Informatique Quantique avec Qiskit

Colin Bossu Réaubourg

29 septembre 2025

Résumé

Ce document constitue un traité de référence complet et technique sur le Kit de Développement Logiciel (SDK) Qiskit. Il est destiné aux étudiants, chercheurs et ingénieurs possédant de solides connaissances en mécanique quantique et une maîtrise de la programmation en Python. Conçu comme un cours personnel, ce guide explore en profondeur l'architecture moderne de Qiskit (v1.0+), de la construction fondamentale des circuits quantiques à leur exécution sur simulateurs haute performance et sur les processeurs quantiques d'IBM.

Nous aborderons systématiquement les concepts clés : la manipulation des circuits, la transpilation, le modèle des Primitives (Sampler et Estimator), la gestion du bruit, ainsi que les techniques avancées comme les circuits paramétrés et la création de portes personnalisées. Enfin, ce traité offre une introduction aux modules applicatifs de plus haut niveau tels que Qiskit Nature et Qiskit Machine Learning, illustrant comment Qiskit sert de pont entre la recherche théorique et l'expérimentation pratique en informatique quantique.

Table des matières

Chapitre 1

Introduction à l'Écosystème Qiskit 1.0+

1.1 Philosophie et Architecture

Qiskit (Quantum Information Science Kit) est un projet open-source initié par IBM, devenu l'un des standards de l'industrie pour la programmation d'ordinateurs quantiques. Sa philosophie repose sur la modularité, l'extensibilité et l'accessibilité, permettant aux utilisateurs d'interagir avec les systèmes quantiques à différents niveaux d'abstraction, du contrôle par impulsions micro-ondes jusqu'aux algorithmes complexes.

Avec la publication de la version 1.0 en 2024, l'écosystème Qiskit a été rationalisé pour améliorer la clarté et la performance. L'architecture moderne s'articule autour de trois piliers fondamentaux :

- `qiskit` : Le paquet principal, qui constitue le cœur du framework. Il contient les briques de base pour définir et manipuler les objets quantiques : circuits (`QuantumCircuit`), portes, registres, et surtout, le puissant compilateur, le **transpiler**.
- `qiskit-aer` : Le simulateur haute performance. Écrit en C++, Aer permet de simuler des circuits quantiques sur des machines classiques avec une grande efficacité. Il peut simuler des systèmes idéaux (vecteur d'état) ainsi que des systèmes bruités (matrice densité), ce qui en fait un outil indispensable pour le développement et le débogage d'algorithmes.
- `qiskit-ibm-runtime` : Le client d'accès aux services cloud d'IBM Quantum. Ce paquet gère l'authentification et la communication avec les processeurs quantiques (QPU) réels et les simulateurs cloud d'IBM. Il implémente le modèle des **Primitives**, une interface optimisée pour l'exécution de tâches quantiques.

Cette séparation claire permet une maintenance et un développement plus agiles, tout en offrant aux utilisateurs la flexibilité de n'installer que les composants dont ils ont besoin.

1.2 Installation et Configuration de l'Environnement

Une installation propre et isolée est cruciale pour éviter les conflits de dépendances. L'utilisation d'un environnement virtuel Python est fortement recommandée.

Recommandation : Environnements Virtuels

Avant toute installation, créez un environnement dédié. Avec `venv` :

```
python -m venv qiskit_env
source qiskit_env/bin/activate  # Sur Linux/macOS
# qiskit_env\Scripts\activate   # Sur Windows
```

Toutes les commandes `pip` suivantes doivent être exécutées dans cet environnement activé.

L'installation des composants principaux se fait via le gestionnaire de paquets `pip`.

```
pip install qiskit qiskit-aer qiskit-ibm-runtime
```

Pour exploiter pleinement les capacités de visualisation de Qiskit, des dépendances supplémentaires sont nécessaires.

```
pip install matplotlib pylatexenc
```

1.2.1 Connexion aux Systèmes IBM Quantum

Pour exécuter des tâches sur le matériel d'IBM, une authentification est requise. Celle-ci se fait via un jeton API que vous pouvez récupérer depuis votre compte IBM Quantum. La méthode recommandée consiste à sauvegarder ce jeton une seule fois sur votre machine.

```
from qiskit_ibm_runtime import QiskitRuntimeService

# Exécutez cette ligne une seule fois en remplaçant par votre jeton
# QiskitRuntimeService.save_account(channel="ibm_quantum", token="VOTRE_TOKEN_API")

# Une fois le jeton sauvegardé, vous pouvez initialiser le service sans argument
service = QiskitRuntimeService()
```

Cette commande sauvegarde le jeton de manière sécurisée dans un fichier de configuration local. Les appels ultérieurs à `QiskitRuntimeService()` le chargeront automatiquement.

