Discrete Adiabatic Quantum Linear-system Solver

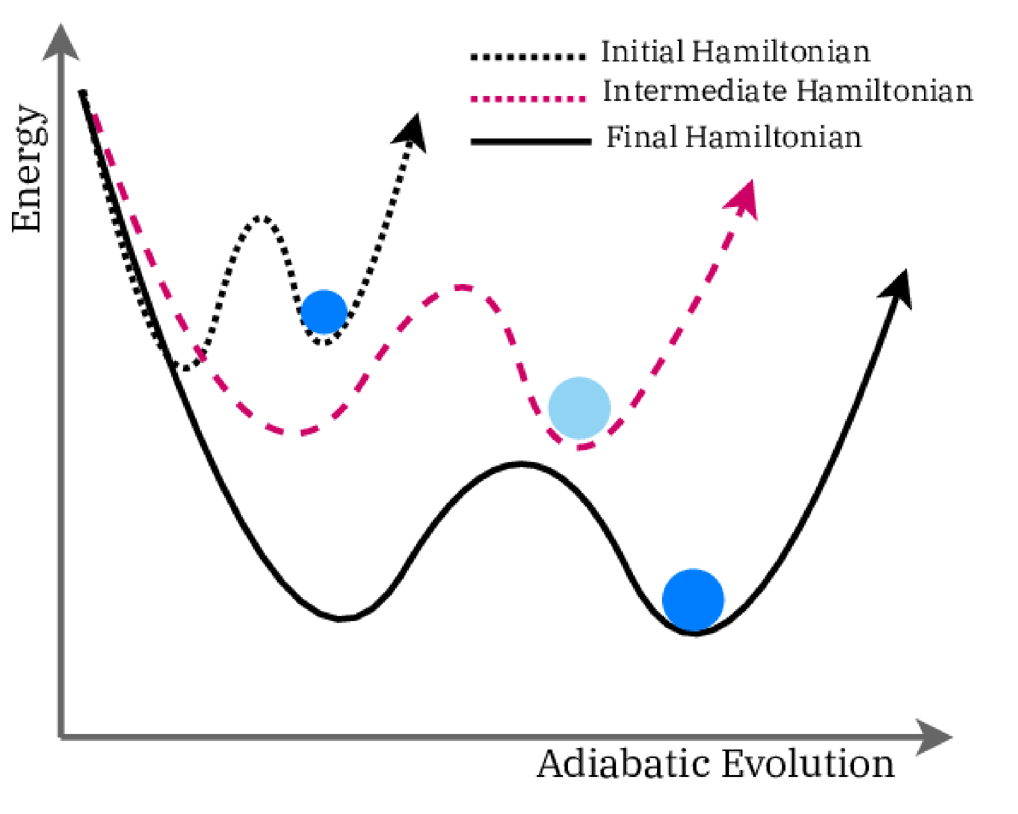
vhaktyr

2024/6/1

Index

- Linear-system Solver via Adiabatic Evolution
  - Preliminary
    - Time Evolution & Adiabatic Evolution
  - Adiabatic Linear Solver
    - Hamiltonian Simulation
    - Block Encoding
    - Schedule Function & Randomization Method
    - Eigenvalue Filtering
- Solutions to the Contest Problem
  - Block Encoding
  - Linear Solver
- Rethinkings

https://www.researchgate.net/figure/Graphical-representation-of-quantum-annealing_fig2_351810686



$$Ax = b \Rightarrow |\psi(t)\rangle = \prod e^{-iHt} |\psi(0)\rangle$$

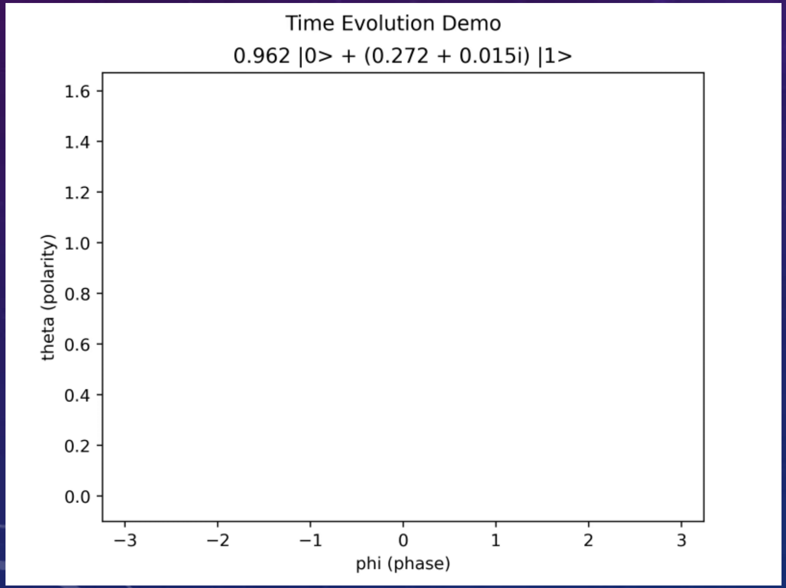
Linear-system Solver via Adiabatic Evolution

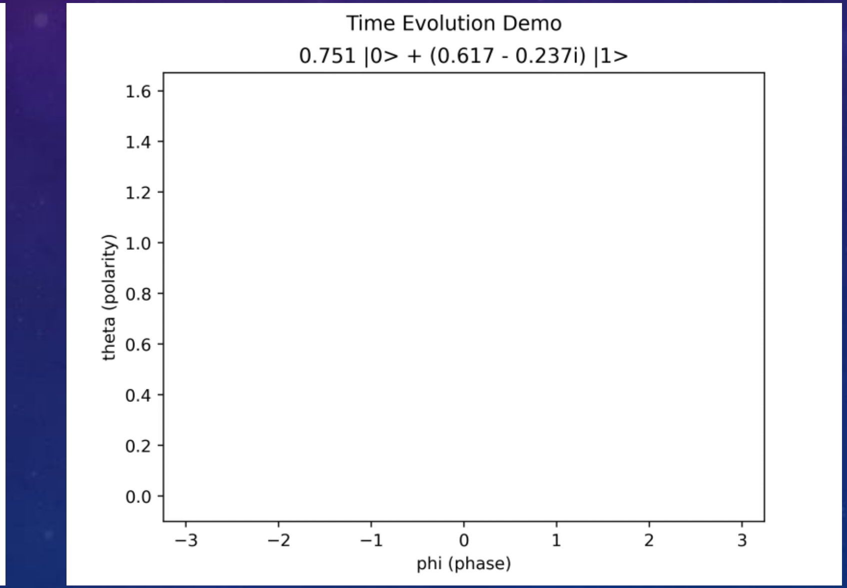
Part 1

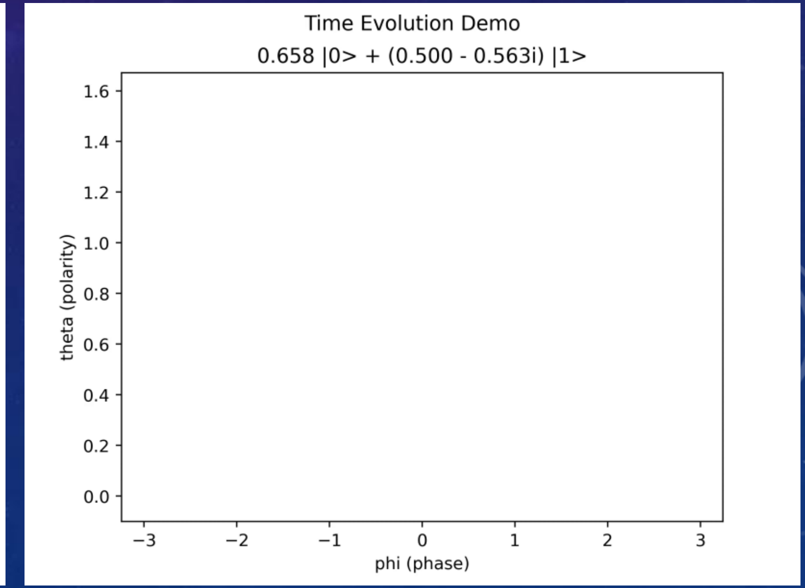
Time Evolution

- A system with time-independent Hamiltonian H and initial state $|\psi(0)\rangle$, the evolution is given by

$$|\psi(t)\rangle \ = e^{-iHt}|\psi(0)\rangle$$







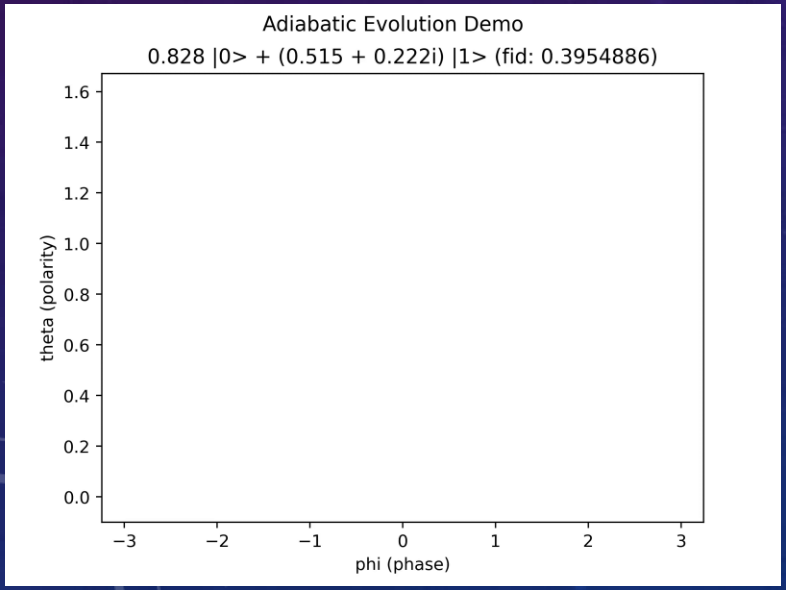
Eigen state

Non-eigen state

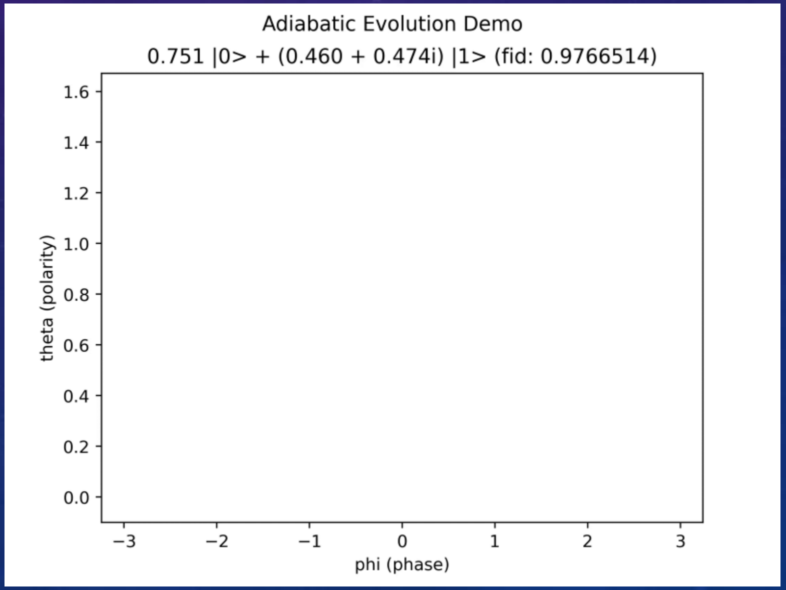
Adiabatic Evolution

- A system with time-dependent Hamiltonian *H(t)* and initial state $|\psi(0)\rangle$ of the *k*-th eigen state of *H(0)*, if the evolution is slow enough, the state $|\psi(t)\rangle$ will always stay at the *k*-th eigen state of *H(t)*. The discrete approximation can be given by Dyson series

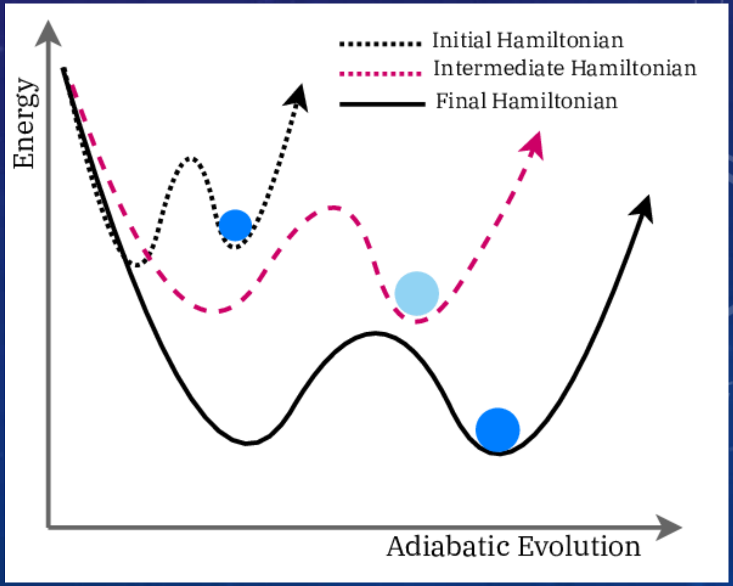
$$|\psi(t)\rangle = \prod_{s=0}^{S-1} e^{-iH_s\Delta t}\,|\psi(0)\rangle$$



Fast



Slow



General Procedure of Adiabatic Linear Solver

- Map Linear System $Ax = b$ to Adiabatic Evolution $|\psi(t)\rangle = \prod e^{-iHt} |\psi(0)\rangle$
  - Pre-transformation
    - A is positive definite, with spectral norm $||A||_2 \leq 1$
    - b is normalized to unit vector
  - Prepare initial state $|\psi(0)\rangle = |b\rangle$
  - Design H(s) w.r.t A
    - $|b\rangle$ and $|x\rangle$ are the nullspace vectors of H_0 and H_1
    - H(s) is a scheduled mixture of H_0 and H_1
  - Perform evolutions $\prod e^{-iHt}$
  - Get final state $|\psi(t)\rangle = |x\rangle$
- Practically, need some ancilla qubits to complete

For example $A = \begin{bmatrix}2 & 1\\ 1 & 0\end{bmatrix}, b = \begin{bmatrix}3\\ 1\end{bmatrix}$

