SOFTWARE-QUANTENSIMULATOREN IN C#

Marke Eigenbau

Zurzeit sind Quantensimulatoren die am besten zugänglichen Werkzeuge für den Entwurf und das Testen von Quantenalgorithmen. Warum diese nicht mit C# selber bauen?

Echte Quantencomputer, also hardwaretechnische Systeme, die mit quantenmechanischen Zuständen arbeiten, sind extrem aufwendig, fehleranfällig und teuer. Daher wird das Computing mit Quantencomputern nach Möglichkeit im Vorfeld simuliert. Neuerdings möchte man allerdings Aufgaben, für deren Lösung in der Vergangenheit aufwendige Quantencomputer notwendig waren, direkt durch Simulatoren lösen. So gibt es in der Quantenchemie schon erste praktische Anwendungen in der Quantensimulation von Materialien.

Für Forschung und Lehre stellen Quantensimulatoren somit einen extrem wichtigen Lösungsansatz dar, um kostengünstig und effektiv variationelle Quantenalgorithmen zu entwickeln und zu testen. Diese Entwicklung kann man auch sehr gut im Quantencomputer-Infoportal Quantiki [1] beobachten. Auf der Webseite gibt es immer wieder neue Simulatoren für Quantencomputer für eine Vielzahl von Programmiersprachen. Zurzeit finden Sie dort rund 150 verschiedene Simulatoren, darunter viele von namhaften Anbietern wie Microsoft, IBM oder dem Fraunhofer-Institut.

Ein einfacher Simulator für das QDK

Wie man unter Einsatz des Microsoft Quantum Development Kits (QDK) einen eigenen reversiblen Simulator in der Programmiersprache C# erstellen kann, zeigt das nachfolgende Beispiel. Als Grundlage genutzt wird der von Microsoft vorgestellte Code aus dem Quellcode-Repository. Grundkenntnisse in der C#-Programmierung werden dabei vorausgesetzt.

Der im Beispielfall vorgestellte Simulator kann nur Quantenprogramme simulieren, die aus klassischen Operationen wie X, CNOT, CCNOT oder beliebig kontrollierten X-Operationen bestehen. Der Simulator stellt den Quantenzustand dar, indem er jedem Qubit einen booleschen Wert zuordnet. Damit ist das Testen von Quantenoperationen möglich, die boolesche Funktionen auswerten.

Download und Start des Simulators in C#

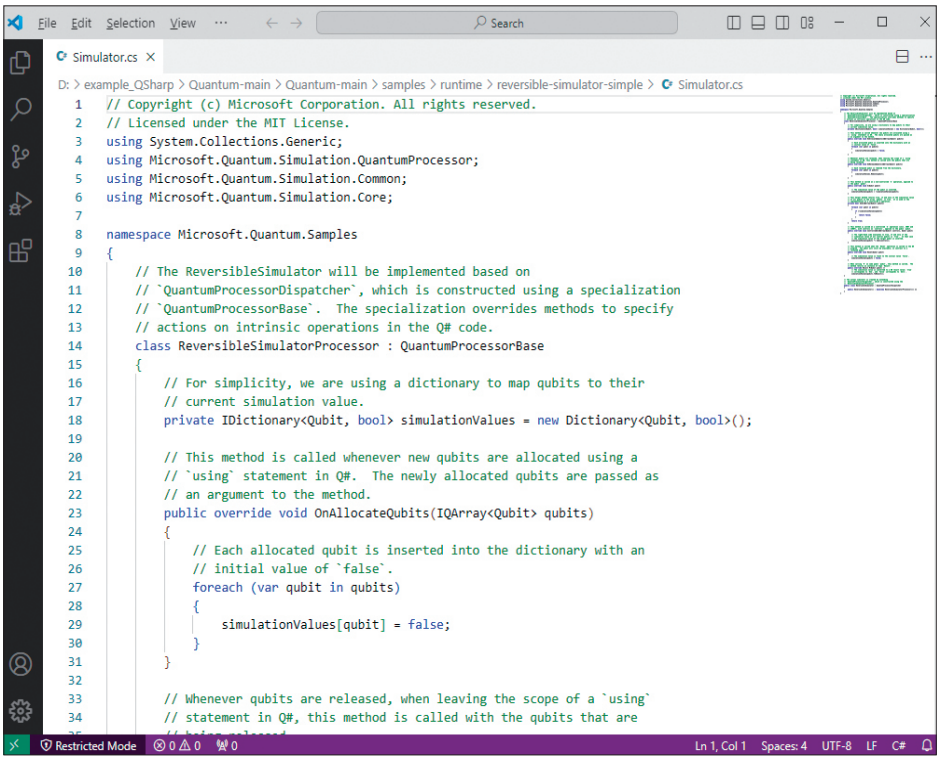
Als Entwickungsumgebung für dieses Programmbeispiel wird Visual Studio Code in Verbindung mit dem C# Dev Kit und dem Quantum Development Kit (QDK) verwendet.