Chapitre 2

Le Circuit Quantique : Le Langage de la Computation

Le circuit quantique est le modèle de calcul central en informatique quantique. Il décrit une séquence d'opérations quantiques (portes) appliquées à un ensemble de qubits, suivies de mesures. Dans Qiskit, l'objet `QuantumCircuit` est la pierre angulaire de toute programmation.

2.1 Fondations : Registres et Initialisation

Un `QuantumCircuit` peut être vu comme un conteneur d'instructions agissant sur des registres quantiques et classiques.

2.1.1 Initialisation Simple

La manière la plus directe d'instancier un circuit est de spécifier le nombre de qubits et de bits classiques.

```
from qiskit import QuantumCircuit

# Crée un circuit avec 3 qubits et 3 bits classiques
qc = QuantumCircuit(3, 3)
```

2.1.2 Utilisation de Registres Nommés

Pour des circuits plus complexes, il est préférable d'utiliser des registres nommés (`QuantumRegister`, `ClassicalRegister`) pour améliorer la lisibilité et l'organisation du code.

```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit

qreg_data = QuantumRegister(4, name='data')
qreg_ancilla = QuantumRegister(2, name='ancilla')
creg_result = ClassicalRegister(4, name='res')

qc_named = QuantumCircuit(qreg_data, qreg_ancilla, creg_result)
```

L'avantage de cette approche est que les qubits peuvent être adressés de manière sémantique, par exemple `qreg_data[0]`.

2.2 Répertoire des Portes et Opérations

Les portes quantiques sont appliquées comme des méthodes sur l'objet `QuantumCircuit`.

Exemple Fondamental : L'état de Bell

L'état de Bell $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ est l'exemple canonique d'un état intriqué à deux qubits. Il est créé en appliquant une porte de Hadamard au premier qubit, suivie d'une porte CNOT.

```python
import numpy as np
from qiskit import QuantumCircuit

qc_bell = QuantumCircuit(2, 2)

# Appliquer une porte Hadamard sur le qubit 0
qc_bell.h(0)

# Appliquer une porte CNOT (Controlled-X) avec le qubit 0 comme contrôle
# et le qubit 1 comme cible
qc_bell.cx(0, 1)

# Appliquer une barrière pour la visualisation et la transpilation
qc_bell.barrier()

# Mesurer les qubits 0 et 1 et stocker les résultats dans les bits classiques 0 et 1
qc_bell.measure([0, 1], [0, 1])
```

2.2.1 Portes à Un Qubit

- **Groupe de Pauli** : Opérateurs fondamentaux.
  - `.x(q)` : Porte NOT (bit-flip), $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
  - `.y(q)` : Porte Y, $Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
  - `.z(q)` : Porte Z (phase-flip), $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
- **Porte de Hadamard** : Crée des superpositions.
  - `.h(q)` : $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
- **Portes de Phase** :
  - `.s(q)` : Porte S (Phase, $\sqrt{Z}$), `.sdg(q)` pour son adjointe.
  - `.t(q)` : Porte T ($\sqrt{S}$), `.tdg(q)` pour son adjointe.
  - `.p(theta, q)` : Rotation de phase générale $U1(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$.
- **Rotations Générales** :
  - `.rx(theta, q)`, `.ry(theta, q)`, `.rz(theta, q)`

2.2.2 Portes à Plusieurs Qubits

- **Portes Contrôlées** :

— `.cx(control, target)` : CNOT (Controlled-X).
— `.cy(c, t)`, `.cz(c, t)`, `.ch(c, t)`, etc.
— `.cp(theta, c, t)` : Phase contrôlée.
— `.ccx(c1, c2, t)` : Porte de Toffoli (CCNOT).
— **Autres** :
— `.swap(q1, q2)` : Échange l'état de deux qubits.

2.2.3 Opérations Non-Unitaires

— `.measure(qubit, cbit)` : Projette un qubit sur la base de calcul et stocke le résultat.
— `.reset(q)` : Réinitialise un qubit à l'état $|0\rangle$. C'est une opération physique non-unitaire.
— `.barrier()` : Sépare les sections d'un circuit. Elle n'a pas d'effet physique mais agit comme une directive pour le transpiler, l'empêchant d'optimiser les portes à travers la barrière.