Let $\widetilde{A} = \frac{A^{\dagger}A}{|A^{\dagger}b|}, \tilde{b} = \frac{A^{\dagger}b}{|A^{\dagger}b|}$ to satisfy conditions

Follow arXiv:1805.10549, define:

$$f(s)\colon\ [0,1] \to [0,1]$$

$$A(s) = (1 - f(s)) \cdot I + f(s) \cdot \widetilde{A}$$

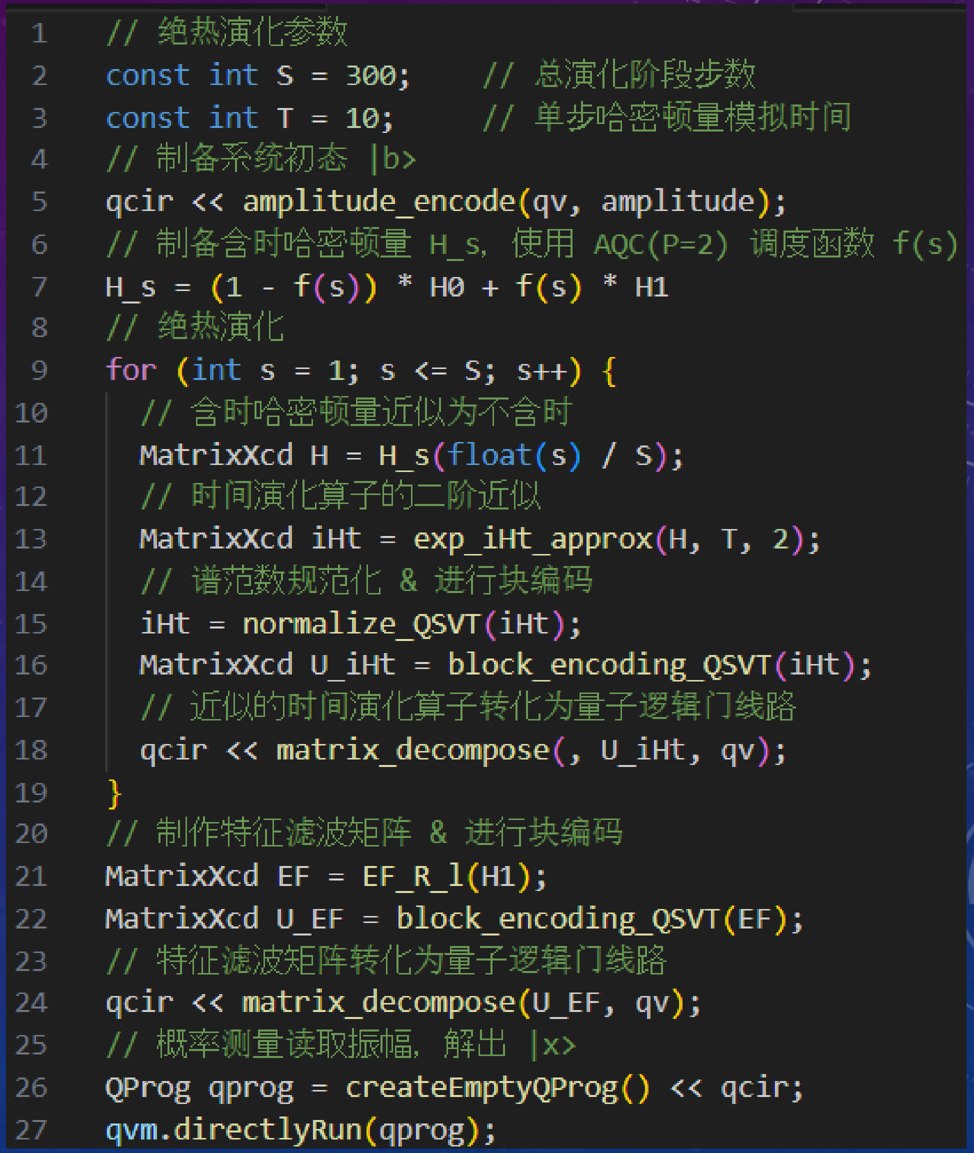
$$H(s) = A(s) \cdot (I - |\tilde{b}\rangle\langle\tilde{b}|) \cdot A(s)$$

st. $H(0)|b\rangle = H(1)|x\rangle = 0$

With $H(s)$ where s: 0 → 1 we'll also have $|b\rangle \to |x\rangle$

Implementation Techniques

- Time evolution operator
    - Hamiltonian Simulation
    - Block Encoding
- Adiabatic evolution control
    - Logical step: schedule function $f(s)$
    - Physical time: randomization method
- Quest for more precision
    - Eigenvalue Filtering

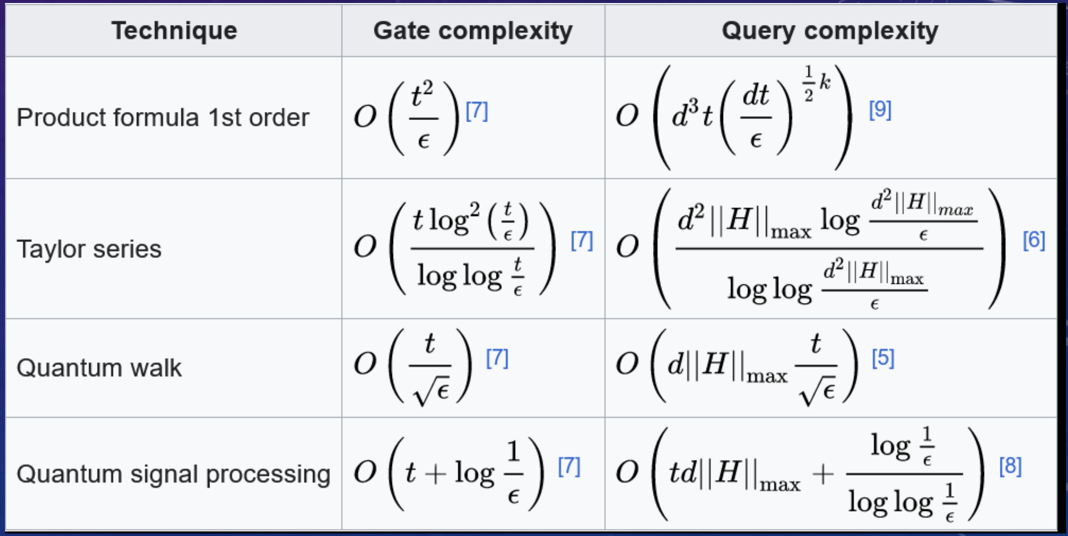


```
// 绝热演化参数
const int S = 300;     // 总演化阶段步数
const int T = 10;      // 单步哈密顿量模拟时间
// 制备系统初态 |b>
qcir << amplitude_encode(qv, amplitude);
// 制备含时哈密顿量 H_s, 使用 AQC(P=2) 调度函数 f(s)
H_s = (1 - f(s)) * H0 + f(s) * H1
// 绝热演化
for (int s = 1; s <= S; s++) {
  // 含时哈密顿量近似为不含时
  MatrixXcd H = H_s(float(s) / S);
  // 时间演化算子的二阶近似
  MatrixXcd iHt = exp_iHt_approx(H, T, 2);
  // 谱范数规范化 & 进行块编码
  iHt = normalize_QSVT(iHt);
  MatrixXcd U_iHt = block_encoding_QSVT(iHt);
  // 近似的时间演化算子转化为量子逻辑门线路
  qcir << matrix_decompose(, U_iHt, qv);
}
// 制作特征滤波矩阵 & 进行块编码
MatrixXcd EF = EF_R_l(H1);
MatrixXcd U_EF = block_encoding_QSVT(EF);
// 特征滤波矩阵转化为量子逻辑门线路
qcir << matrix_decompose(U_EF, qv);
// 概率测量读取振幅, 解出 |x>
QProg qprog = createEmptyQProg() << qcir;
qvm.directlyRun(qprog);
```

pseudo-code for our LS impl.

Hamiltonian Simulation

- Implement time evolution operator e^{-iHt} in quantum circuit
- Techniques
  - Trotterization
    - $e^{-i(A+B)t} = \lim_{r\to\infty} (e^{-i\frac{A}{r}t} e^{-i\frac{B}{r}t})^{\mathrm{r}}$
  - Taylor approx.
    - $e^{-iHt} = \sum_k \frac{(-iHt)^k}{k!} \approx I - iHt$, then block encode
  - Quantum Walk
  - Quantum Signal Processing
    - So-far the optimal way ⭐



Technique	Gate complexity	Query complexity
Product formula 1st order	$O\left(\frac{t^2}{\epsilon}\right)$ [7]	$O\left(d^3 t \left(\frac{dt}{\epsilon}\right)^{\frac{1}{2}k}\right)$ [9]
Taylor series	$O\left(\frac{t \log^2\left(\frac{t}{\epsilon}\right)}{\log\log\frac{t}{\epsilon}}\right)$ [7]	$O\left(\frac{d^2 \lVert H\rVert_{\max} \log \frac{d^2 \lVert H\rVert_{max}}{\epsilon}}{\log\log \frac{d^2 \lVert H\rVert_{\max}}{\epsilon}}\right)$ [6]
Quantum walk	$O\left(\frac{t}{\sqrt{\epsilon}}\right)$ [7]	$O\left(d \lVert H\rVert_{\max} \frac{t}{\sqrt{\epsilon}}\right)$ [5]
Quantum signal processing	$O\left(t + \log\frac{1}{\epsilon}\right)$ [7]	$O\left(td \lVert H\rVert_{\max} + \frac{\log\frac{1}{\epsilon}}{\log\log\frac{1}{\epsilon}}\right)$ [8]

https://en.wikipedia.org/wiki/Hamiltonian_simulation

Block Encoding (arXiv:2402.17529)

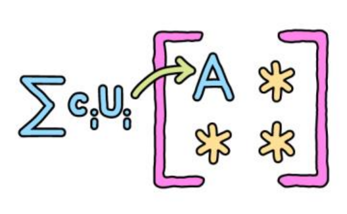
- Introducing non-unitary to quantum computing
- Finds a unitary U_A that expands an arbitrary given matrix A by

$$U_\mathrm{A} = \begin{bmatrix} \mathrm{A} & * \\ * & * \end{bmatrix}$$

so that general linear algebra like $\mathrm{A} \cdot \vec{b}$ could be easily implemented on gate-based quantum computers like

$$U_\mathrm{A}|b\rangle = \begin{bmatrix} \mathrm{A} & * \\ * & * \end{bmatrix} \begin{bmatrix} b \\ 0 \end{bmatrix} = \begin{bmatrix} \mathrm{A}b \\ * \end{bmatrix}$$

- Techniques (see part 2)
  - direct construction: any matrix
  - prepare-select based: LCU matrix
  - query-oracle based: d-sparse matrix
  - BE upon another BE…



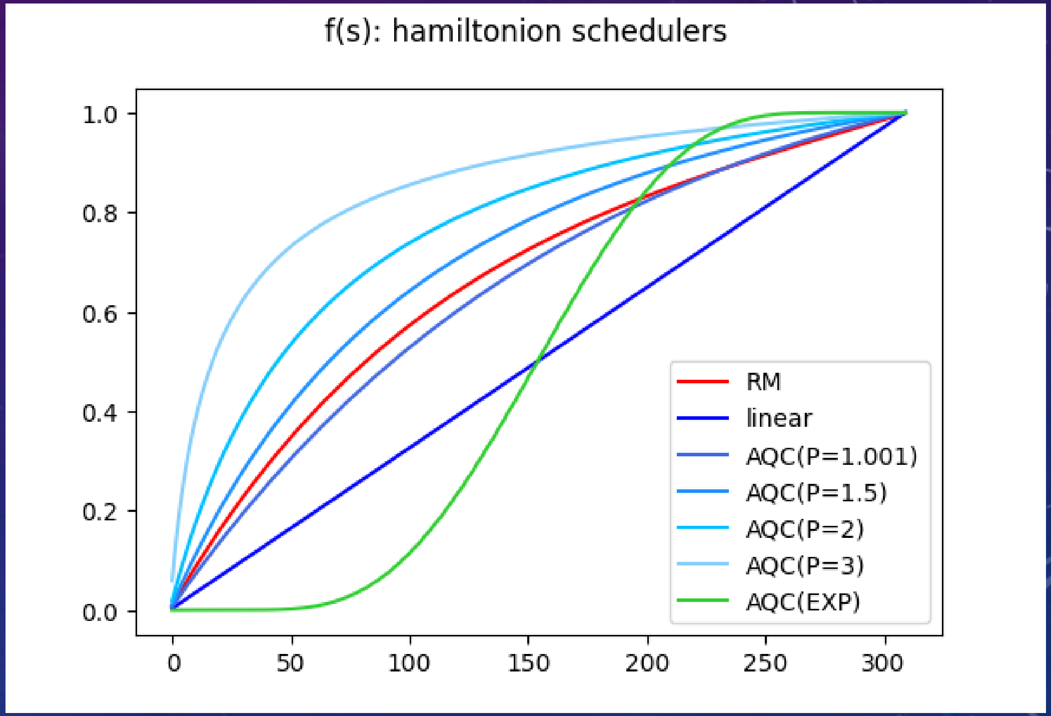
https://pennylane.ai/qml/demos/tutorial_lcu_blockencoding
https://pennylane.ai/qml/demos/tutorial_block_encoding

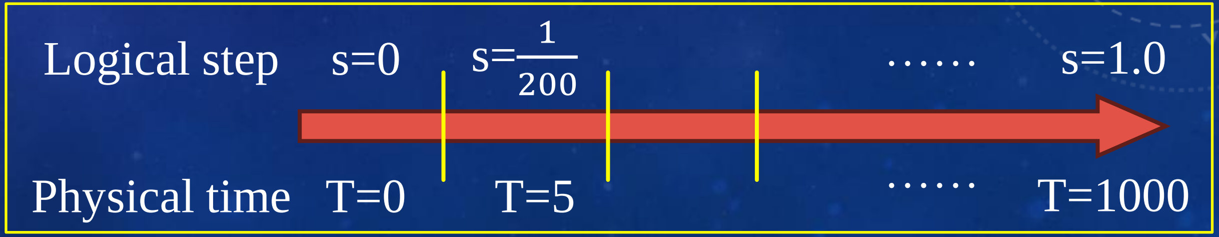
Schedule Function & Randomizatiom Method