Um das hier vorgestellte Beispiel nachvollziehen zu können, benötigen Sie das aktuelle GitHub-Repository *microsoft/Quantum* als Download-ZIP-Datei [2].

Nachdem Sie das benötigte Repository eingerichtet beziehungsweise die ZIP-Datei entpackt haben, können Sie auch schon mit dem Simulator-Beispiel beginnen. Öffnen Sie hierfür die Datei *Simulator.cs* unter der Verzeichnungsstruktur *Quantum-main/samples/runtime/reversible-simulator-simple* mit Visual Studio Code (Bild 1).

Die Klasse ReversibleSimulatorProcessor

Den Einstieg in die Erstellung eines eigenen Simulators ermöglicht Ihnen die Klasse *ReversibleSimulatorProcessor*, indem Sie die *QuantumProcessorBase*-Klasse erweitern. Das *IQuantumProcessor*-Interface macht es für Sie sehr einfach, eigene Simulatoren für klassische Operationen zu schreiben.



Die Datei Simulator.cs in Visual Studio Code (Bild 1)

```
class ReversibleSimulatorProcessor :
    QuantumProcessorBase {
  private IDictionary<Qubit, bool>
    simulationValues = new Dictionary<Qubit, bool>();
  // Aktion und Methoden zum Definieren von Operationen
}
```

Der Klasse wird im Beispielcode ein *Dictionary* hinzugefügt, das den aktuellen Simulationswert für jedes Qubit im Programm speichert. Die Quantenzustände |0⟩ und |1⟩ werden als boolesche Werte mit *True* und *False* dargestellt.

Die Basisklasse *QuantumProcessorBase* enthält eine Methode, die es ermöglicht, jede intrinsische Operation [3], die Sie überschreiben möchten, zu definieren, um ihr Verhalten im neuen Simulator festzulegen.

Achten Sie bei der Verwendung eines reversiblen Simulators darauf, keine Quantenprogramme zu simulieren, die nichtklassische Operationen wie Hadamard-Gatter [4] und Quanten-T-Gatter [5] enthalten.

Im nächsten Schritt beginnen Sie damit, neu zugewiesene Qubits mit *false* zu initialisieren, indem Sie die Methode *OnAllocateQubits* überschreiben:

```
public override void OnAllocateQubits(IQArray<Qubit>
    qubits) {
  foreach (var qubit in qubits) {
    simulationValues[qubit] = false;
  }
}
```

Über die Methode *OnReleaseQubits* werden Qubits aus dem *Dictionary* entfernt, wenn sie freigegeben werden.

```
public override void OnReleaseQubits(IQArray<Qubit>
    qubits) {
  foreach (var qubit in qubits) {
    simulationValues.Remove(qubit);
  }
}
```

Diese beiden Quantenoperationen gewährleisten, dass die Simulationswerte immer dann zur Verfügung stehen, wenn eine Operation auf ein Qubit angewendet wird, und dass keine Simulationswerte für Qubits im *Dictionary* vorhanden sind, zu denen es im aktuellen Kontext keinen Bezug mehr gibt.