2.3 Composition et Inspection de Circuits

Les circuits complexes sont souvent construits en assemblant des sous-circuits. La méthode `compose` (ou l'opérateur `+`) permet de combiner des circuits.

```
# Circuit 1: prépare un état
qc1 = QuantumCircuit(2, name='state_prep')
qc1.h(0)
qc1.x(1)

# Circuit 2: applique une transformation
qc2 = QuantumCircuit(2, name='transform')
qc2.cx(0, 1)
qc2.rz(np.pi/2, 1)

# Composition
full_qc = qc1.compose(qc2)
```

On peut inspecter les propriétés d'un circuit :
— `qc.depth()` : Profondeur du circuit, une mesure de sa complexité temporelle.
— `qc.width()` : Nombre total de qubits et de bits classiques.
— `qc.count_ops()` : Dictionnaire comptant chaque type de porte.

Chapitre 3

Simulation Locale et Analyse Théorique

Avant d'engager des ressources coûteuses sur un QPU réel, la simulation sur un ordinateur classique est une étape indispensable. Le paquet `qiskit-aer` fournit le `AerSimulator`, un backend de simulation extrêmement performant et configurable.

3.1 Simulation par Échantillonnage (Shots)

Cette méthode est la plus fidèle à une expérience réelle. Le simulateur exécute le circuit un grand nombre de fois (défini par le paramètre `shots`) et collecte les résultats de mesure, qui sont probabilistes par nature.

Le résultat est un dictionnaire de comptes, où les clés sont les états de bits classiques mesurés (sous forme de chaînes de caractères) et les valeurs sont le nombre de fois où cet état a été observé.

```
from qiskit_aer import AerSimulator

# Reprenons notre circuit de Bell
qc_bell = QuantumCircuit(2)
qc_bell.h(0)
qc_bell.cx(0, 1)
qc_bell.measure_all() # Raccourci pour qc.measure([0,1], [0,1])

# Initialisation du simulateur
simulator = AerSimulator()

# Exécution du circuit
job = simulator.run(qc_bell, shots=4096)

# Récupération des résultats
result = job.result()
counts = result.get_counts(qc_bell)

print(f"Résultats de l'échantillonnage : {counts}")
```

Pour un état de Bell idéal, les comptes pour `'00'` et `'11'` devraient être approximativement égaux, et ceux pour `'01'` et `'10'` nuls.

3.2 Simulation du Vecteur d'État Exact

Pour une analyse théorique ou pour le débogage, il est souvent utile d'obtenir le vecteur d'état final complet du système, $|\psi_{final}\rangle$.

Important : Mesure et Vecteur d'État

La mesure étant une opération non-unitaire qui projette l'état, un circuit contenant des mesures terminales ne peut pas avoir de vecteur d'état unique. Pour cette simulation, il faut donc **omettre** les mesures finales ou utiliser une instruction spéciale `save_statevector`.

```python
from qiskit.quantum_info import Statevector

# Circuit sans mesure
qc_sv = QuantumCircuit(2)
qc_sv.h(0)
qc_sv.cx(0, 1)

# Méthode 1: Utiliser l'objet Statevector directement
final_state = Statevector.from_instruction(qc_sv)
print("Vecteur d'état (méthode 1):")
print(final_state)

# Méthode 2: Utiliser AerSimulator avec save_statevector
qc_sv.save_statevector()
simulator = AerSimulator(method='statevector')
job = simulator.run(qc_sv)
result = job.result()
statevector_from_sim = result.get_statevector(qc_sv)
print("\nVecteur d'état (méthode 2):")
print(statevector_from_sim)
```

Le vecteur d'état résultant est un tableau de nombres complexes de dimension 2^N pour N qubits. Pour l'état de Bell, il sera proche de $[1/\sqrt{2}, 0, 0, 1/\sqrt{2}]$.

3.3 Le Module `qiskit.quantum_info`

Ce module est une boîte à outils puissante pour la manipulation d'objets mathématiques de la mécanique quantique, indépendamment de toute simulation.