(arXiv:1909.05500) (arXiv:1805.10549)

- Hamiltonian schedule: s in H(s)
  - $f(s)$: $[0,1] \rightarrow [0,1]$
  - $H(s) = (1 - f(s)) \cdot H_0 + f(s) \cdot H_1$
  - Principle: $f(s) \uparrow,\ f'(s) \downarrow$

- Evolution time schedule: t in e^{-iHt}
  - constant: t $=$ T / S
  - randomized: t $\sim U[0, 2\pi\Delta^*]$
    - $\Delta^*(\mathrm{s}) = 1 - f(s) + f(s)/\kappa$
    - κ is the condition number of A





Eigenvalue Filtering (arXiv:1910.14596)

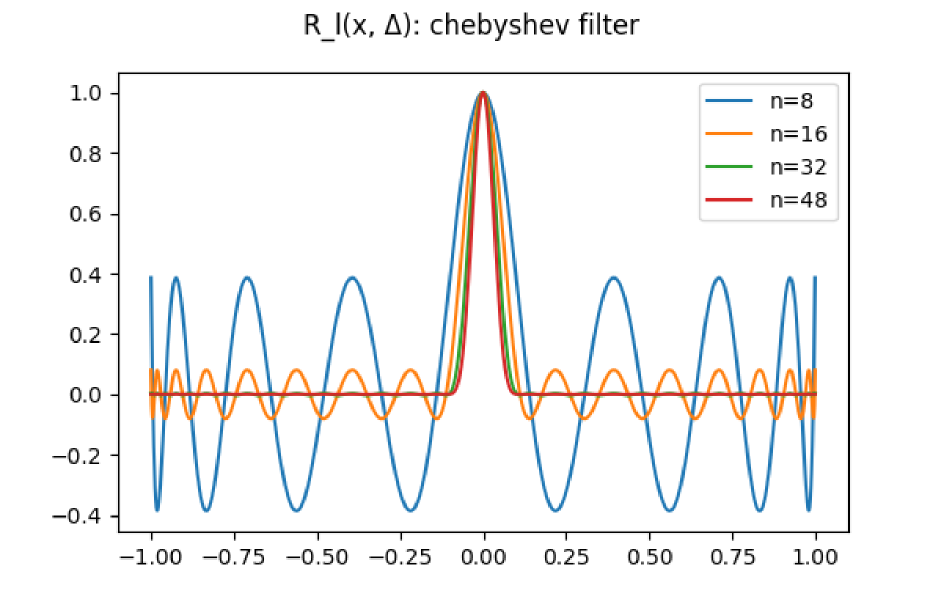
- Projection operator P_λ preserving λ-eigenstate, filtering out others eigenstates
- The ℓ-th Chebyshev polynomial of the first kind

$$T_\ell(x) = \begin{cases} \cos(\ell \arccos x)\,, |x| \leq 1 \\ \cosh(\ell \operatorname{arccosh} x), x > 1 \\ (-1)^\ell \cos h(\ell \operatorname{arccosh} -x)\,, x < -1 \end{cases}$$

- Define the 2ℓ-th degree polynomial

$$R_\ell(x;\Delta) = \frac{T_\ell(-1 + 2\frac{x^2 - \Delta^2}{1 - \Delta^2})}{T_\ell(-1 + 2\frac{-\Delta^2}{1 - \Delta^2})}$$

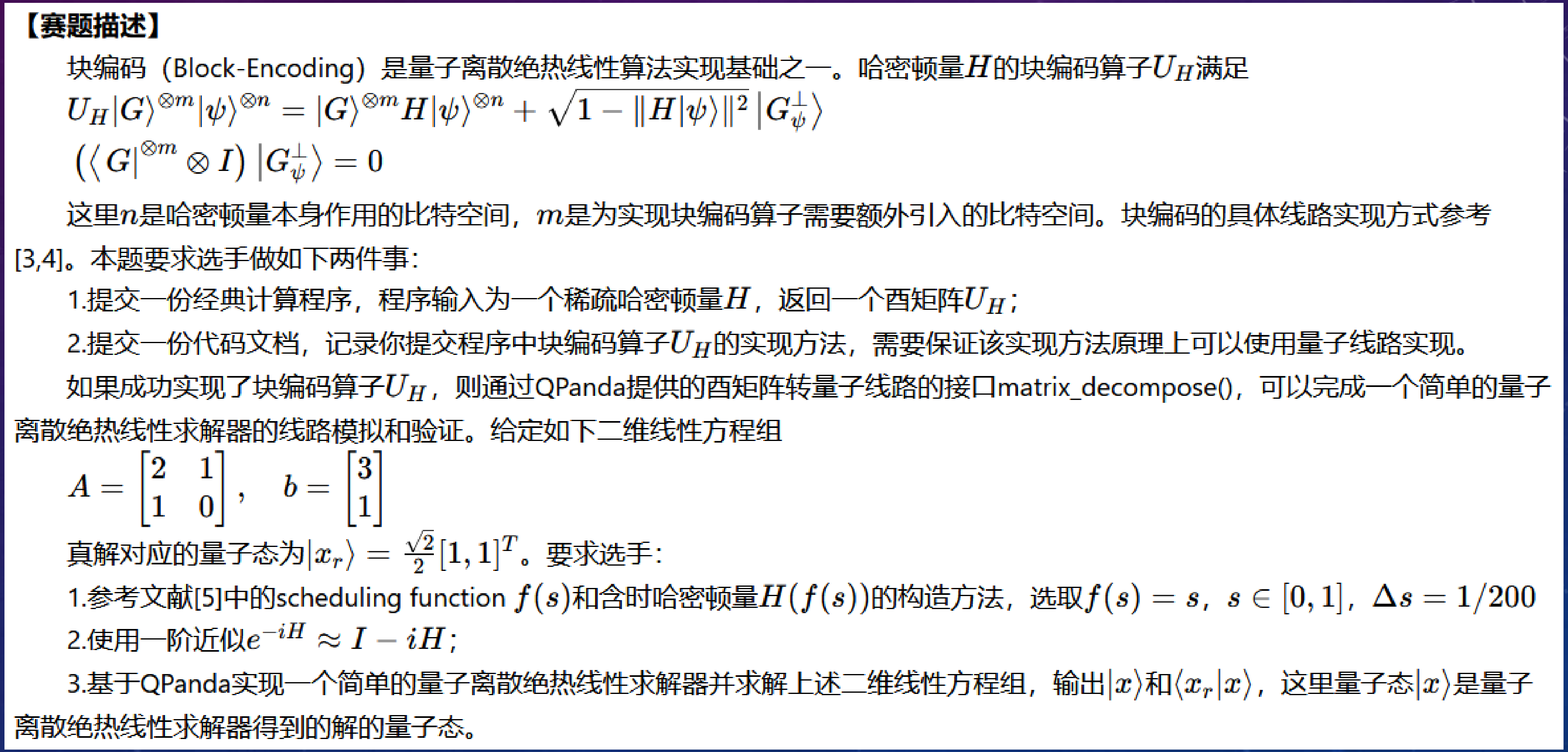
- Perform QSVT over H$_1$: $R_\ell(H;\Delta) = \mathrm{V} \cdot diag(R_\ell(\lambda_i;\Delta)) \cdot \mathrm{V}^{-1}$



Solutions to the Contest Problem

Part 2

Problem Description



【赛题描述】

块编码（Block-Encoding）是量子离散绝热线性算法实现基础之一。哈密顿量H的块编码算子U_H满足

$$U_H|G\rangle^{\otimes m}|\psi\rangle^{\otimes n} = |G\rangle^{\otimes m}H|\psi\rangle^{\otimes n} + \sqrt{1-\|H|\psi\rangle\|^2}\,|G_\psi^\perp\rangle$$

$$\left(\langle G|^{\otimes m}\otimes I\right)|G_\psi^\perp\rangle = 0$$

这里n是哈密顿量本身作用的比特空间，m是为实现块编码算子需要额外引入的比特空间。块编码的具体线路实现方式参考[3,4]。本题要求选手做如下两件事：

1.提交一份经典计算程序，程序输入为一个稀疏哈密顿量H，返回一个酉矩阵U_H；

2.提交一份代码文档，记录你提交程序中块编码算子U_H的实现方法，需要保证该实现方法原理上可以使用量子线路实现。

如果成功实现了块编码算子U_H，则通过QPanda提供的酉矩阵转量子线路的接口matrix_decompose()，可以完成一个简单的量子离散绝热线性求解器的线路模拟和验证。给定如下二维线性方程组

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}, \quad b = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

真解对应的量子态为$|x_r\rangle = \frac{\sqrt{2}}{2}[1,1]^T$。要求选手：

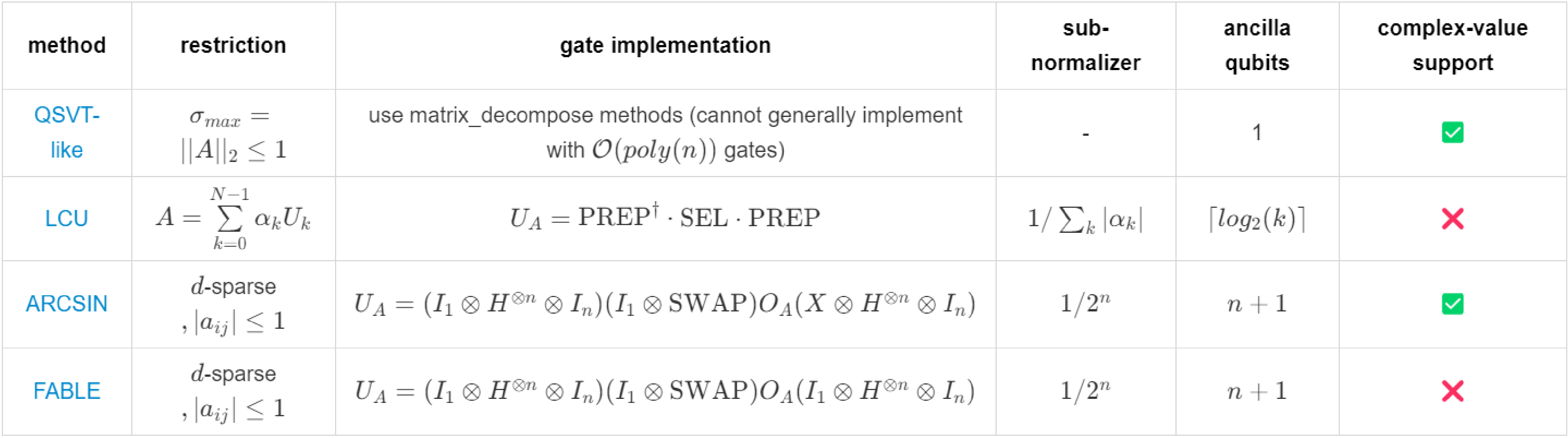
1.参考文献[5]中的scheduling function $f(s)$和含时哈密顿量$H(f(s))$的构造方法，选取$f(s)=s$，$s\in[0,1]$，$\Delta s = 1/200$

2.使用一阶近似$e^{-iH}\approx I - iH$；

3.基于QPanda实现一个简单的量子离散绝热线性求解器并求解上述二维线性方程组，输出$|x\rangle$和$\langle x_r|x\rangle$，这里量子态$|x\rangle$是量子离散绝热线性求解器得到的解的量子态。

Task 1: Implementing the Block Encoding

- 我们实现了 4 种不同的块编码算法
  - 以 QSVT 方法为第一题的最终提交



method	restriction	gate implementation	sub-normalizer	ancilla qubits	complex-value support
QSVT-like	$\sigma_{max} = \|\|A\|\|_2 \leq 1$	use matrix_decompose methods (cannot generally implement with $\mathcal{O}(poly(n))$ gates)	-	1	✅
LCU	$A = \sum_{k=0}^{N-1} \alpha_k U_k$	$U_A = \mathrm{PREP}^{\dagger} \cdot \mathrm{SEL} \cdot \mathrm{PREP}$	$1/\sum_k \|\alpha_k\|$	$\lceil log_2(k) \rceil$	❌
ARCSIN	d-sparse, $\|a_{ij}\| \leq 1$	$U_A = (I_1 \otimes H^{\otimes n} \otimes I_n)(I_1 \otimes \mathrm{SWAP})O_A(X \otimes H^{\otimes n} \otimes I_n)$	$1/2^n$	$n+1$	✅
FABLE	d-sparse, $\|a_{ij}\| \leq 1$	$U_A = (I_1 \otimes H^{\otimes n} \otimes I_n)(I_1 \otimes \mathrm{SWAP})O_A(I_1 \otimes H^{\otimes n} \otimes I_n)$	$1/2^n$	$n+1$	❌

Block Encoding: QSVT-like

(arXiv:2203.10236)

- 对任意形状的复矩阵A，满足谱范数 $||A||_2 \leq 1$，直接构造如下矩阵

$$U_A = \begin{bmatrix} A & \sqrt{I - AA^\dagger} \\ \sqrt{I - A^\dagger A} & -A^\dagger \end{bmatrix}$$

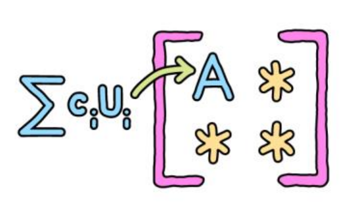
- 其中 $\sqrt{\cdot}$ 定义为矩阵开根运算，即在 A 的特征分解中对谱系数分别开根

$$\sqrt{A} = V\sqrt{D}V^{-1} = V \cdot diag(\sqrt{\lambda_i}) \cdot V^{-1}$$

- 易验证 U_A 为A的一个 (λ=1,m=1,ε=0)-块编码
  - U_A 满足酉性 $U_A{}^\dagger U_A = I$，且A出现在其左上角的子空间
  - 缩放因子 λ=1；辅助比特数 m=1；误差 ε=0
- 当A不满足谱范数条件时，可以转而考虑对规范化后的 $\tilde{A} = \frac{A}{||A||_2}$ 作块编码
- U_A 的逻辑门线路可以借助 QR/CSD/QSD 等矩阵分解技术实现