Die beiden im weiteren Fortgang verwendeten Methoden *X()* und *ControlledX()* implementieren die Aktionen der klassischen Operationen:

```
public override void X(Qubit qubit) {
  simulationValues[qubit] = !simulationValues[qubit];
}

public override void ControlledX(IQArray<Qubit>
    controls, Qubit qubit) {
  simulationValues[qubit] ^= And(controls);
}
```

Listing 1: Q#–Mehrheitsoperation

```
operation ApplyMajority(a : Qubit, b : Qubit,
    c : Qubit, f : Qubit) : Unit {
  within {
    CNOT(b, a);
    CNOT(b, c);
  } apply {
    CCNOT(a, c, f);
    CNOT(b, f);
  }
}

operation RunMajority(a : Bool, b : Bool, c : Bool)
: Bool {
  using ((qa, qb, qc, f) =
      (Qubit(), Qubit(), Qubit(), Qubit())) {
    within {
      ApplyPauliFromBitString(
        PauliX, true, [a, b, c], [qa, qb, qc]);
    } apply {
      ApplyMajority(qa, qb, qc, f);
    }
    return MResetZ(f) == One;
  }
}
```

Die Methode *X()* wird aufgerufen, wenn eine einfache X-Operation, also eine direkte Aktion mit einem Qubit, simuliert wird. Eine willkürlich kontrollierte X-Operation kann hierbei auch die Simulation von CNOT- und CCNOT-Gattern umfassen. Die Wirkung einer X-Operation invertiert den Simulationswert eines Qubits, während im Fall einer willkürlich kontrollierten X-Operation das Ziel-Qubit nur dann invertiert wird, wenn alle Kontroll-Qubits mit *true* belegt sind.

Die Methode *Result* liefert die Messung eines Qubits in einem Q#-Programm mit dem Ergebnis *One* oder *Zero*. Die *Reset*-Methode entfernt in diesem Fall nicht das letzte Qubit aus dem aktuellen Kontext, sondern setzt den Simulationswert auf seinen Anfangswert *false* zurück.

```
public override Result M(Qubit qubit) {
  return simulationValues[qubit] ? Result.One :
    Result.Zero;
}

public override void Reset(Qubit qubit) {
  simulationValues[qubit] = false;
}
```

Die *ReversibleSimulatorProcessor*-Klasse sollte als Simulator verwendet werden, indem eine *QuantumProcessorDispatcher*-Instanz erstellt wird. Das Quantum-Entwicklerteam rät, immer eine eigene Klasse für den Simulator zu erstellen. ▶

```
public class ReversibleSimulator :
    QuantumProcessorDispatcher {
  public ReversibleSimulator() :
    base(new ReversibleSimulatorProcessor()) {}
}
```

Hiermit haben Sie Ihren ersten eigenen Simulator in Verbindung mit dem Quantum Development Kit für Q# erstellt.

Verwendung des neuen Simulators

Ist die Implementierung des Simulators abgeschlossen, können Sie anschließend den neuen Simulator mit einer Q#-Operation ausführen, die eine Mehrheitsoperation durchführt, indem sie drei boolesche Eingabewerte bereitstellt. Listing 1 zeigt die Implementierung dieser Q#-Operation – siehe auch den Artikel zur Programmierung mit Q# unter [6] in dieser dotnetpro-Ausgabe – in einer *.qs-Datei für Visual Studio Code.

Zum Schluss können Sie alles zusammenfügen, indem Sie die Q#-Operation mit dem neu erstellten Simulator in einem C#-Host-Programm ausführen. Listing 2 zeigt den Aufruf der Q#-Operation aus dem C#-Code heraus. Der Code wertet die Mehrheitsoperation für alle verschiedenen Eingaben aus und gibt alle Simulationsergebnisse zurück.