- **`Statevector`** : Représente un état pur $|\psi\rangle$.
- **`DensityMatrix`** : Représente un état mixte $\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|$. Essentiel pour la simulation de bruit.
- **`Operator`** : Représente un opérateur unitaire U, souvent généré à partir d'un circuit.
- **`SparsePauliOp`** : Une représentation efficace pour les observables (e.g., Hamiltoniens) qui sont des sommes de produits d'opérateurs de Pauli.

```python
from qiskit.quantum_info import Operator, SparsePauliOp

# Créer un opérateur unitaire à partir d'un circuit
```

```
qc_op = QuantumCircuit(2)
qc_op.h(0)
qc_op.cx(0, 1)
U = Operator(qc_op)
print("Matrice unitaire du circuit:")
print(U.data)

# Définir un Hamiltonien simple H = 1.0 * Z_0 Z_1 + 0.5 * X_0 X_1
H = SparsePauliOp(["ZZ", "XX"], coeffs=[1.0, 0.5])
print("\nMatrice de l'observable H:")
print(H.to_matrix())
```

Chapitre 4

Visualisation : De l'Abstrait au Concret

Qiskit intègre des outils de visualisation qui sont essentiels pour comprendre la structure d'un circuit et analyser les résultats d'une exécution.

4.1 Dessin de Circuits

La méthode `.draw()` d'un objet `QuantumCircuit` est l'outil principal. Elle supporte plusieurs backends.

- `'text'` : Un dessin en ASCII, simple et rapide, utile dans un terminal.
- `'mpl'` : Utilise Matplotlib pour créer un schéma de haute qualité, suitable pour les publications. Nécessite `matplotlib`.
- `'latex'` : Génère du code source LaTeX pour une intégration parfaite dans des documents scientifiques.

```
import matplotlib.pyplot as plt

qc_visual = QuantumCircuit(3)
qc_visual.h(0)
qc_visual.cx(0, 1)
qc_visual.cx(0, 2)
qc_visual.measure_all()

# Pour afficher dans une fenêtre ou un notebook
fig = qc_visual.draw('mpl')
plt.show()

# Pour sauvegarder dans un fichier
fig.savefig("circuit_diagram.png")
```

4.2 Visualisation des Résultats

Le module `qiskit.visualization` contient les fonctions nécessaires.

4.2.1 Histogrammes

La fonction `plot_histogram` prend en entrée un dictionnaire de comptes (obtenu après une simulation par échantillonnage) et génère un histogramme.

```
from qiskit.visualization import plot_histogram

# Supposons que 'counts' a été obtenu comme au chapitre précédent
# counts = {'00': 2050, '11': 2046}
# plot_histogram(counts, title="Résultats de la mesure de l'état de Bell")
# plt.show()
```

4.2.2 Représentation des États Quantiques

Pour visualiser un `Statevector`, plusieurs options existent, chacune offrant une perspective différente.

- **`plot_bloch_multivector`** : Affiche la sphère de Bloch pour chaque qubit. Cette représentation est rigoureuse uniquement si les qubits sont décorrélés (non-intriqués). Pour un état intriqué, elle représente la matrice densité réduite de chaque qubit.
- **`plot_state_qsphere`** : La Q-sphere offre une représentation globale de l'état quantique, où la phase est encodée par la couleur et l'amplitude par la taille des points sur une sphère.

```
from qiskit.visualization import plot_bloch_multivector, plot_state_qsphere

# final_state a été obtenu via la simulation de vecteur d'état
# plot_bloch_multivector(final_state, title="Sphères de Bloch pour l'état de Bell")
# plt.show()
#
# plot_state_qsphere(final_state, title="Q-sphere pour l'état de Bell")
# plt.show()
```

Pour l'état de Bell, les sphères de Bloch individuelles montreront un vecteur nul au centre, indiquant une intrication maximale (l'état de chaque qubit est indéfini si l'on ignore l'autre).

Chapitre 5

Exécution sur Matériel Quantique

L'objectif ultime de Qiskit est de servir d'interface avec de véritables processeurs quantiques. Cette interaction est gérée par `qiskit-ibm-runtime` et son modèle basé sur les Primitives, conçu pour minimiser la latence et maximiser l'efficacité de l'utilisation du matériel.

5.1 Le Processus de Transpilation

Transpilation

La transpilation est le processus de compilation qui traduit un circuit quantique abstrait, conçu par l'utilisateur, en une séquence d'instructions exécutables par un backend quantique spécifique. Cette traduction doit respecter deux contraintes majeures du matériel :

1. **Le jeu de portes natif (Basis Gates)** : Chaque QPU ne peut exécuter physiquement qu'un ensemble limité de portes (par ex., CNOT, RZ, SX, X). Toute autre porte doit être décomposée en une séquence équivalente de ces portes de base.
2. **La topologie de couplage (Coupling Map)** : Les portes à deux qubits (comme CNOT) ne peuvent être appliquées qu'entre des paires de qubits physiquement connectés.