Block Encoding: LCU

(arXiv:1202.5822)



$$\mathrm{PREP}\,|0\rangle = \sum_{i=0}^{M-1} \sqrt{\frac{\alpha_i}{s}}\,|i\rangle$$

$$\mathrm{SELECT} = \sum_{i=0}^{M-1} |i\rangle\langle i| \otimes U_i$$

$$\mathrm{PREP}^{\dagger} \cdot \mathrm{SELECT} \cdot \mathrm{PREP} = \sum_{i=0}^{M-1} \frac{\alpha_i}{s} U_i = \frac{1}{s} A$$

- 若复方阵A可以分解为若干酉阵的线性和，即存在 A = $\sum_{i=0}^{M-1} \alpha_i U_i$
- 则可以通过下述 PREPARE-SELECT 结构的线路对A进行块编码
  - PREP: 振幅编码提供各项对应的系数 α_i
  - SEL: 控制比特按位序和门控条件来选择各项对应的酉矩阵 U_i

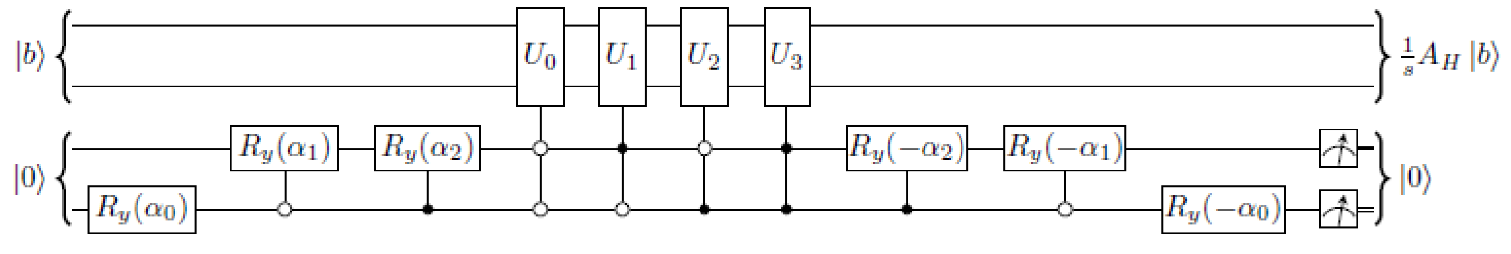
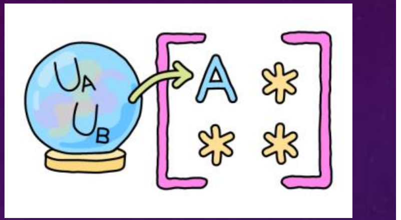


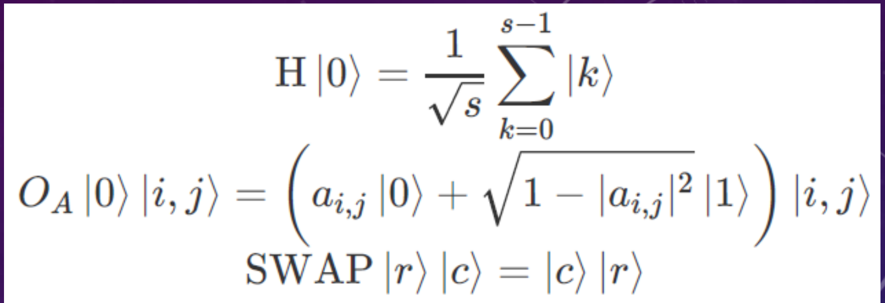
Figure 7: Circuit for encoding a 4x4 matrix using PREPARE-SELECT.

- 易验证该线路所对应的酉矩阵 U_A 为A的一个 (λ= 1/$\sum_{i=0}^{M-1} |\alpha_i|$,m=$\lceil log_2(M) \rceil$,ε)-块编码
  - 缩放因子 λ=1/s 来源于振幅编码时的概率归一化
  - 引入的 m 个辅助比特用于 PREP 线路
  - 误差 ε 取决于舍去小项的系数

Block Encoding: ARCCOS

(arXiv:2402.17529)





$$\mathrm{H}\,|0\rangle = \frac{1}{\sqrt{s}}\sum_{k=0}^{s-1}|k\rangle$$

$$O_A\,|0\rangle\,|i,j\rangle = \left(a_{i,j}\,|0\rangle + \sqrt{1-|a_{i,j}|^2}\,|1\rangle\right)|i,j\rangle$$

$$\mathrm{SWAP}\,|r\rangle\,|c\rangle = |c\rangle\,|r\rangle$$

- 若复方阵A为 d-稀疏矩阵， 则可以通过下述 query-oracle 结构的线路对A进行块编码

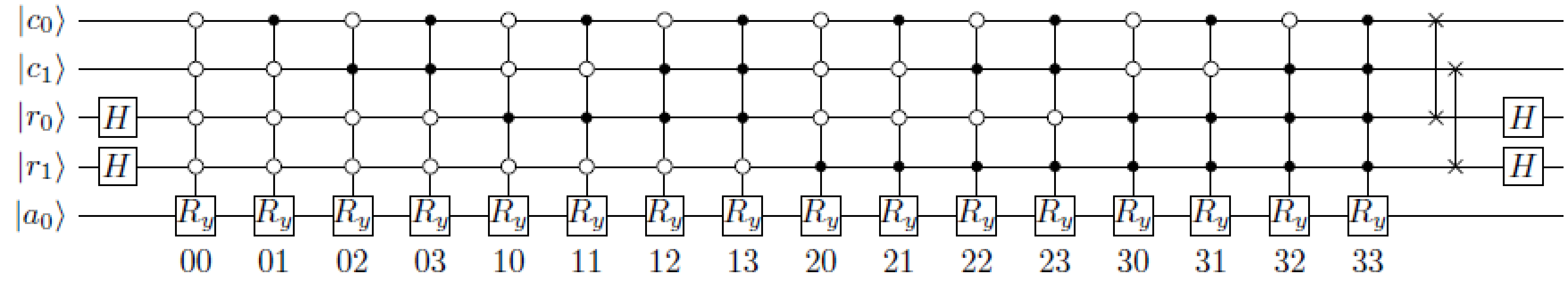
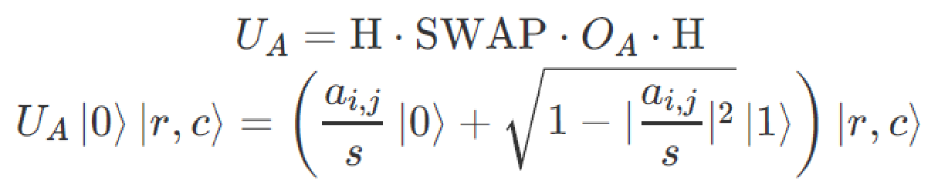


Figure 3: Block encoding circuit for a 4x4 matrix. Labels below the circuit indicate the row and column indices for the controlled rotations.

- 易验证该线路所对应的酉矩阵 U_A 为A的一个 (λ=$\frac{1}{2^N}$,m=N,ε)-块编码
  - N为原矩阵A的行/列数
  - 缩放因子 λ=1/s 来源于哈达玛打散时的振幅均一化
  - 引入的 m 个辅助比特用于行列索引

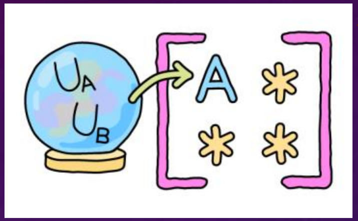


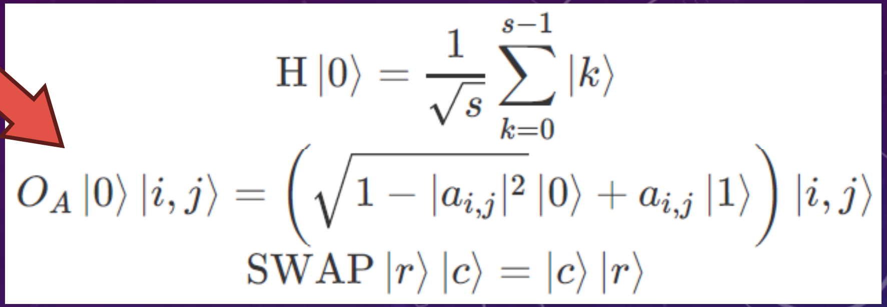
$$U_A = \mathrm{H}\cdot\mathrm{SWAP}\cdot O_A\cdot\mathrm{H}$$

$$U_A\,|0\rangle\,|r,c\rangle = \left(\frac{a_{i,j}}{s}\,|0\rangle + \sqrt{1-|\frac{a_{i,j}}{s}|^2}\,|1\rangle\right)|r,c\rangle$$

Block Encoding: ARCSIN

(arXiv:2402.17529)





$$\mathrm{H}\,|0\rangle = \frac{1}{\sqrt{s}}\sum_{k=0}^{s-1}|k\rangle$$

$$O_A\,|0\rangle\,|i,j\rangle = \left(\sqrt{1-|a_{i,j}|^2}\,|0\rangle + a_{i,j}\,|1\rangle\right)|i,j\rangle$$

$$\mathrm{SWAP}\,|r\rangle\,|c\rangle = |c\rangle\,|r\rangle$$

- 考虑到稀疏矩阵中存在大量0，将 arccos 改为 arcsin 编码以消除旋转角度为 arccos(π/2) 的项
    - 辅助比特线路末尾引入X门

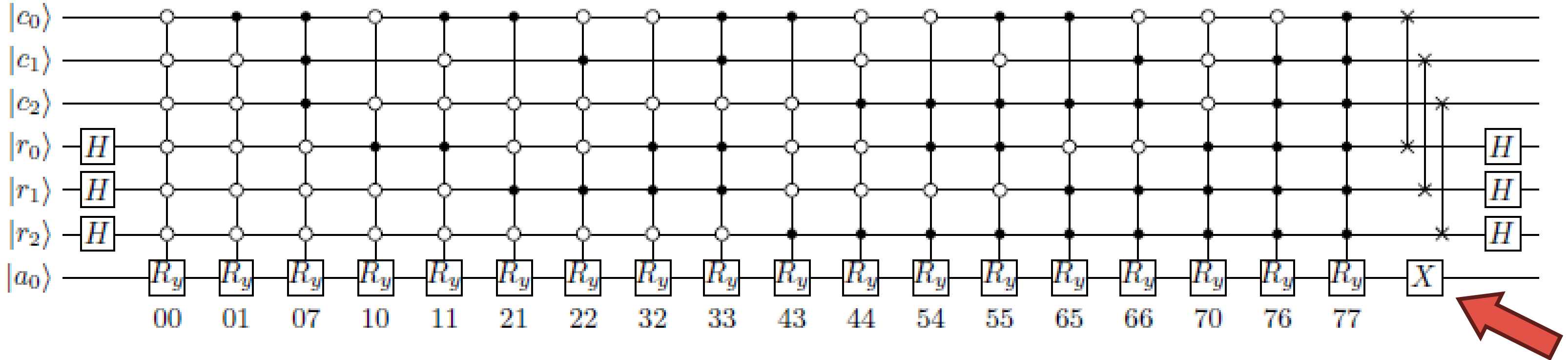
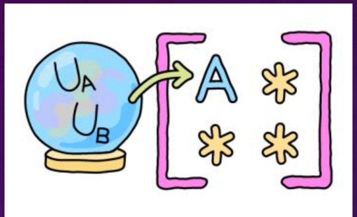


Figure 5: ARCSIN encoding circuit for 8x8 tri-diagonal matrix with coalesence of equal valued off-diagonal rotations on rows 1 to 6.

- 块编码参数 (λ= $\frac{1}{2^N}$,m=N,ε) 与 ARCCOS 一致

Block Encoding: FABLE

(arXiv:2402.17529)



- 同样基于 query-oracle 框架，适用于d-稀疏矩阵，通过一些技术进行了线路深度压缩
    - 对易的多比特控制门进行对消
    - 格雷码编码：连续的旋转门进行角度合并

