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ ,  $\$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ ,  $\$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$ ,  $\$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ . For more information, see [Single-qubit operations](#).

## Quantum circuit diagram

A method to graphically represent the sequence of [gates](#) for simple quantum programs, for example



For more information, see [Quantum circuits](#).

## Quantum libraries

Collections of [operations](#), [functions](#) and [user-defined types](#) for creating Q# programs. The [Standard library](#) is installed by default. Other libraries available are the [Chemistry library](#), the [Numerics library](#) and the [Machine learning library](#).

## Quantum state

The precise state of an isolated quantum system, from which [measurement](#) probabilities can be extracted. In quantum computing, the quantum simulator uses this information to simulate how qubits respond to operations. For more information, see [The Qubit](#).

## Qubit

A basic unit of quantum information, analogous to a *bit* in classical computing. For more information, see [The Qubit](#).

## Repeat-until-success

A concept often used in quantum algorithms that consists of repeatedly applying a computation until a certain condition is satisfied. When the condition is not satisfied, often a fixup is required before retrying by entering the next iteration. For more information, see the [Q# user guide](#)

## Standard libraries

[Operations](#), [functions](#) and [user-defined types](#) that are installed along with the Q# compiler during installation. The standard library implementation is agnostic with respect to target machines. For more information, see [Standard libraries](#).

## Superposition

The concept in quantum computing that a [qubit](#) is a linear combination of two states,  $\lvert\text{ket}\{0\}\rangle$  and  $\lvert\text{ket}\{1\}\rangle$ , until it is [measured](#). For more information, see [Understanding quantum computing](#).

## Target machine

A compilation target that lowers an abstract quantum program towards hardware or simulation. This typically include re-writes for many purposes including gate replacement, encoding for error correction, geometric layout and others. For more information, see [Quantum simulators and host applications](#).

## Teleportation

A method for regenerating data, or the [quantum state](#), of one [qubit](#) from one place to another without physically moving the qubit, using [entanglement](#) and [measurement](#). For more information, see [Quantum circuits](#) and the respective kata at [Quantum Katas](#).

## Tuple

A collection of comma-separated values that acts as a single value. The *type* of a tuple is defined by the types of values it contains. In Q#, tuples are [immutable](#) and can be nested, contain arrays, or used in an array. For more information, see [Tuples](#).

## Unitary operator

An operator whose inverse is equal to its [adjoint](#), for example,  $UU^{\{\dagger\}} = \text{id}$ .

## User-defined type

A custom type that may contain one or more named or anonymous items. For more information, see [Type declarations](#).

# Learning resources

5/27/2021 • 5 minutes to read • [Edit Online](#)

## Optimization learning resources

The following publications might be good starting points to understand the mathematical principles behind the solvers:

- Glover F, Kochenberger G., Du Y. (2019). A Tutorial on Formulating and Using QUBO Models.
- Lucas, Andrew. (2014) Ising formulations of many NP problems. *Frontiers in Physics*

## Quantum computing learning resources

While the tools discussed in the [quantum computing concepts](#) section are foundational for any developer of quantum software, they by no means span the depth or breadth of what is known about quantum computer programming and algorithm design. Since quantum computing remains a rapidly developing field, there is no one resource that has all of the information needed to learn how to best use these tools in order to solve problems. For this reason we have compiled a reference list that may interest the reader who wishes to learn more about the art and beauty of quantum computer programming. This section contains selected references to deep coverage of quantum computing topics.

### Basic quantum computing

- Nielsen, M. A. & Chuang, I. L. (2010). Quantum Computation and Quantum Information. *Quantum Computation and Quantum Information*, UK: Cambridge University Press, 2010.
- Kaye, P., LaBamme, R., & Mosca, M. (2007). An introduction to quantum computing. Oxford University Press.
- Rieffel, E. G., & Polak, W. H. (2011). Quantum computing: A gentle introduction. MIT Press.
- Sarah C. Kaiser and Christopher E. Granade (Manning Early Access Program began April 2019 Publication in Fall 2020). [Learn Quantum Computing with Python and Q# - A hands-on approach](#).

### Community made content

- [Awesome qsharp](#): An open source list of Q# code and resources.
- [Quantum computing StackExchange](#): an online community for quantum developers to learn, and share their knowledge.
- [Subreddit for the Q# programming language](#): an online community in Reddit to discuss the latest news and developments of Q#.
- [Subreddit for the quantum computing](#): an online community in Reddit to discuss the latest news and developments of quantum computing.
- [Q# Community](#): A community group that collects and maintains projects related to the Q# programming language by a community of folks who are excited about quantum programming.

### Online Courses

- [MS Learn Quantum Computing modules](#). A step-by-step tutorial of quantum computing concepts and practices.
- [Quantum Computing - Brilliant Course](#). Learn to build quantum algorithms from the ground up with a quantum computer simulated in your browser.
- [Introduction to Quantum Computing - LinkedIn Learning](#). 1h25m video introduction.
- [Quantum Computing through Comics](#). Sunday weekly hands-on community class.