La transpilation inclut également une phase d'optimisation agressive pour réduire la profondeur et le nombre de portes du circuit, afin de minimiser l'impact du bruit (décohérence).

La fonction `transpile` est l'outil central de ce processus.

```
from qiskit import transpile
from qiskit_ibm_runtime import QiskitRuntimeService

# service = QiskitRuntimeService()

# Sélectionner un backend réel (le moins occupé)
# backend = service.least_busy(simulator=False, operational=True)
# print(f"Backend sélectionné : {backend.name}")

# Circuit abstrait
qc = QuantumCircuit(3)
qc.h(0)
qc.cx(0, 2) # Supposons que les qubits 0 et 2 ne sont pas connectés directement
```

```

# Transpilation
# L'optimization_level (0 à 3) contrôle l'agressivité de l'optimisation.
# 3 est le plus poussé, mais plus long à compiler.
# transpiled_qc = transpile(qc, backend, optimization_level=3)

# On peut visualiser le circuit transpilé pour voir les changements
# transpiled_qc.draw('mpl')
# plt.show()
```

Le circuit transpilé contiendra typiquement des portes SWAP pour déplacer les états des qubits afin de réaliser des CNOT entre des qubits non-adjacents, et toutes les portes seront décomposées dans la base du backend.

5.2 Le Modèle des Primitives V2

Les Primitives V2 (`SamplerV2` et `EstimatorV2`) sont des programmes pré-définis sur les serveurs d'IBM qui sont optimisés pour des tâches spécifiques. L'envoi de circuits à ces primitives est la méthode moderne d'exécution.

L'utilisation d'une `Session` est recommandée. Une session regroupe une série de jobs sur un même backend, ce qui réduit la latence en évitant les files d'attente répétées.

5.2.1 SamplerV2 : Échantillonnage de Distributions

Le `Sampler` est l'analogue de la simulation par échantillonnage, mais pour le matériel réel. Il retourne une distribution de quasi-probabilités des résultats de mesure.

```
from qiskit_ibm_runtime import SamplerV2 as Sampler

# qc_bell est notre circuit de Bell avec mesures
# transpiled_bell_qc = transpile(qc_bell, backend)

# sampler = Sampler(backend=backend)
# job = sampler.run([transpiled_bell_qc], shots=4096)
# print(f"Job ID: {job.job_id()}")

# result = job.result()
# pub_result = result[0] # Primitive Unified Bloc result

# Les données contiennent des informations binaires brutes
# Pour obtenir les comptes, on peut utiliser get_counts()
# counts = pub_result.data.meas.get_counts()
# print(f"Comptes obtenus du QPU : {counts}")
# plot_histogram(counts)
# plt.show()
```

Contrairement à la simulation idéale, les résultats d'un QPU réel montreront des comptes non-nuls pour les états `'01'` et `'10'` en raison du bruit.

5.2.2 EstimatorV2 : Calcul de Valeurs d'Espérance

L'`Estimator` est conçu pour une tâche fondamentale : calculer la valeur d'espérance d'un ou plusieurs observables $\langle O \rangle$ pour un état préparé par un circuit, $\langle O \rangle = \langle \psi | O | \psi \rangle$. C'est le cœur des algorithmes variationnels comme le VQE.

```
from qiskit_ibm_runtime import EstimatorV2 as Estimator
from qiskit.quantum_info import SparsePauliOp

# Circuit préparant un état (sans mesure)
qc_est = QuantumCircuit(2)
qc_est.h(0)
qc_est.cx(0, 1)
# transpiled_est_qc = transpile(qc_est, backend)

# Observable à mesurer
observable = SparsePauliOp.from_list([("ZZ", 1), ("IX", -0.5)])

# estimator = Estimator(backend=backend)
# job = estimator.run([(transpiled_est_qc, observable)])
# print(f"Job ID: {job.job_id()}")

# result = job.result()
# pub_result = result[0]
# expectation_value = pub_result.data.evs
# print(f"Valeur d'espérance <H> : {expectation_value}")
```

L'Estimator gère en interne la transpilation des circuits et l'ajout des mesures appropriées pour évaluer chaque terme de Pauli de l'observable.