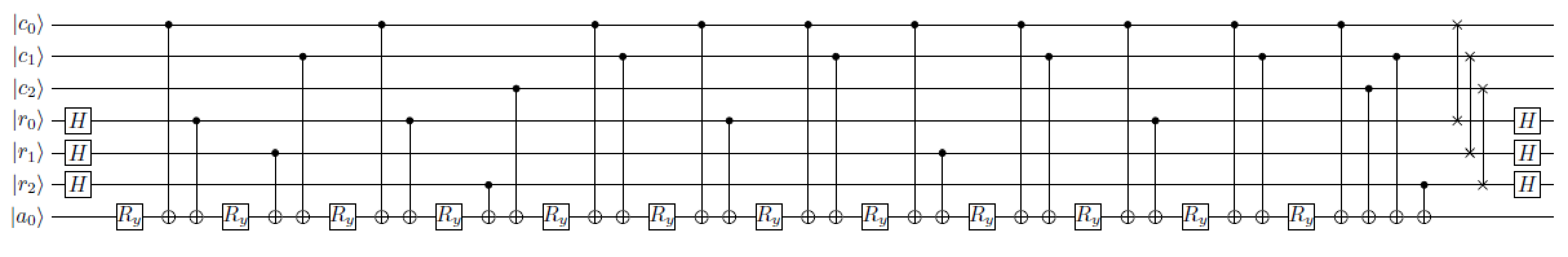
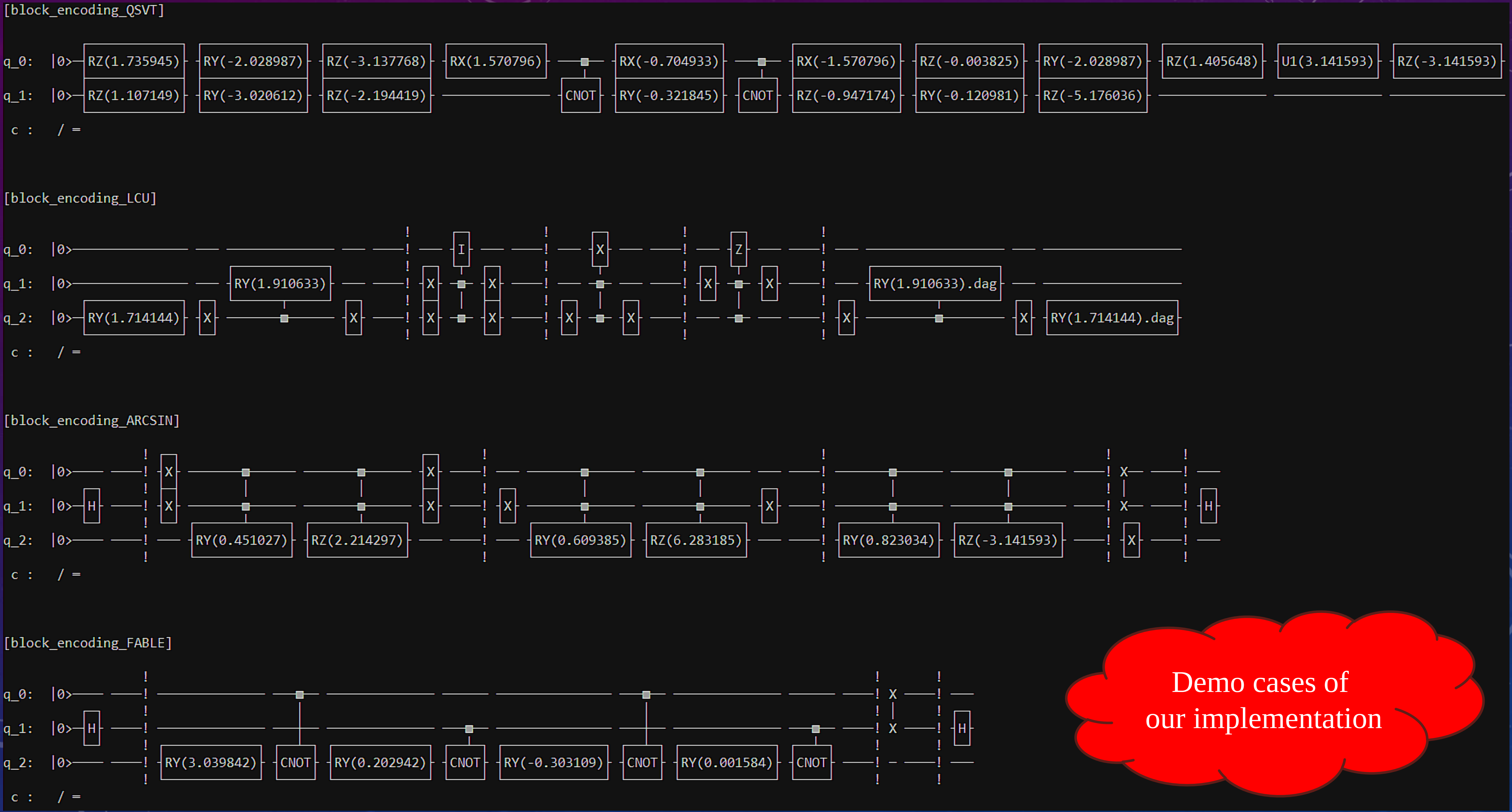


Figure 6: FABLE encoding circuit for 8x8 tri-diagonal matrix.

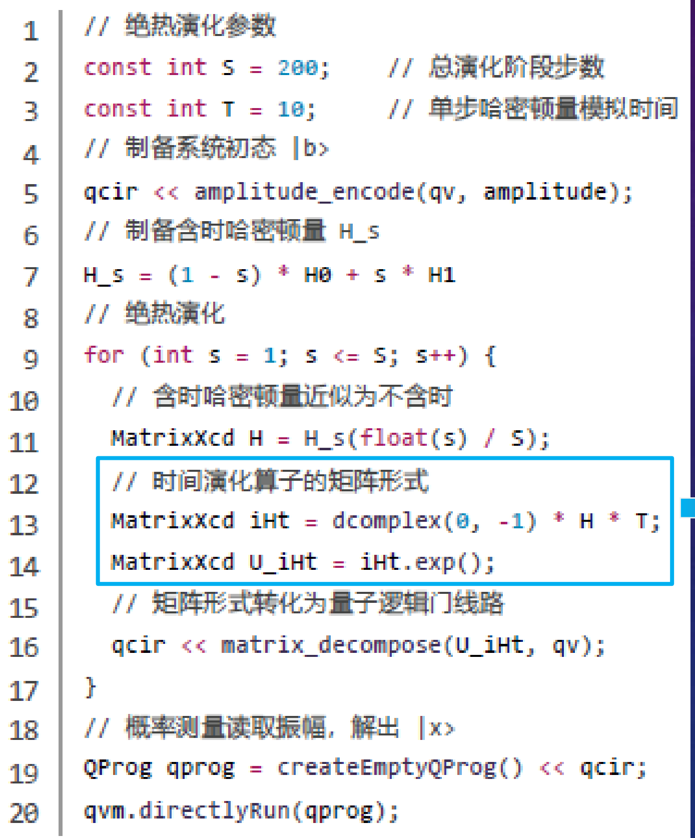
- 块编码参数 ($\lambda=\frac{1}{2^N}$,m=N,ε) 与 ARCCOS 一致



```
[block_encoding_QSVT]

q_0:  |0>─┤RZ(1.735945)├─┤RY(-2.028987)├─┤RZ(-3.137768)├─┤RX(1.570796)├─ ───■──── ─┤RX(-0.704933)├─ ───■──── ─┤RX(-1.570796)├─┤RZ(-0.003825)├─┤RY(-2.028987)├─┤RZ(1.405648)├─┤U1(3.141593)├─┤RZ(-3.141593)├
q_1:  |0>─┤RZ(1.107149)├─┤RY(-3.020612)├─┤RZ(-2.194419)├─ ────────────── ─┤CNOT├─ ─┤RY(-0.321845)├─ ─┤CNOT├─ ─┤RZ(-0.947174)├─┤RY(-0.120981)├─┤RZ(-5.176036)├─ ────────────── ────────────── ──────────────
 c :   / =


[block_encoding_LCU]

q_0:  |0>─────────────── ─── ─────────────── ─── ───! ─── ┤I├ ─── ───! ─── ┤X├ ─── ───! ─── ┤Z├ ─── ───! ─── ──────────────────── ─── ───────────────────
q_1:  |0>─────────────── ─── ┤RY(1.910633)├ ─── ───! ┤X├ ─■─ ┤X├ ───! ─── ─■─ ─── ───! ┤X├ ─■─ ┤X├ ───! ─── ┤RY(1.910633).dag├ ─── ───────────────────
q_2:  |0>┤RY(1.714144)├ ┤X├ ──────■──────── ┤X├ ───! ┤X├ ─■─ ┤X├ ───! ┤X├ ─■─ ┤X├ ───! ─── ─■─ ─── ───! ┤X├ ─────────■────────── ┤X├ ┤RY(1.714144).dag├
 c :   / =


[block_encoding_ARCSIN]

q_0:  |0>─── ───! ┤X├ ───────■────── ───────■────── ┤X├ ───! ─── ───────■────── ───────■────── ─── ───! ───────■────── ───────■─────── ───! X─ ───! ───
q_1:  |0>┤H├ ───! ┤X├ ───────■────── ───────■────── ┤X├ ───! ┤X├ ───────■────── ───────■────── ┤X├ ───! ───────■────── ───────■─────── ───! X─ ───! ┤H├
q_2:  |0>─── ───! ─── ┤RY(0.451027)├ ┤RZ(2.214297)├ ─── ───! ─── ┤RY(0.609385)├ ┤RZ(6.283185)├ ─── ───! ┤RY(0.823034)├ ┤RZ(-3.141593)├ ───! ┤X├ ───! ───
 c :   / =


[block_encoding_FABLE]

q_0:  |0>─── ───! ─────────────── ──■─── ─────────────── ────── ──────────────── ──■─── ─────────────── ────── ───! X ───! ───
q_1:  |0>┤H├ ───! ─────────────── ────── ─────────────── ──■─── ──────────────── ────── ─────────────── ──■─── ───! X ───! ┤H├
q_2:  |0>─── ───! ┤RY(3.039842)├ ┤CNOT├ ┤RY(0.202942)├ ┤CNOT├ ┤RY(-0.303109)├ ┤CNOT├ ┤RY(0.001584)├ ┤CNOT├ ───! ─ ───! ───
 c :   / =
```

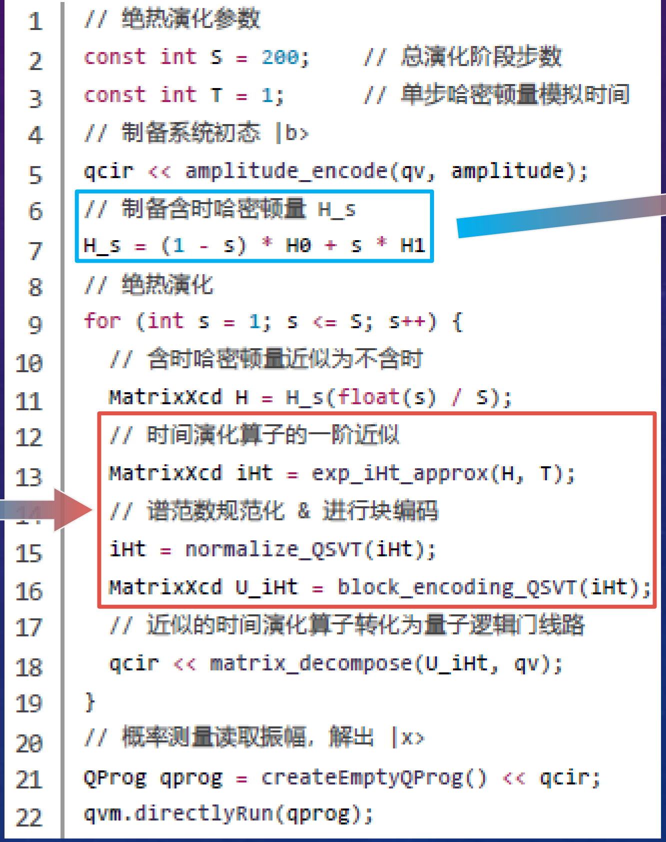
Demo cases of
our implementation

Task 2: Implementing the Linear Solver



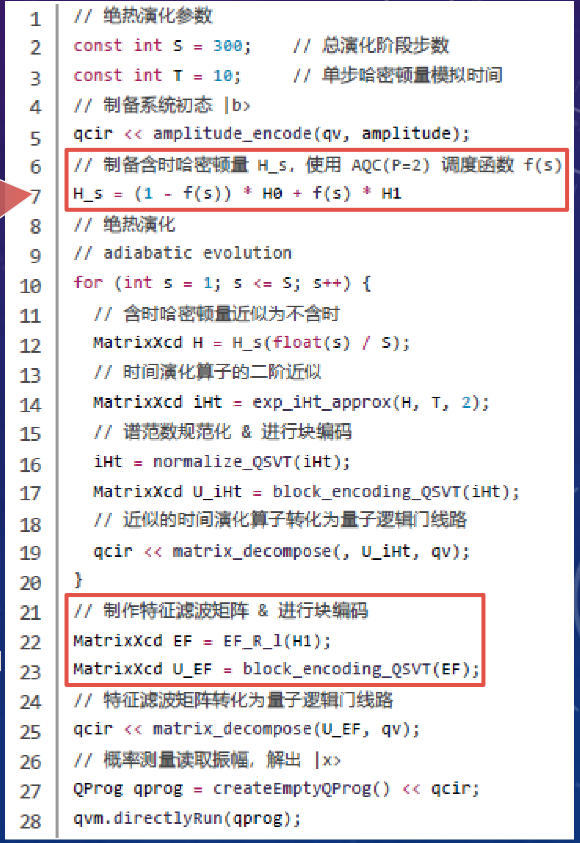
```
// 绝热演化参数
const int S = 200;     // 总演化阶段步数
const int T = 10;      // 单步哈密顿量模拟时间
// 制备系统初态 |b>
qcir << amplitude_encode(qv, amplitude);
// 制备含时哈密顿量 H_s
H_s = (1 - s) * H0 + s * H1
// 绝热演化
for (int s = 1; s <= S; s++) {
  // 含时哈密顿量近似为不含时
  MatrixXcd H = H_s(float(s) / S);
  // 时间演化算子的矩阵形式
  MatrixXcd iHt = dcomplex(0, -1) * H * T;
  MatrixXcd U_iHt = iHt.exp();
  // 矩阵形式转化为量子逻辑门线路
  qcir << matrix_decompose(U_iHt, qv);
}
// 概率测量读取振幅, 解出 |x>
QProg qprog = createEmptyQProg() << qcir;
qvm.directlyRun(qprog);
```

ideal



```
// 绝热演化参数
const int S = 200;     // 总演化阶段步数
const int T = 1;       // 单步哈密顿量模拟时间
// 制备系统初态 |b>
qcir << amplitude_encode(qv, amplitude);
// 制备含时哈密顿量 H_s
H_s = (1 - s) * H0 + s * H1
// 绝热演化
for (int s = 1; s <= S; s++) {
  // 含时哈密顿量近似为不含时
  MatrixXcd H = H_s(float(s) / S);
  // 时间演化算子的一阶近似
  MatrixXcd iHt = exp_iHt_approx(H, T);
  // 谱范数规范化 & 进行块编码
  iHt = normalize_QSVT(iHt);
  MatrixXcd U_iHt = block_encoding_QSVT(iHt);
  // 近似的时间演化算子转化为量子逻辑门线路
  qcir << matrix_decompose(U_iHt, qv);
}
// 概率测量读取振幅, 解出 |x>
QProg qprog = createEmptyQProg() << qcir;
qvm.directlyRun(qprog);
```