Dieses Beispiel sollte aufzeigen, wie Sie alternative Anwendungen für benutzerdefinierte Simulatoren im Umfeld des Quantum Development Kits implementieren können.

C#-Quantengatter-Simulator

Der oben vorgestellte Quantensimulator oder auch alternativ der QDK-Simulator (siehe [6]) unterstützen eine Vielzahl von Quantengattern und -schaltungen. Bei der Entwicklung im Quantum Computing handelt es sich in den meisten Fällen um eine sehr kleine Teilmenge von Quantengattern und Mechanismen, die tatsächlich genutzt werden.

Der Vorteil, wenn man etwas von Grund auf neu entwickelt: Man erhält eine Menge Wissen darüber, wie bestimmte Vorgänge und Verfahren konkret funktionieren. Im Quantum Computing werden viele Theorien in mathematischen Operationen ausgedrückt – siehe den Artikel zum Quantum Computing unter [7] in dieser dotnetpro-Ausgabe. Diese Operationen werden dann in den tatsächlichen Quantenschaltkreisen auf einer realen Hardware simuliert. Bei der Entwicklung des Codes fällt aber auch auf, dass dieser in erster Linie aus einer Reihe von mathematischen Schritten besteht.

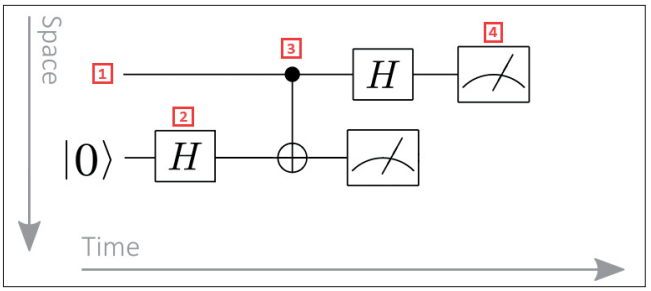


Bild: Microsoft [10]

Hadamard-Quantenschaltkreis (Bild 2)

Das nachfolgende C#-Programm soll als Simulator in der Lage sein, verschiedene Quantengatter-Operationen auf einem Zwei-Qubit-System zu simulieren. Mit diesem Programm können Sie Operationen wie Pauli-X [8], oft auch einfach nur als X-Gatter bezeichnet, Hadamard [4] und CNOT (Controlled-not-Gate) [9] simulieren.

Für Quantengatter, die in diesem Fall auf zwei Qubits arbeiten, ist eine Wechselwirkung zwischen den fraglichen Qubits erforderlich. Das X-Gatter verschränkt ein Qubit-Paar, das Hadamard-Gatter sorgt für Überlagerungszustände, und beim CNOT-Gatter wird abhängig vom Zustand des Kontroll-Qubits der Zustand des Ziel-Qubits entweder negiert oder beibehalten. Bild 2 zeigt ein Beispiel anhand eines Hadamard-Quantenschaltungsdiagramms [10]. Hierbei wird das Qubit-Register (1) als horizontale Linien angezeigt. Die Quantenoperationen werden durch das verwendete Quantengatter (2) dargestellt. Das kontrollierte Gatter (3) wirkt auf zwei oder mehr Qubits. Im Beispiel stellt das Symbol ein CNOT-Gatter dar. Über Punkt 4 (4) erfolgt der Messvorgang. Hierbei ist zu beachten, dass in einer Quantenschaltung die Zeit immer von links nach rechts fließt.

Quantengesteuerte Gatter sind immer Zwei-Qubit-Gatter, die ein Ein-Qubit-Gatter auf ein Ziel-Qubit anwenden, wenn sich ein Kontroll-Qubit in einem bestimmten Zustand befindet. Beachten Sie, dass für Multi-Qubit-Vorgänge die gleichen Konventionen wie für Ein-Qubit-Vorgänge gelten.