Chapitre 6

Techniques Avancées de Construction de Circuits

6.1 Circuits Paramétrés

Pour les algorithmes variationnels ou l'apprentissage automatique quantique, il est nécessaire de créer des circuits dont les portes dépendent de paramètres classiques optimisables. Qiskit gère cela de manière très efficace avec la classe `Parameter`.

L'avantage est que le circuit est défini et transpilé une seule fois. Ensuite, seules les valeurs des paramètres sont envoyées au backend pour chaque évaluation, ce qui est beaucoup plus rapide.

```python
from qiskit.circuit import Parameter, ParameterVector

# Définir des paramètres
theta = Parameter('theta')
phi_vec = ParameterVector('phi', length=2)

# Construire un circuit paramétré
pqc = QuantumCircuit(2)
pqc.ry(theta, 0)
pqc.crz(phi_vec[0], 0, 1)
pqc.rx(phi_vec[1], 1)

print(f"Paramètres du circuit : {pqc.parameters}")

# Assigner des valeurs pour une exécution unique
qc_bound = pqc.assign_parameters({theta: np.pi/2, phi_vec: [0.1, 0.2]})

# Avec les Primitives, on passe directement les valeurs lors de l'appel
# observable = SparsePauliOp("ZZ")
# job = estimator.run([(pqc, observable, [[np.pi/2, 0.1, 0.2], [np.pi/4, 0.3, 0.4]])])
```

6.2 Création de Portes Personnalisées

Pour des algorithmes complexes, il est essentiel d'abstraire des sous-routines en portes uniques. Cela améliore la lisibilité et la modularité.

— `.to_instruction()` : Convertit un circuit en une "boîte noire". Le transpiler ne la décomposera pas.
— `.to_gate()` : Convertit un circuit en une porte à part entière. Cette porte peut ensuite être contrôlée, inversée, etc.

```python
# Créer un sous-circuit, par exemple un diffuseur de Grover pour 2 qubits
diffuser_qc = QuantumCircuit(2, name='Diffuser')
diffuser_qc.h([0, 1])
diffuser_qc.x([0, 1])
diffuser_qc.cz(0, 1)
diffuser_qc.x([0, 1])
diffuser_qc.h([0, 1])

# Convertir en une porte
diffuser_gate = diffuser_qc.to_gate()

# Créer une version contrôlée de cette porte
controlled_diffuser = diffuser_gate.control(1, label='C-Diffuser')

# Utiliser dans un circuit plus grand
main_qc = QuantumCircuit(3)
main_qc.append(diffuser_gate, [0, 1])
main_qc.barrier()
main_qc.h(2)
main_qc.append(controlled_diffuser, [2, 0, 1])

main_qc.draw('mpl')
plt.show()
```

6.3 La Bibliothèque de Circuits Standard

Qiskit fournit une riche bibliothèque de circuits standards dans `qiskit.circuit.library`. Il est souvent inutile de réimplémenter des circuits connus comme la Transformée de Fourier Quantique (QFT) ou les ansatzes pour le VQE.

```python
from qiskit.circuit.library import QFT, TwoLocal

# Obtenir un circuit QFT pour 4 qubits
qft_circuit = QFT(4)
qft_circuit.decompose().draw('mpl')
plt.show()

# Créer un ansatz variationnel standard
ansatz = TwoLocal(num_qubits=3, rotation_blocks=['ry', 'rz'], entanglement='cx')
ansatz.decompose().draw('mpl')
plt.show()
```

Chapitre 7

Introduction aux Modules Applicatifs

Au-delà de la manipulation de circuits, l'écosystème Qiskit inclut des modules applicatifs qui fournissent des outils de plus haut niveau pour résoudre des problèmes dans des domaines spécifiques. Ces modules s'appuient sur les fondations présentées précédemment.

7.1 Qiskit Nature : Chimie Quantique et Physique

Qiskit Nature est dédié à la simulation de systèmes naturels. Son application principale est la chimie quantique, notamment le calcul de l'état fondamental de molécules, un problème notoirement difficile pour les ordinateurs classiques.

Le flux de travail typique dans Qiskit Nature est le suivant :

1. **Définition du problème physique** : Spécifier la structure moléculaire (atomes, positions).
2. **Mapping vers un problème de qubits** : Qiskit Nature gère la transformation du Hamiltonien électronique (opérateurs fermioniques) en un Hamiltonien de qubits (opérateurs de Pauli) via des mappings comme Jordan-Wigner ou Bravyi-Kitaev.
3. **Résolution algorithmique** : Utiliser un algorithme quantique, typiquement le VQE (`VQE`), avec un ansatz adapté (par exemple, `UCCSD`), pour trouver l'énergie de l'état fondamental.
4. **Post-traitement** : Extraire des propriétés chimiques à partir des résultats.