contest-specified

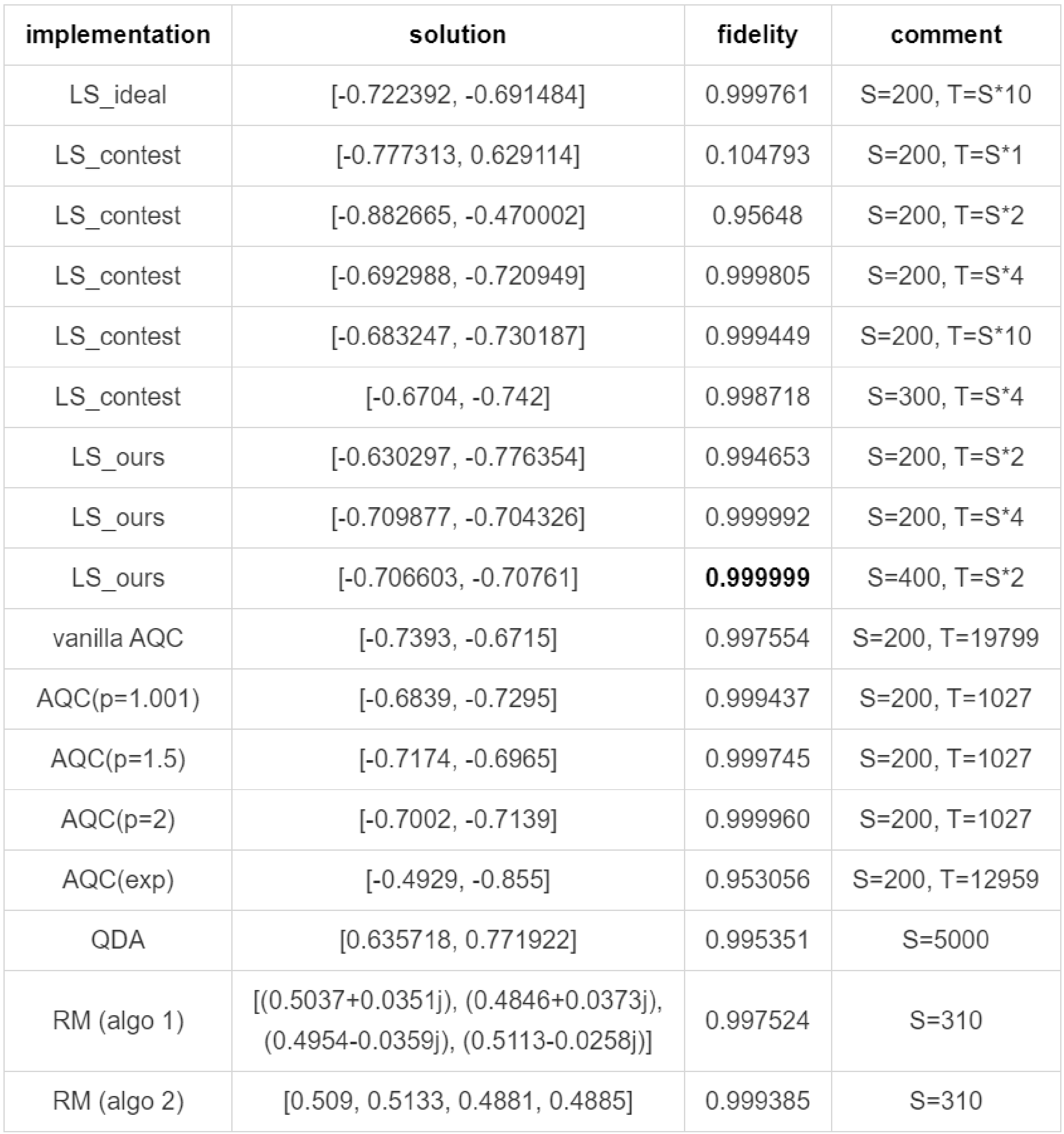


```
// 绝热演化参数
const int S = 300;     // 总演化阶段步数
const int T = 10;      // 单步哈密顿量模拟时间
// 制备系统初态 |b>
qcir << amplitude_encode(qv, amplitude);
// 制备含时哈密顿量 H_s, 使用 AQC(P=2) 调度函数 f(s)
H_s = (1 - f(s)) * H0 + f(s) * H1
// 绝热演化
// adiabatic evolution
for (int s = 1; s <= S; s++) {
  // 含时哈密顿量近似为不含时
  MatrixXcd H = H_s(float(s) / S);
  // 时间演化算子的二阶近似
  MatrixXcd iHt = exp_iHt_approx(H, T, 2);
  // 谱范数规范化 & 进行块编码
  iHt = normalize_QSVT(iHt);
  MatrixXcd U_iHt = block_encoding_QSVT(iHt);
  // 近似的时间演化算子转化为量子逻辑门线路
  qcir << matrix_decompose(, U_iHt, qv);
}
// 制作特征滤波矩阵 & 进行块编码
MatrixXcd EF = EF_R_l(H1);
MatrixXcd U_EF = block_encoding_QSVT(EF);
// 特征滤波矩阵转化为量子逻辑门线路
qcir << matrix_decompose(U_EF, qv);
// 概率测量读取振幅, 解出 |x>
QProg qprog = createEmptyQProg() << qcir;
qvm.directlyRun(qprog);
```

ours

Numerical Results

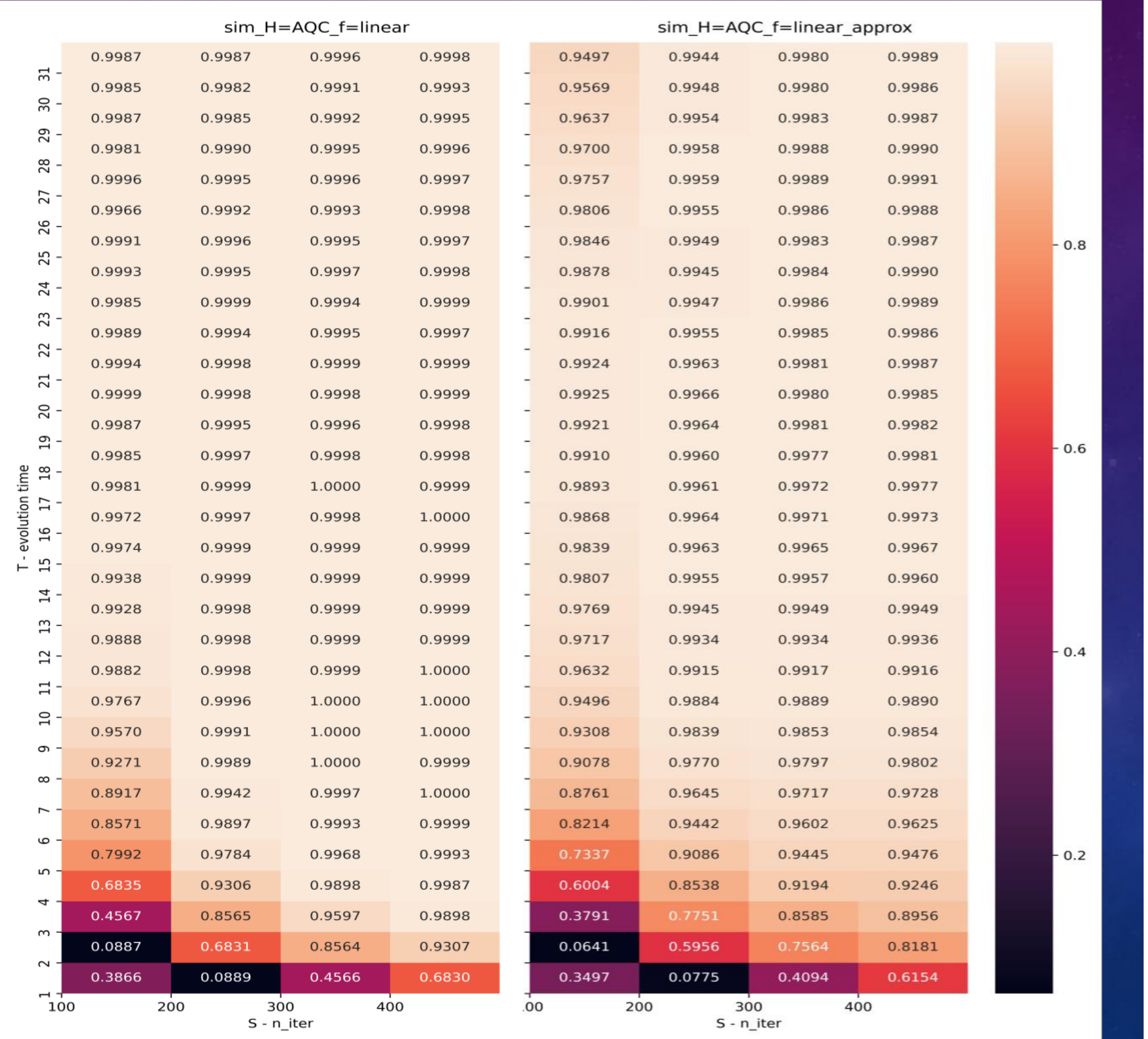
- RM & AQC as baseline
  - QDA leave to discuss
- LS as our final submission
  - ideal: TE ideal + linear
  - contest: TE approx. BE + linear
  - ours: TE approx. BE + AQC(P=2) + EF
- Precision reference
  - RM for ε = 0.01
    - N=16, κ ≈ 10 → S ≈ 500
    - N=32, κ ≈ 50 → S ≈ 1000
  - AQC(2) for ε = 0.001
    - κ = 20 → T ≈ 200



implementation	solution	fidelity	comment
LS_ideal	[-0.722392, -0.691484]	0.999761	S=200, T=S*10
LS_contest	[-0.777313, 0.629114]	0.104793	S=200, T=S*1
LS_contest	[-0.882665, -0.470002]	0.95648	S=200, T=S*2
LS_contest	[-0.692988, -0.720949]	0.999805	S=200, T=S*4
LS_contest	[-0.683247, -0.730187]	0.999449	S=200, T=S*10
LS_contest	[-0.6704, -0.742]	0.998718	S=300, T=S*4
LS_ours	[-0.630297, -0.776354]	0.994653	S=200, T=S*2
LS_ours	[-0.709877, -0.704326]	0.999992	S=200, T=S*4
LS_ours	[-0.706603, -0.70761]	**0.999999**	S=400, T=S*2
vanilla AQC	[-0.7393, -0.6715]	0.997554	S=200, T=19799
AQC(p=1.001)	[-0.6839, -0.7295]	0.999437	S=200, T=1027
AQC(p=1.5)	[-0.7174, -0.6965]	0.999745	S=200, T=1027
AQC(p=2)	[-0.7002, -0.7139]	0.999960	S=200, T=1027
AQC(exp)	[-0.4929, -0.855]	0.953056	S=200, T=12959
QDA	[0.635718, 0.771922]	0.995351	S=5000
RM (algo 1)	[(0.5037+0.0351j), (0.4846+0.0373j), (0.4954-0.0359j), (0.5113-0.0258j)]	0.997524	S=310
RM (algo 2)	[0.509, 0.5133, 0.4881, 0.4885]	0.999385	S=310

Error Analysis

- Impact factor: schedule f(s) > TE approx. > steps S > time T

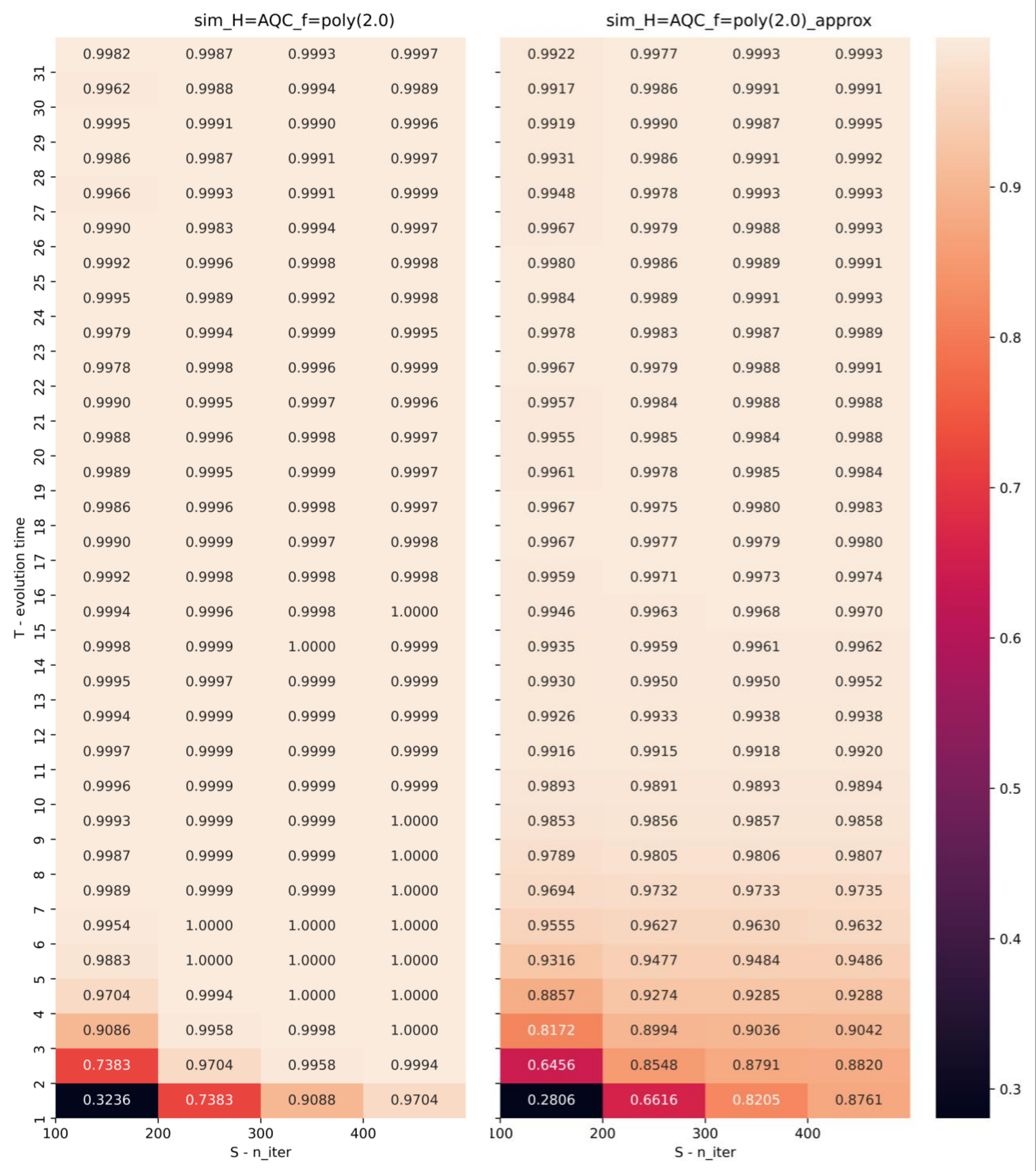


sim_H=AQC_f=linear

	100	200	300	400
31	0.9987	0.9987	0.9996	0.9998
30	0.9985	0.9982	0.9991	0.9993
29	0.9987	0.9985	0.9992	0.9995
28	0.9981	0.9990	0.9995	0.9996
27	0.9996	0.9995	0.9996	0.9997
26	0.9966	0.9992	0.9993	0.9998
25	0.9991	0.9996	0.9995	0.9997
24	0.9993	0.9995	0.9997	0.9998
23	0.9985	0.9999	0.9994	0.9999
22	0.9989	0.9994	0.9995	0.9997
21	0.9994	0.9998	0.9999	0.9999
20	0.9999	0.9998	0.9998	0.9999
19	0.9987	0.9995	0.9996	0.9998
18	0.9985	0.9997	0.9998	0.9998
17	0.9981	0.9999	1.0000	0.9999
16	0.9972	0.9997	0.9998	1.0000
15	0.9974	0.9999	0.9999	0.9999
14	0.9938	0.9999	0.9999	0.9999
13	0.9928	0.9998	0.9999	0.9999
12	0.9888	0.9998	0.9999	0.9999
11	0.9882	0.9998	0.9999	1.0000
10	0.9767	0.9996	1.0000	1.0000
9	0.9570	0.9991	1.0000	1.0000
8	0.9271	0.9989	1.0000	0.9999
7	0.8917	0.9942	0.9997	1.0000
6	0.8571	0.9897	0.9993	0.9999
5	0.7992	0.9784	0.9968	0.9993
4	0.6835	0.9306	0.9898	0.9987
3	0.4567	0.8565	0.9597	0.9898
2	0.0887	0.6831	0.8564	0.9307
1	0.3866	0.0889	0.4566	0.6830