Die Aufgabe

Das hier vorgestellte Simulationsprogramm soll programmtechnisch dem Quantenschaltungsdiagramm aus Bild 3 folgen. Das heißt, beim Starten des Programms wird standardmäßig der Anfangszustand |0⟩ verwendet.

Sobald der Start des Programms abgeschlossen ist, fordert der Simulator Sie auf, eine Sequenz von Quantengattern an-

Listing 2: Ausführung des neuen Simulators mit Q#-Operation

```
public static void Main(string[] args) {
  var sim = new ReversibleSimulator();
  var bits = new[] {false, true};

  foreach (var a in bits) {
    foreach (var b in bits) {
      foreach (var c in bits) {
        var f = RunMajority.Run(sim, a, b, c).Result;
        Console.WriteLine(
          $"Majority({a,5}, {b,5}, {c,5})  =  {f,5}");
      }
    }
  }
}
```

zuwenden. Wenn Sie in die Konsole beispielsweise *h0,cnot* eingeben, wird der Simulator angewiesen, das Hadamard-Gatter auf Qubit |0⟩ anzuwenden, gefolgt von einer CNOT-Gatteroperation. Im Anschluss an diese Zustandsmanipulation führt der Simulator eine definierte Anzahl von Iterationen durch (im Beispiel sind es 100) und simuliert die Entwicklung des Quantenzustands mit jedem Schritt.

Am Ende des Prozesses wird in der Konsole die Verteilung der Quantenzustände ausgegeben.

Implementierung in C#

Zur Erstellung des Simulators wird als Entwicklungsumgebung Visual Studio 2022 Version (in einer Version ab Community Edition) mit dem .NET Framework 8.0 verwendet. Alternativ können Sie auch .NET Framework 6.0 benutzen.

Legen Sie für das Beispiel ein neues Console-App-Projekt in Visual Studio an. Vergeben Sie als *Project name* den Begriff *QuantumSimulator* und wählen Sie .NET 8.0 als Framework aus. Nach dem Klick auf die Schaltfläche *Create* erstellt Visual Studio automatisch ein lauffähiges Konsolenprogramm mit der Ausgabe *Hello World!*

Im nächsten Schritt können Sie für das Beispiel die beiden Codezeilen in der Klasse *Program.cs* entfernen.

Der Programmcode in Listing 3 stellt den kompletten Beispiel-Simulator dar. Die definierten Variablen für die Simulationsdaten werden beim Start alle mit 0 initialisiert. Mit deren Hilfe werden die Qubits programmtechnisch nachgebildet. Im Beispiel wird gedanklich von zwei Bits ausgegangen. Diese beiden Bits können sich in verschiedenen Kombinationen von Zuständen befinden, denn wenn eines 0 ist, kann das andere entweder 0 oder 1 sein und umgekehrt. Die möglichen Zustände sind also *00, 01, 10* und *11*. Mathematisch gesehen kann es bei n Bits insgesamt 2^n Kombinationen geben. In dem Beispiel gehen wir von der Annahme aus, dass man ein einzelnes ursprüngliches Bit in zwei Listen darstellen kann. Bit 0 ist [1,0] und Bit 1 ist [0,1].

Ein Qubit stellt fast dasselbe dar. Die beiden Zustände eines einzelnen Qubits werden ja immer als |0⟩ oder |1⟩ dargestellt und sind damit in Form eines Spaltenvektors wieder gleich: Qubit |0⟩ ist [1 0] und Qubit |1⟩ ist [0 1]. Das heißt, ein Qubit kann sich entweder in einem Zustand |0⟩ oder im Zustand |1⟩ befinden, oder in einem beliebigen Zustand dazwischen. Das bezeichnet den Superpositionszustand des Qubits.