Ce module abstrait une grande partie de la complexité de la physique sous-jacente, permettant aux chercheurs de se concentrer sur l'aspect quantique de la simulation.

7.2 Qiskit Machine Learning : Apprentissage Automatique Quantique

Ce module explore l'intersection de l'informatique quantique et de l'apprentissage automatique. Il fournit des outils pour construire, entraîner et utiliser des modèles d'apprentissage automatique quantiques. Les principaux domaines incluent :

— **Noyaux Quantiques (Quantum Kernels)** : L'idée est d'utiliser un circuit quantique pour "mapper" les données classiques dans un espace de Hilbert quantique, potentiellement très grand. On peut ensuite utiliser un algorithme d'apprentissage classique, comme les machines à vecteurs de support (SVM), avec ce noyau quantique. Qiskit fournit la classe `QSVC` (Quantum Support Vector Classifier).

— **Réseaux de Neurones Quantiques (QNNs)** : Ce sont des modèles hybrides où un circuit quantique paramétré est utilisé comme une couche dans un réseau de neurones. L'entraînement se fait par descente de gradient, en utilisant des techniques comme le "parameter shift rule" pour calculer les gradients du circuit quantique.

Ces modules sont en développement rapide et représentent des domaines de recherche actifs visant à trouver un avantage quantique pour des problèmes pratiques.

Annexe A

Annexe : Exemples de Code Concrets

Cette annexe fournit une série d'exemples de code complets et prêts à l'emploi, illustrant des cas d'usage variés de l'écosystème Qiskit.

A.1 Création, Simulation et Visualisation d'un État de Bell

Cet exemple montre le flux de travail le plus fondamental : construire un circuit, l'exécuter sur un simulateur local, récupérer les résultats et les visualiser sous forme d'histogramme.

```python
import numpy as np
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

qc_bell = QuantumCircuit(2, 2)
qc_bell.h(0)
qc_bell.cx(0, 1)
qc_bell.barrier()
qc_bell.measure([0, 1], [0, 1])

simulator = AerSimulator()

job = simulator.run(qc_bell, shots=8192)
result = job.result()
counts = result.get_counts(qc_bell)

print(f"Comptes obtenus : {counts}")

fig = plot_histogram(counts, title="Distribution des résultats de l'état de Bell")
plt.show()
```

A.2 Protocole de Téléportation Quantique

Ici, nous simulons la téléportation de l'état d'un qubit (`q0`) vers un autre (`q2`) en utilisant une paire de qubits intriqués (`q1`, `q2`) et un canal de communication classique.

```python
import numpy as np
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.quantum_info import Statevector

q = QuantumRegister(3, name="q")
c_z = ClassicalRegister(1, name="cz")
c_x = ClassicalRegister(1, name="cx")
qc = QuantumCircuit(q, c_z, c_x)

# Créer un état initial aléatoire pour q0
random_state = [np.sqrt(0.3), np.sqrt(0.7)]
qc.initialize(random_state, 0)
qc.barrier()

# Préparer la paire de Bell pour la téléportation (q1, q2)
qc.h(1)
qc.cx(1, 2)
qc.barrier()

# Protocole de téléportation
qc.cx(0, 1)
qc.h(0)
qc.measure(0, c_z)
qc.measure(1, c_x)
qc.barrier()

# Application des corrections sur q2
with qc.if_test((c_x, 1)):
    qc.x(2)
with qc.if_test((c_z, 1)):
    qc.z(2)

initial_sv = Statevector(random_state)
final_sv = Statevector(qc).partial_trace([0, 1])

print("Vecteur d'état initial du qubit 0 :")
print(initial_sv)
print("\nVecteur d'état final du qubit 2 (après téléportation) :")
print(final_sv)
```