sim_H=AQC_f=linear_approx

	100	200	300	400
	0.9497	0.9944	0.9980	0.9989
	0.9569	0.9948	0.9980	0.9986
	0.9637	0.9954	0.9983	0.9987
	0.9700	0.9958	0.9988	0.9990
	0.9757	0.9959	0.9989	0.9991
	0.9806	0.9955	0.9986	0.9988
	0.9846	0.9949	0.9983	0.9987
	0.9878	0.9945	0.9984	0.9990
	0.9901	0.9947	0.9986	0.9989
	0.9916	0.9955	0.9985	0.9986
	0.9924	0.9963	0.9981	0.9987
	0.9925	0.9966	0.9980	0.9985
	0.9921	0.9964	0.9981	0.9982
	0.9910	0.9960	0.9977	0.9981
	0.9893	0.9961	0.9972	0.9977
	0.9868	0.9964	0.9971	0.9973
	0.9839	0.9963	0.9965	0.9967
	0.9807	0.9955	0.9957	0.9960
	0.9769	0.9945	0.9949	0.9949
	0.9717	0.9934	0.9934	0.9936
	0.9632	0.9915	0.9917	0.9916
	0.9496	0.9884	0.9889	0.9890
	0.9308	0.9839	0.9853	0.9854
	0.9078	0.9770	0.9797	0.9802
	0.8761	0.9645	0.9717	0.9728
	0.8214	0.9442	0.9602	0.9625
	0.7337	0.9086	0.9445	0.9476
	0.6004	0.8538	0.9194	0.9246
	0.3791	0.7751	0.8585	0.8956
	0.0641	0.5956	0.7564	0.8181
	0.3497	0.0775	0.4094	0.6154



sim_H=AQC_f=poly(2.0)

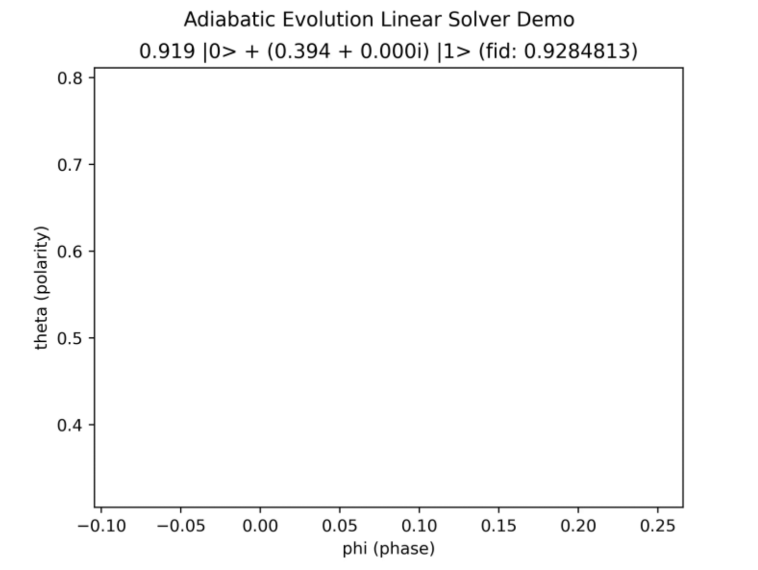
	100	200	300	400
31	0.9982	0.9987	0.9993	0.9997
30	0.9962	0.9988	0.9994	0.9989
29	0.9995	0.9991	0.9990	0.9996
28	0.9986	0.9987	0.9991	0.9997
27	0.9966	0.9993	0.9991	0.9999
26	0.9990	0.9983	0.9994	0.9997
25	0.9992	0.9996	0.9998	0.9998
24	0.9995	0.9989	0.9992	0.9998
23	0.9979	0.9994	0.9999	0.9995
22	0.9978	0.9998	0.9996	0.9999
21	0.9990	0.9995	0.9997	0.9996
20	0.9988	0.9996	0.9998	0.9997
19	0.9989	0.9995	0.9999	0.9997
18	0.9986	0.9996	0.9998	0.9997
17	0.9990	0.9999	0.9997	0.9998
16	0.9992	0.9998	0.9998	0.9998
15	0.9994	0.9996	0.9998	1.0000
14	0.9998	0.9999	1.0000	0.9999
13	0.9995	0.9997	0.9999	0.9999
12	0.9994	0.9999	0.9999	0.9999
11	0.9997	0.9999	0.9999	0.9999
10	0.9996	0.9999	0.9999	0.9999
9	0.9993	0.9999	0.9999	1.0000
8	0.9987	0.9999	0.9999	1.0000
7	0.9989	0.9999	0.9999	1.0000
6	0.9954	1.0000	1.0000	1.0000
5	0.9883	1.0000	1.0000	1.0000
4	0.9704	0.9994	1.0000	1.0000
3	0.9086	0.9958	0.9998	1.0000
2	0.7383	0.9704	0.9958	0.9994
1	0.3236	0.7383	0.9088	0.9704

sim_H=AQC_f=poly(2.0)_approx

	100	200	300	400
	0.9922	0.9977	0.9993	0.9993
	0.9917	0.9986	0.9991	0.9991
	0.9919	0.9990	0.9987	0.9995
	0.9931	0.9986	0.9991	0.9992
	0.9948	0.9978	0.9993	0.9993
	0.9967	0.9979	0.9988	0.9993
	0.9980	0.9986	0.9989	0.9991
	0.9984	0.9989	0.9991	0.9993
	0.9978	0.9983	0.9987	0.9989
	0.9967	0.9979	0.9988	0.9991
	0.9957	0.9984	0.9988	0.9988
	0.9955	0.9985	0.9984	0.9988
	0.9961	0.9978	0.9985	0.9984
	0.9967	0.9975	0.9980	0.9983
	0.9967	0.9977	0.9979	0.9980
	0.9959	0.9971	0.9973	0.9974
	0.9946	0.9963	0.9968	0.9970
	0.9935	0.9959	0.9961	0.9962
	0.9930	0.9950	0.9950	0.9952
	0.9926	0.9933	0.9938	0.9938
	0.9916	0.9915	0.9918	0.9920
	0.9893	0.9891	0.9893	0.9894
	0.9853	0.9856	0.9857	0.9858
	0.9789	0.9805	0.9806	0.9807
	0.9694	0.9732	0.9733	0.9735
	0.9555	0.9627	0.9630	0.9632
	0.9316	0.9477	0.9484	0.9486
	0.8857	0.9274	0.9285	0.9288
	0.8172	0.8994	0.9036	0.9042
	0.6456	0.8548	0.8791	0.8820
	0.2806	0.6616	0.8205	0.8761

Visualize the Adiabatic Solver Solutions

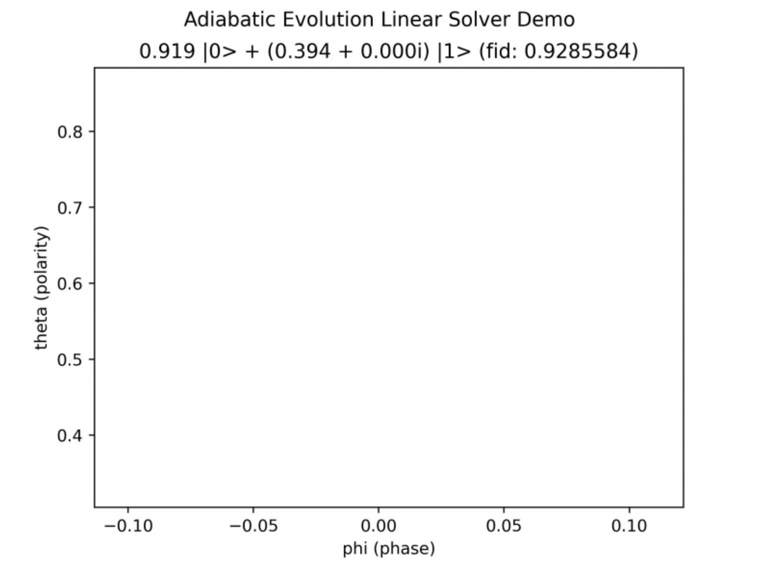
- These are NOT exactly our LS implementations, but simplified version just for concept sketch
  - phase shift may be caused by floating-point error, temporarily ignored…



Adiabatic Evolution Linear Solver Demo

0.919 |0> + (0.394 + 0.000i) |1> (fid: 0.9284813)

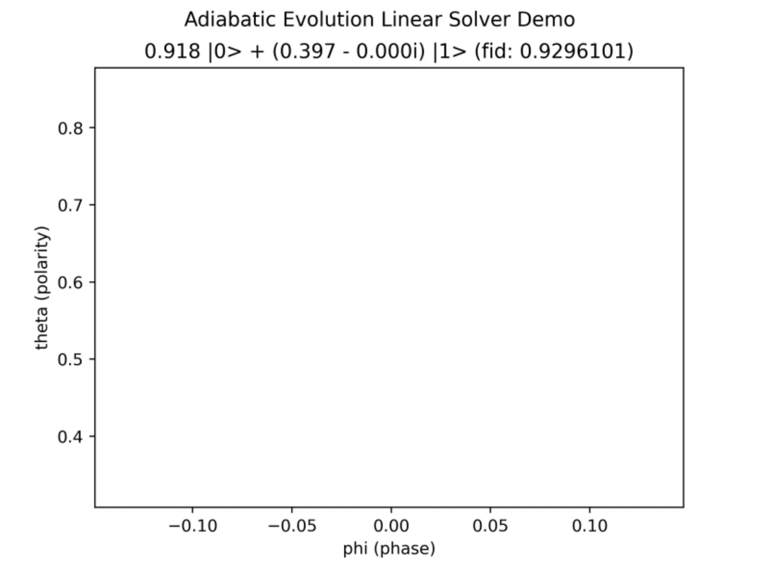
S=1000
f=linear



Adiabatic Evolution Linear Solver Demo

0.919 |0> + (0.394 + 0.000i) |1> (fid: 0.9285584)

S=1000
f=poly(2.0)



Adiabatic Evolution Linear Solver Demo

0.918 |0> + (0.397 - 0.000i) |1> (fid: 0.9296101)

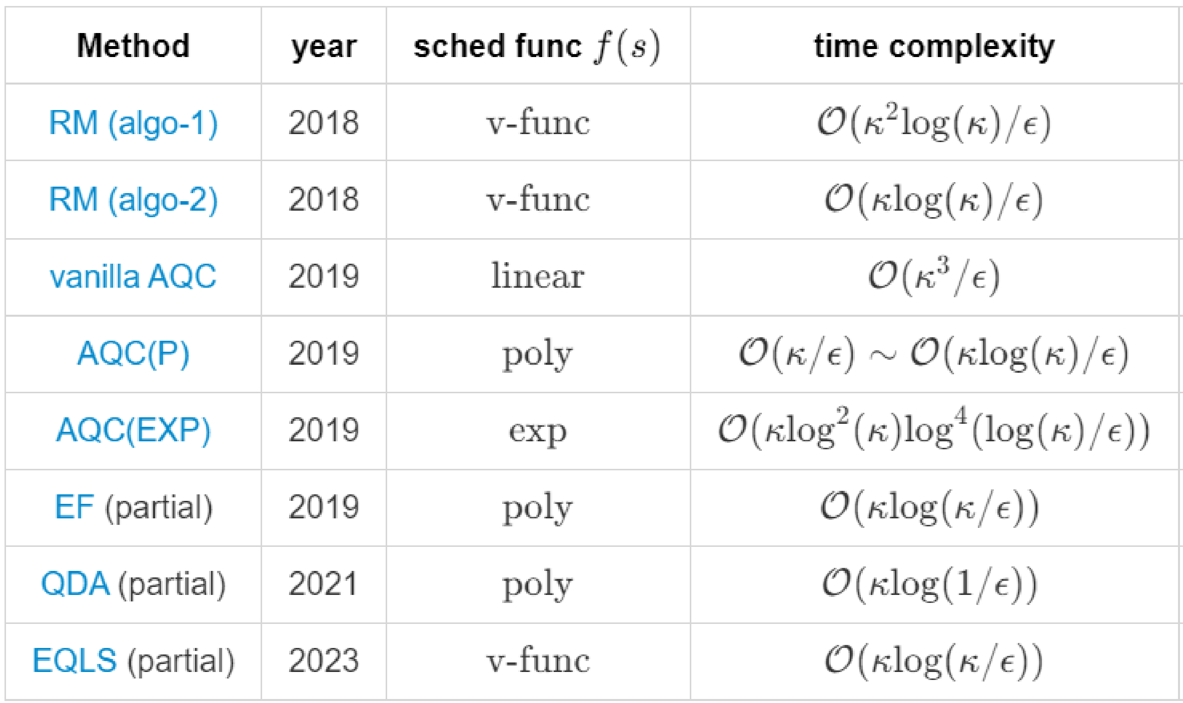
S=200
f=poly(2.0)