Aus dem allgemeinen Spaltenvektor eines Qubits können Sie ersehen, dass wenn *b=0* ist, Sie |0⟩ erhalten, und wenn *a=0* ist, Sie |1⟩ erhalten. Dies unterscheidet den Spaltenvektor eines klassischen Bits von einem Spaltenvektor eines Qubits. Bei klassischen Bits kann der Spaltenvektor in der Zeile nie etwas anderes als 1 enthalten, und es kann immer nur eine der Zeilen 1 sein. Bei Qubits können in jeder Zeile des Spaltenvektors beliebige komplexe Zahlen stehen, solange diese alle quadriert zu 1 summiert werden.

Quantengatter

Qubits oder besser gesagt die Simulation von Qubits reichen allein aber nicht aus. Sie müssen auch einige Operationen mit ihnen durchführen. Hier kommen die Quantengatter ins ▶

Listing 3: Implementierung des Simulators (Teil 1)

```
using System;

namespace QuantumSimulator
{
  class Program
  {
    static double r0 = 1; static double r1 = 0;
      static double r2 = 0; static double r3 = 0;
    static double i0 = 0; static double i1 = 0;
      static double i2 = 0; static double i3 = 0;
    static double a0 = 0; static double a1 = 0;
    static double b0 = 0; static double b1 = 0;
    static int shots = 100;
    static void Main(string[] args)
    {
      Console.Clear();
      Console.WriteLine(
        "C# Quanten-Gatter-Simulator");
      Console.WriteLine(
        "Mögliche Gatter sind: x0, x1, h0, h1, cnot");

      string gate = Console.ReadLine();
      string[] gates = gate.Split(',');

      foreach(var g in gates)
      {
        switch (g)
        {
          case ("x0"):
            Console.WriteLine("Initialisierung " +
              "Gatter Pauli-X Qubit 0");
            InitGateX0();
            break;
          case ("x1"):
            Console.WriteLine("Initialisierung " +
              "Gatter Pauli-X Qubit 1");
            InitGateX1();
            break;
          case ("h0"):
            Console.WriteLine("Initialisierung " +
              "Gatter Hadamard Qubit 0");
            InitGateH0();
            break;
          case ("h1"):
            Console.WriteLine("Initialisierung " +
              "Gatter Hadamard Qubit 1");
            InitGateH1();
            break;
          case ("cnot"):
            Console.WriteLine("Initialisierung " +
              "Gatter CNOT");
            InitGateCNOT();
            break;
```

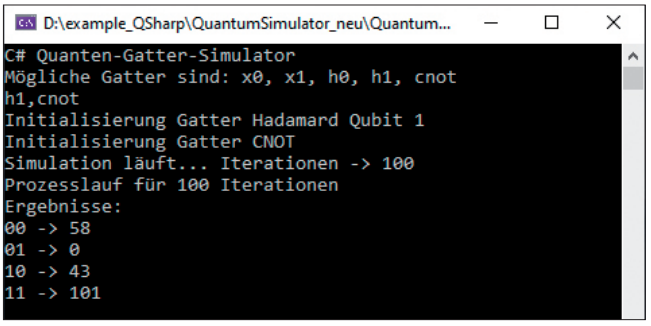
Listing 3: Implementierung des Simulators (Teil 2)

```
      default:
        Console.WriteLine(
          "Keine Auswahl getroffen!");
        break;
    }
  }

  Console.WriteLine("Simulation läuft... " +
    "Iterationen -> " + shots.ToString());
  double sq = r0 * r0 + i0 * i0 + r1 * r1 + i1 *
    i1 + r2 * r2 + i2 * i2 + r3 * r3 + i3 * i3;
  if (Math.Abs(sq - 1) > 0.00001)
    normalizationStateVector(sq);

  Console.WriteLine($"Prozesslauf für " *
    "{shots.ToString()} Iterationen");
  double z0 = 0; double z1 = 0; double z2 = 0;
    double z3 = 0;
  double p0 = (r0 * r0 + i0 * i0);
  double p1 = (r1 * r1 + i1 * i1) + p0;
  double p2 = (r2 * r2 + i2 * i2) + p1;
  double p3 = (r3 * r3 + i3 * i3) + p2;

  for (int i = 0; i <= shots; i++)
  {
    Random rnd = new Random();
    double r = rnd.NextDouble();
    if (r < p0) z0 = z0 + 1;
    if (r >= p0 && r < p1) z1 = z1 + 1;
    if (r >= p1 && r < p2) z2 = z2 + 1;
    if (r <= p2 && r < p3) z3 = z3 + 1;
  }

  Console.WriteLine("Ergebnisse:");
  Console.WriteLine($"00 -> {z0}");
  Console.WriteLine($"01 -> {z1}");
  Console.WriteLine($"10 -> {z2}");
  Console.WriteLine($"11 -> {z3}");

  Console.ReadLine();

}

private static void normalizationStateVector(
    double sq)
{
  double nf = Math.Sqrt(1/sq);
  r0 = r0 * nf;
  i0 = i0 * nf;
  r1 = r1 * nf;
  i1 = i1 * nf;
  r2 = r2 * nf;
  i2 = i2 * nf;
  r3 = r3 * nf;
```