### Advanced topics

#### **Elementary techniques for building controlled gates**

- Barenco, A., Bennett, C. H., Cleve, R., DiVincenzo, D. P., Margolus, N., Shor, P., ... & Weinfurter, H. (1995). Elementary gates for quantum computation. *Physical Review A*, 52(5), 3457.
- Jones, C. (2013). Low-overhead constructions for the fault-tolerant Toffoli gate. *Physical Review A*, 87(2), 022328

#### **Techniques for preparing quantum states**

- Shende, V. V., Markov, I. L., & Bullock, S. S. (2004). Minimal universal two-qubit controlled-NOT-based circuits. *Physical Review A*, 69(6), 062321.
- Ozols, M., Roetteler, M., & Roland, J. (2013). Quantum rejection sampling. *ACM Transactions on Computation Theory (TOCT)*, 5(3), 11.
- Grover, L., & Rudolph, T. (2002). Creating superpositions that correspond to efficiently integrable probability distributions. *arXiv preprint quant-ph/0208112*.
- Farhi, E., Goldstone, J., Gutmann, S., & Sipser, M. (2000). Quantum computation by adiabatic evolution. *arXiv preprint quant-ph/0001106*.

#### **Approaches for synthesizing circuits out of H, T and CNOT gates**

- Kliuchnikov, V., Maslov, D., & Mosca, M. (2013). Asymptotically optimal approximation of single qubit unitaries by Clifford and T circuits using a constant number of ancillary qubits. *Physical Review Letters*, 110(19), 190502.
- Ross, N. J., & Selinger, P. (2014). Optimal ancilla-free Clifford+ T approximation of z-rotations. *arXiv preprint arXiv:1403.2975*.
- Kliuchnikov, V. (2013). Synthesis of unitaries with Clifford+ T circuits. *arXiv preprint arXiv:1306.3200*.
- Jones, N. C., Whitfield, J. D., McMahon, P. L., Yung, M. H., Van Meter, R., Aspuru-Guzik, A., & Yamamoto, Y. (2012). Faster quantum chemistry simulation on fault-tolerant quantum computers. *New Journal of Physics*, 14(11), 115023.

#### **Approaches for quantum arithmetic**

- Takahashi, Y., & Kunihiro, N. (2005). A linear-size quantum circuit for addition with no ancillary qubits. *Quantum Information & Computation*, 5(6), 440-448.
- Draper, T. G. (2000). Addition on a quantum computer. *arXiv preprint quant-ph/0008033*.
- Soeken, M., Roetteler, M., Wiebe, N., & De Micheli, G. (2017, March). Design automation and design space exploration for quantum computers. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (pp. 470-475). IEEE.

#### **Methods for fast quantum sampling (amplitude amplification) and probability estimation**

- Brassard, G., Hoyer, P., Mosca, M., & Tapp, A. (2002). Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305, 53-74.
- Grover, L. K. (2005). Fixed-point quantum search. *Physical Review Letters*, 95(15), 150501.
- Berry, D. W., Childs, A. M., & Kothari, R. (2015, October). Hamiltonian simulation with nearly optimal dependence on all parameters. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on* (pp. 792-809). IEEE.

#### **Algorithms for quantum simulation**

- Lloyd, S. (1996). Universal Quantum Simulators. *Science (New York, NY)*, 273(5278), 1073.
- Berry, D. W., Childs, A. M., Cleve, R., Kothari, R., & Somma, R. D. (2015). Simulating Hamiltonian dynamics with a truncated Taylor series. *Physical Review Letters*, 114(9), 090502.
- Low, G. H., & Chuang, I. L. (2017). Optimal Hamiltonian simulation by quantum signal processing. *Physical Review Letters*, 118(1), 010501.
- Low, G. H., & Chuang, I. L. (2016). Hamiltonian simulation by qubitization. *arXiv preprint:1610.06546*.
- Reiher, M., Wiebe, N., Svore, K. M., Wecker, D., & Troyer, M. (2017). Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences*, 201619152.
- Wiebe, N., Berry, D. W., Høyer, P., & Sanders, B. C. (2011). Simulating quantum dynamics on a quantum

computer. *Journal of Physics A: Mathematical and Theoretical*, 44(44), 445308.

- Peruzzo, A., McClean, J., Shadbolt, P., Yung, M. H., Zhou, X. Q., Love, P. J., ... & Obrien, J. L. (2014). A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5.

#### **Procedures for quantum optimization**

- Durr, C., & Høyer, P. (1996). A quantum algorithm for finding the minimum. arXiv preprint quantph/9607014.
- Farhi, E., Goldstone, J., & Gutmann, S. (2014). A quantum approximate optimization algorithm. arXiv preprint arXiv:1411.4028.
- Brandao, F. G., Svore, K. M. (2017). Quantum Speed-ups for Semidefinite Programming. In Annual Symposium on Foundations of Computer Science (FOCS 2017).