A.3 Algorithme de Grover pour la Recherche

Cet exemple implémente l'algorithme de Grover pour trouver l'état $|11\rangle$ dans un système à 2 qubits. Il illustre la construction d'un oracle et d'un circuit de diffusion.

```python
from qiskit import QuantumCircuit, Aer
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

# Oracle pour marquer l'état |11>
```

```
oracle = QuantumCircuit(2, name='oracle')
oracle.cz(0, 1)
oracle_gate = oracle.to_gate()

# Circuit de diffusion de Grover
diffuser = QuantumCircuit(2, name='diffuser')
diffuser.h([0, 1])
diffuser.x([0, 1])
diffuser.cz(0, 1)
diffuser.x([0, 1])
diffuser.h([0, 1])
diffuser_gate = diffuser.to_gate()

# Circuit principal de Grover
n_qubits = 2
grover_qc = QuantumCircuit(n_qubits, n_qubits)
grover_qc.h(range(n_qubits))
grover_qc.append(oracle_gate, [0, 1])
grover_qc.append(diffuser_gate, [0, 1])
grover_qc.measure(range(n_qubits), range(n_qubits))

simulator = Aer.get_backend('qasm_simulator')
job = simulator.run(grover_qc, shots=1024)
result = job.result()
counts = result.get_counts()

plot_histogram(counts)
plt.show()
\end{rinted}

\section{Exécution sur un Backend IBM Quantum avec les Primitives}
L'exemple suivant montre le flux de travail moderne pour exécuter un circuit sur un vrai QPU, en utilisant la transpilation et la primitive \texttt{SamplerV2}.

\begin{minted}{python}
from qiskit import QuantumCircuit, transpile
from qiskit_ibm_runtime import QiskitRuntimeService, SamplerV2 as Sampler
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

# QiskitRuntimeService.save_account(channel="ibm_quantum", token="VOTRE_TOKEN_API", overwrite=True)
service = QiskitRuntimeService()

# Sélectionner le backend le moins occupé
backend = service.least_busy(simulator=False, operational=True)
print(f"Backend sélectionné : {backend.name}")

# Circuit de l'état GHZ à 3 qubits
qc_ghz = QuantumCircuit(3)
qc_ghz.h(0)
qc_ghz.cx(0, 1)
qc_ghz.cx(0, 2)
```

```
qc_ghz.measure_all()

# Transpilation pour le backend cible
transpiled_qc = transpile(qc_ghz, backend)

# Exécution avec le Sampler
sampler = Sampler(backend=backend)
job = sampler.run([transpiled_qc], shots=4096)
print(f"Job ID: {job.job_id()}")

# Récupération des résultats
result = job.result()
pub_result = result[0]
counts = pub_result.data.meas.get_counts()

print(f"Comptes obtenus du QPU : {counts}")
plot_histogram(counts)
plt.show()
```

A.4 Calcul de l'Énergie Fondamentale de H_2 avec Qiskit Nature

Ce dernier exemple est un cas d'usage avancé utilisant le module applicatif Qiskit Nature pour résoudre un problème de chimie quantique : trouver l'énergie de l'état fondamental de la molécule d'hydrogène.

```
import numpy as np
from qiskit_nature.units import DistanceUnit
from qiskit_nature.second_q.drivers import PySCFDriver
from qiskit_nature.second_q.mappers import JordanWignerMapper
from qiskit_nature.second_q.circuit.library import UCCSD, HartreeFock

from qiskit_algorithms.minimum_eigensolvers import VQE
from qiskit_algorithms.optimizers import SLSQP
from qiskit.primitives import Estimator
from qiskit_aer.primitives import Estimator as AerEstimator


# Étape 1: Définir le problème moléculaire
driver = PySCFDriver(
    atom=f"H 0 0 0; H 0 0 0.735",
    basis="sto3g",
    charge=0,
    spin=0,
    unit=DistanceUnit.ANGSTROM,
)
problem = driver.run()
mapper = JordanWignerMapper()

# Étape 2: Définir l'algorithme VQE
estimator = AerEstimator()
optimizer = SLSQP()
```

```
ansatz = UCCSD(
    num_spatial_orbitals=problem.num_spatial_orbitals,
    num_particles=problem.num_particles,
    mapper=mapper,
    initial_state=HartreeFock(
        num_spatial_orbitals=problem.num_spatial_orbitals,
        num_particles=problem.num_particles,
        mapper=mapper,
    ),
)
vqe_solver = VQE(estimator, ansatz, optimizer)
vqe_solver.initial_point = np.zeros(ansatz.num_parameters)

# Étape 3: Exécuter le solveur
hamiltonian = mapper.map(problem.hamiltonian.second_q_op())
result = vqe_solver.compute_minimum_eigenvalue(hamiltonian)

print("Résultat du VQE :")
print(result.eigenvalue)
```