Spiel. Deren Name stammt noch aus der klassischen Informatik von den wohlbekannten *AND*-, *OR*-, und *NOT*-Gattern. Im Unterschied zum klassischen booleschen Gatter haben Quantengatter jedoch einige Eigenschaften, die erfüllt sein müssen.

- Quantengatter können durch unitäre Matrizen [11] dargestellt werden.
- Kann die Operation eines Quantengatters nicht durch eine entsprechende unitäre Matrix ausgedrückt werden, so kann es nicht als gültiges Gatter betrachtet werden.
- Die Operationen von Quantengattern sind vollständig umkehrbar. Dies ist ein großer Unterschied zu booleschen Gattern.
- Bei einem *UND*-Gatter können Sie die Eingänge nicht bestimmen, wenn Sie die Ergebnisse betrachten. Sie können nur sagen: Wenn das Ergebnis 1 ist, dann sind beide Eingänge 1. Wenn das Ergebnis aber 0 ist, können Sie nicht bestimmen, ob die Eingänge (1,0), (0,1) oder (0,0) sind.
- Bei einem Quantengatter lässt sich der Eingang nur allein durch die Betrachtung des Ausgangs rekonstruieren.

Da Quantengatter nur Matrizen sind, können Sie diese wie aufgezeigt im Code leicht konstruieren, indem Sie in der Methode die entsprechende Matrix definieren. Die Anwendung eines Gatters ist dann einfach die Multiplikation des Gatters mit dem Qubit.



Verteilung der Quantenzustände (Bild 3)