Rethinkings 🤔

Part 3

The Linear Solver “Losers” in Complexity Race

- RM: randomized evolution time
  - predictable running time
  - loyal error bound
- AQC(P/EXP): time-optimal f(s) schedule
  - AQC really better than RM v-func?
- QEF = AQC(P) + EF (via QSVT)
- QDA = AQC(P) + QWalk + EF (via LCU)
  - QWalk really better than Dyson series?
- EQLS = RM + EF
  - nothing new, but you might be the wise 😲

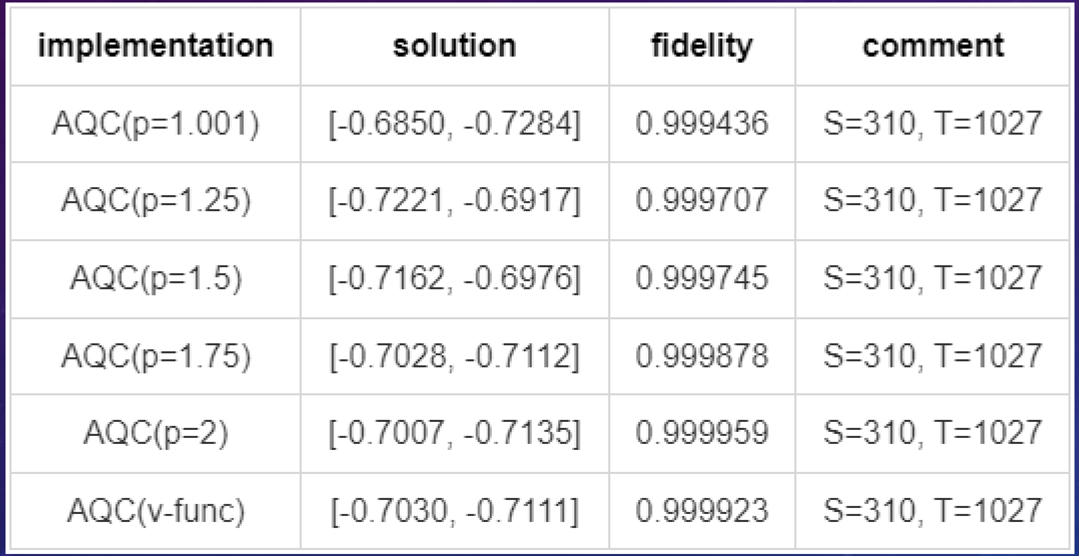


Method	year	sched func $f(s)$	time complexity
RM (algo-1)	2018	v-func	$\mathcal{O}(\kappa^2\log(\kappa)/\epsilon)$
RM (algo-2)	2018	v-func	$\mathcal{O}(\kappa\log(\kappa)/\epsilon)$
vanilla AQC	2019	linear	$\mathcal{O}(\kappa^3/\epsilon)$
AQC(P)	2019	poly	$\mathcal{O}(\kappa/\epsilon) \sim \mathcal{O}(\kappa\log(\kappa)/\epsilon)$
AQC(EXP)	2019	exp	$\mathcal{O}(\kappa\log^2(\kappa)\log^4(\log(\kappa)/\epsilon))$
EF (partial)	2019	poly	$\mathcal{O}(\kappa\log(\kappa/\epsilon))$
QDA (partial)	2021	poly	$\mathcal{O}(\kappa\log(1/\epsilon))$
EQLS (partial)	2023	v-func	$\mathcal{O}(\kappa\log(\kappa/\epsilon))$

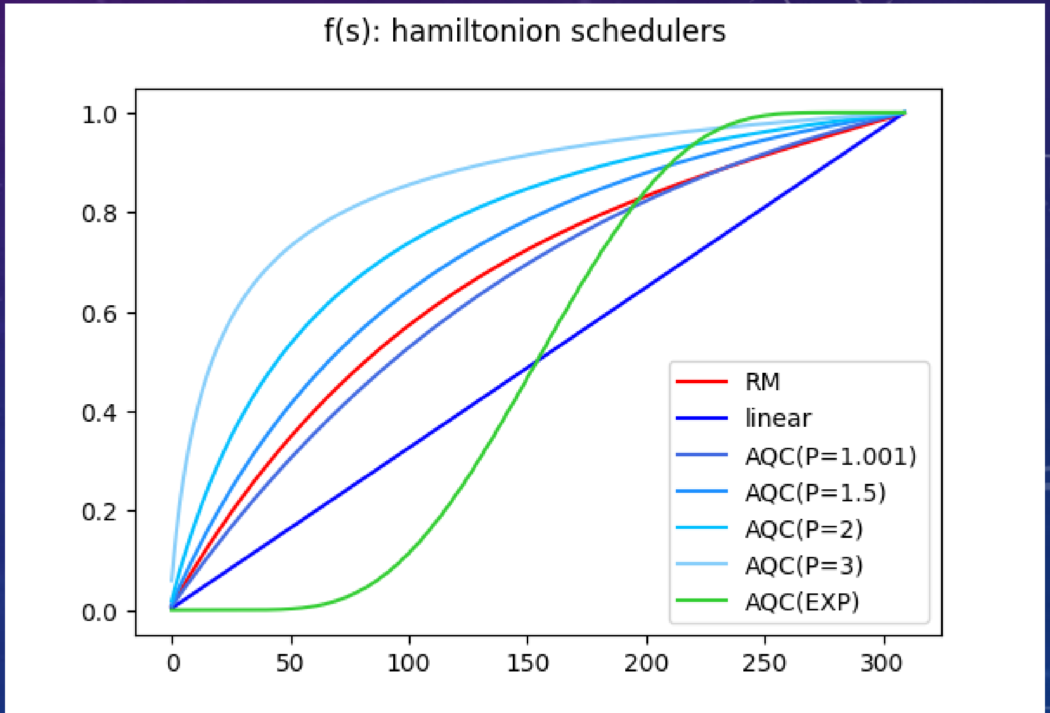
Literature Chronicle

The AQC Myths (arXiv:1909.05500)

- AQC(P) ~ RM v-func



implementation	solution	fidelity	comment
AQC(p=1.001)	[-0.6850, -0.7284]	0.999436	S=310, T=1027
AQC(p=1.25)	[-0.7221, -0.6917]	0.999707	S=310, T=1027
AQC(p=1.5)	[-0.7162, -0.6976]	0.999745	S=310, T=1027
AQC(p=1.75)	[-0.7028, -0.7112]	0.999878	S=310, T=1027
AQC(p=2)	[-0.7007, -0.7135]	0.999959	S=310, T=1027
AQC(v-func)	[-0.7030, -0.7111]	0.999923	S=310, T=1027



Note that fidelity also counts for ancillas, in this case AQC(v-func) gives a slight better results than AQC(p=2)

The QDA Myths (arXiv:2111.08152/2312.07690)

- QDA = W_T + EF
  - $W_T(s) = Reflect\left(BE\left(H(s)\right)\right) = -I + \frac{H(s)}{\lambda} \sim -e^{H} \sim e^{-iH}$
  - Up to a global phase, why can this kick out $\mathcal{O}(\log\kappa)$ in Dyson series??
- Practically far too slow to meet a acceptable precision
  - T=50000 in the essay setting, is it universal??

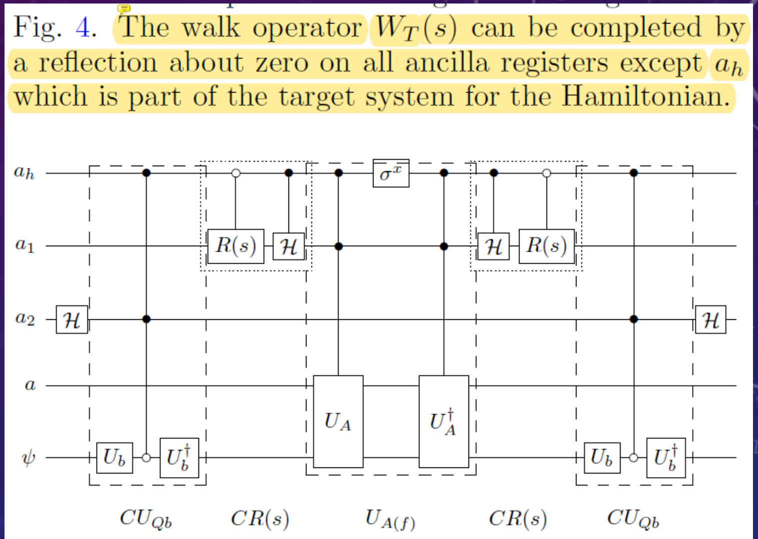
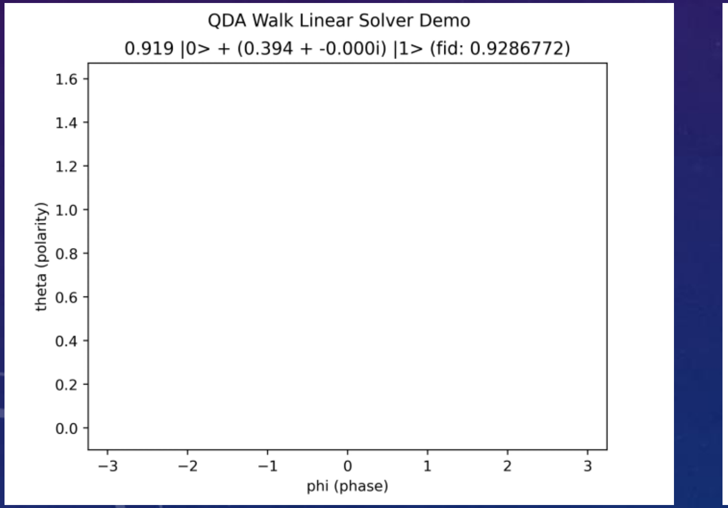


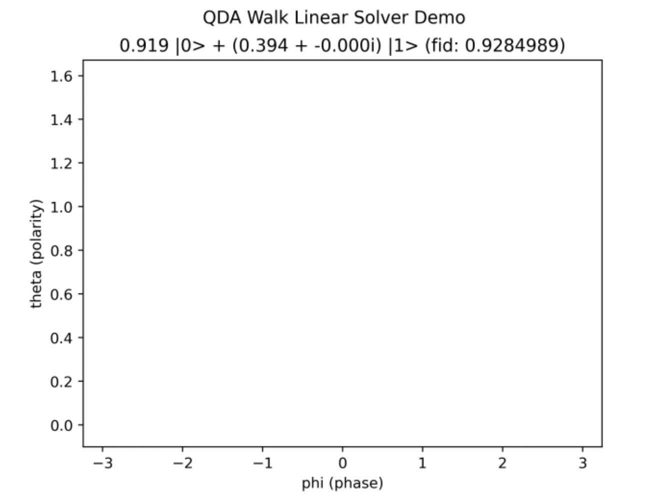
Fig. 4. The walk operator $W_T(s)$ can be completed by a reflection about zero on all ancilla registers except a_h which is part of the target system for the Hamiltonian.

BE upon another BE



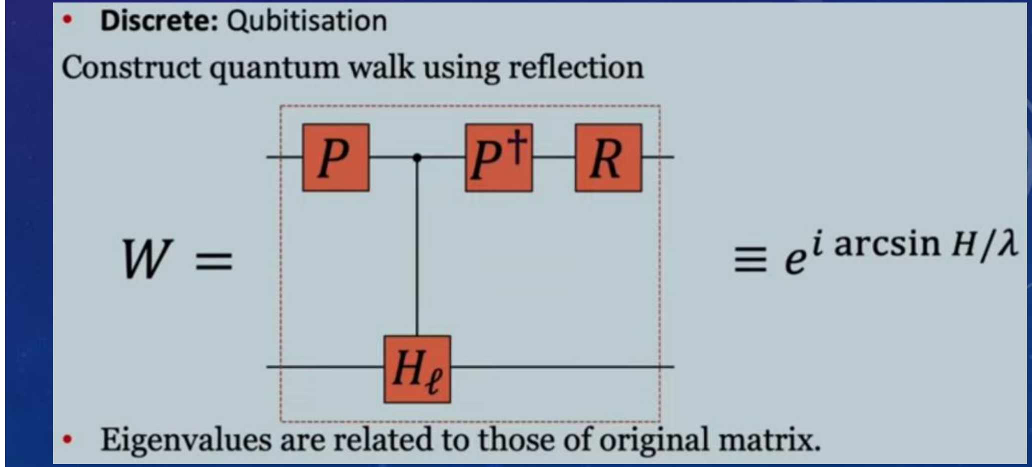
QDA Walk Linear Solver Demo
0.919 |0> + (0.394 + -0.000i) |1> (fid: 0.9286772)

T=500



QDA Walk Linear Solver Demo
0.919 |0> + (0.394 + -0.000i) |1> (fid: 0.9284989)

T=5000



- **Discrete:** Qubitisation

Construct quantum walk using reflection

- Eigenvalues are related to those of original matrix.

https://www.youtube.com/watch?v=WfByvOf3N3Y

Core References

- [RM] Quantum algorithms for systems of linear equations inspired by adiabatic quantum computing (arXiv:1805.10549)
- [AQC] Quantum linear system solver based on time-optimal adiabatic quantum computing and quantum approximate optimization algorithm (arXiv:1909.05500)
- [EF] Optimal polynomial based quantum eigenstate filtering with application to solving quantum linear systems (arXiv:1910.14596)
- [QDA] Optimal scaling quantum linear systems solver via discrete adiabatic theorem (arXiv:2111.08152)
- [EQLS]Efficient quantum linear solver algorithm with detailed running costs (arXiv:2305.11352)
- https://pennylane.ai/qml/demos/tutorial_intro_qsvt/
- https://pennylane.ai/qml/demos/tutorial_lcu_blockencoding/
- https://pennylane.ai/qml/demos/tutorial_block_encoding/
- FABLE: Fast Approximate Quantum Circuits for Block-Encodings (arXiv:2205.00081)
- Evaluation of block encoding for sparse matrix inversion using QSVT (arXiv:2402.17529)

Citation

- Github: https://github.com/Kahsolt/Adiabatic-Linear-Solver-QPanda
- If you find this work useful, please give us a star ⭐ and cite~

```
@misc{kahsolt2024,
    author = {Kahsolt},
    title = {Adiabatic Linear Solver QPanda},
    howpublished = {\url{https://github.com/Kahsolt/Adiabatic-Linear-Solver-QPanda}}
    month = {June},
    year = {2024}
}
```

Thanks for your watching 🎉