Normalisierter Zustand

Die Methode *normalizationStateVector(double sq)* sorgt dafür, dass die Summe der quadrierten Amplituden gleich eins ist, wodurch sichergestellt wird, dass die mit jedem Zustand verbundenen Wahrscheinlichkeiten konsistent sind. Bild 3 zeigt die Verteilung der Quantenzustände nach 100 Iterationen bei einem Hadamard-Gatter auf Qubit 1, gefolgt von einer *CNOT*-Gatteroperation.

```
    i3 = i3 * nf;
  }

  private static void InitGateX0()
  {
    a0 = r0; r0 = r1; r1 = a0;
    a0 = i0; i0 = i1; i1 = a0;
    a0 = r2; r2 = r3; r3 = a0;
    a0 = i2; i2 = i3; i3 = a0;
  }

  private static void InitGateX1()
  {
    a0 = r1; r1 = r3; r3 = a0;
    a0 = i1; i1 = i3; i3 = a0;
    a0 = r0; r0 = r2; r2 = a0;
    a0 = i0; i0 = i2; i2 = a0;
  }

  private static void InitGateH0()
  {
    a0 = (r0 + r1) / Math.Sqrt(2);
      a1 = (i0 + i1) / Math.Sqrt(2);
    b0 = (r0 - r1) / Math.Sqrt(2);
      b1 = (i0 - i1) / Math.Sqrt(2);
    r0 = a0; i0 = a1; r1 = b0; i1 = b1;
    a0 = (r2 + r3) / Math.Sqrt(2);
      a1 = (i2 + i3) / Math.Sqrt(2);
    b0 = (r2 - r3) / Math.Sqrt(2);
      b1 = (i2 - i3) / Math.Sqrt(2);
    r2 = a0; i2 = a1; r3 = b0; i3 = b1;
  }

  private static void InitGateH1()
  {
    a0 = (r0 + r2) / Math.Sqrt(2);
      a1 = (i0 + i2) / Math.Sqrt(2);
    b0 = (r0 - r2) / Math.Sqrt(2);
      b1 = (i0 - i2) / Math.Sqrt(2);
    r0 = a0;  i0 = a1; r2 = b0; i2 = b1;
    a0 = (r1 + r3) / Math.Sqrt(2);
      a1 = (i1 + i3) / Math.Sqrt(2);
    b0 = (r1 - r3) / Math.Sqrt(2);
      b1 = (i1 - i3) / Math.Sqrt(2);
    r1 = a0; i1 = a1; r3 = b0; i3 = b1;
  }

  private static void InitGateCNOT()
  {
    a0 = r1; r1 = r3; r3 = a0;
    a0 = i1; i1 = i3; i3 = a0;
  }
 }
}
```

Fazit

Dieser kleine Workshop sollte zeigen, welche Möglichkeiten für die Programmierung von Quantensimulationen eine Rolle spielen können. Die Mehrzahl der verfügbaren Bibliotheken, als Beispiel sei Qiskit genannt, sind schwergewichtig und benötigen eine Menge Abhängigkeiten, um zu funktionieren.

Das C#-Beispiel will aufzeigen, dass es sich bei Simulationen im Kern um eine Reihe von mathematischen Transformationen handelt, die man nicht nur auf fortschrittlicher Quanten-Hardware simulieren kann. Für kleine Modelle kann sich deshalb die Erstellung eines eigenen Simulators lohnen.

Das Thema Quantensimulatoren und -programmiersprachen bleibt ein spannendes Feld für Forschung und Entwicklung. Bleiben Sie auf jeden Fall am Ball. ■

[1] Quantum Information Portal, https://www.quantiki.org
[2] Microsoft QDK Samples, www.dotnetpro.de/SL2502Quantensimulator1
[3] Intrinsische Funktion bei Wikipedia, www.dotnetpro.de/SL2502Quantensimulator2
[4] openHPI, Hadamard-Gatter, www.dotnetpro.de/SL2502Quantensimulator3
[5] Quanten-T-Gatter bei Wikipedia, www.dotnetpro.de/SL2502Quantensimulator4
[6] Daniel Basler, Q# für Einsteiger, dotnetpro 2/2025, Seite 38 ff., www.dotnetpro.de/A2502QSharp
[7] Mykola Dobrochynskyy, Wegweisende Symbiose, dotnetpro 2/2025, Seite 20 ff., www.dotnetpro.de/A2502QuantumCloud
[8] openHPI, Pauli-X-Gatter, www.dotnetpro.de/SL2502Quantensimulator5
[9] openHPI, CNOT-Gatter, www.dotnetpro.de/SL2502Quantensimulator6
[10] Microsoft Learn, Quantum circuit diagram conventions, www.dotnetpro.de/SL2502Quantensimulator7
[11] Unitäre Matrizen bei Wikipedia, www.dotnetpro.de/SL2502Quantensimulator8

Daniel Basler
arbeitet als Lead Developer und Softwarearchitekt. Seine Schwerpunkte liegen auf Cross-Plattform-Apps, Android, JavaScript und Microsoft-Technologien.

dnpCode A2502Quantensimulator