HHL Algorithm 的研究

物理 32 杨泽宇 2023011329

一 综述

线性方程组 $A\boldsymbol{x} = \boldsymbol{b}$ 的求解是科学与工程中常见的问题. 随着数据规模的增长，处理线性方程组所需的计算时间也随之增加. 传统算法通常需要 $\mathcal{O}(N)$ 的时间来近似求解 N 个未知数的线性方程组，且仅仅是写出解就需要 $\mathcal{O}(N)$ 的时间. 然而许多情况下，我们并不关心完整的解 $\boldsymbol{x}$，而只需要解的某个函数，例如 $\boldsymbol{x}^T M\boldsymbol{x}$ 的期望值.

为了解决这个问题，Harrow、Hassidim 和 Lloyd 在 2009 年提出了一种量子算法 [1]，通称为 HHL 算法. 对于稀疏矩阵 A（$N \times N$ 维，条件数 κ），最快的经典算法求解 $\boldsymbol{x}$ 并估计 $\boldsymbol{x}^T M\boldsymbol{x}$ 的时间复杂度约为 $\mathcal{O}(N^\kappa)$. 而 Harrow 等三人发现 HHL 算法估计 $\boldsymbol{x}^T M\boldsymbol{x}$ 的运行时间约为 $\log(N)$ 和 κ 的多项式函数；对于小 κ 值，则变为 $\mathcal{O}(\text{poly}(\log N))$. 由此可见，HHL 在通常情况下算法比经典算法快指数级，具有重要应用价值.

作者三人在论文中首先介绍了 HHL 算法的基本步骤：初始态制备，**量子相位估计（QPE），受控旋转，逆量子相位估计**，测量和最后的期望值评估. 其中，QPE，受控旋转和逆 QPE 是算法能够有效实现的核心.

随后文章进行了复杂性分析，指出算法的运行时间主要取决于矩阵 A 的条件数 κ 和所需精度 ϵ，并给出其时间复杂度为 $\mathcal{O}(\log(N)s^2\kappa^2/\epsilon)$，其中 s 为矩阵稀疏度. 如果 κ 和 $1/\epsilon$ 都是 $\text{poly}(\log N)$，则算法可以实现指数级加速. 然而，条件数 κ 常常随问题规模 N 呈多项式甚至指数增长，这会限制算法的实际优势.

文章分析，与经典算法比较，HHL 算法的优势在于它只需要 $\mathcal{O}(\log N)$ 的量子比特寄存器来存储输入状态 $|b\rangle$，而不需要显式地写出矩阵 A、向量 b 或解 x 的所有分量. 这与经典的蒙特卡洛算法有相似之处，后者通过从概率分布中采样而不是写出所有分量来实现加速.

文章最后还讨论了算法的一些应用或推广，例如广义矩阵求逆，处理病态矩阵等.

在此基础上，Dervovic D. 等人于 2019 年发表的著作 Quantum linear systems algorithms: a primer[2] 的第三章中进一步讨论了 HLL 算法，指出其特别适用于高效求解如下的量子线性系统问题（QLSP）：

定义 (QLSP)： 设 A 是一个 $N \times N$ 的 Hermitian 矩阵，且 $\det(\text{A}) = 1$. 设 $\boldsymbol{b}$ 和 $\boldsymbol{x}$ 是 N 维列向量，且满足 $\boldsymbol{x} = A^{-1}\boldsymbol{b}$. 定义 $\lceil \log N \rceil$ 个量子比特上的量子态 $|b\rangle$ 为：

$$|b\rangle := \frac{\sum_{i=0} b_i|i\rangle}{\| \sum_{i=0} b_i|i\rangle \|_2} \tag{1.1}$$

以及 $|x\rangle$ 为：

$$|x\rangle := \frac{\sum_{i=0} x_i|i\rangle}{\| \sum_{i=0} x_i|i\rangle \|_2} \tag{1.2}$$

其中 b_i 和 x_i 分别是向量 b 和 x 的第 i 个分量. 给定矩阵 A（通过 $Oracle$ 访问其元素）和状态 $|b\rangle$，在 $P > 1/2$ 概率下输出一个状态 $|\tilde{x}\rangle$，使得 $\| |\tilde{x}\rangle - |x\rangle \|_2 \leq \epsilon$.

书中先详细推导了 HHL 算法的实现细节，引入了滤波函数等重要概念；随后进行了详细的误差分析，讨论了相位估计和后选择过程中的错误是如何影响算法性能的. 另外在第四章中，作者还简单讨论了 HHL 算法的改进方案，包括指数级提高的精度对于系统规模的依赖性等.

二 论文解析与公式推导

1. HHL 基本原理

为了实现 HHL 算法，我们需要 3 个量子寄存器——寄存器 n_l 用于存储矩阵 A 特征值的二进制表示，寄存器 n_b 用于存储输入状态 $|b\rangle$，另有一个辅助寄存器用于在计算过程中执行中间操作.

首先来做一些数学准备. 我们设 A 的谱分解如下：

$$A=\sum_{j=1}^{N}\lambda_j|u_j\rangle\langle u_j|,\quad \lambda_j\in\mathbb{R} \tag{2.1}$$

其中 λ_j 是 A 的特征值，$|u_j\rangle$ 是对应的特征向量. 由此可以将 A 的逆表示为：

$$A^{-1}=\sum_{j=1}^{N}\frac{1}{\lambda_j}|u_j\rangle\langle u_j|,\quad \lambda_j\neq 0 \tag{2.2}$$

而 $|b\rangle$ 可以按 $\{|u_j\rangle\}$ 展开为：

$$|b\rangle=\sum_{j=1}^{N}\beta_j|u_j\rangle,\quad b_j\in\mathbb{C} \tag{2.2}$$

由此即可得到 $|x\rangle$ 在 $\{|u_j\rangle\}$ 下的表示：

$$|x\rangle=A^{-1}|b\rangle=\sum_{j=1}^{N}\frac{\beta_j}{\lambda_j}|u_j\rangle \tag{2.3}$$

这样，我们就将求解 $|x\rangle$ 的问题转化为对 $|b\rangle$ 进行演化和测量的问题，最后的输出 $|x\rangle$ 与输入 $|b\rangle$ 在同一个寄存器中.

下面具体来看算法的过程. 当然，在之前应先制备好量子态 $|b\rangle$.

首先是 QPE. 给定一个具有特征态 $|u_j\rangle$ 和对应复特征值 $e^{\mathrm{i}\varphi_j t}$ 的酉算子 U，QPE 技术可以实现以下映射：[3]

$$|0\rangle|u_j\rangle\xrightarrow{QPE}|\widetilde{\varphi}_j\rangle|u_j\rangle, \tag{2.4}$$

其中 $|\tilde{\varphi}_j\rangle$ 是 φ_j 的二进制表示，精度为 n_l 位. 若取 $U=e^{\mathrm{i}At}$，则由于其具有特征向量 $\{|u_j\rangle\}$ 和对应特征值 $\{e^{\mathrm{i}\lambda_j t}\}$，从而 QPE 操作对于每一个特征态给出：

$$|0\rangle^{\otimes n_l}|u_j\rangle\xrightarrow{QPE}|\widetilde{\lambda}_j\rangle|u_j\rangle, \tag{2.5}$$

从而对于 $|b\rangle$ 有：

$$|0\rangle^{\otimes n_l}|b\rangle\xrightarrow{QPE}\sum_j e^{\mathrm{i}\lambda_j t}|\widetilde{\lambda}_j\rangle_{n_l}|u_j\rangle_{n_b} \tag{2.6}$$

算法的第二步，是以 $|\tilde{\lambda}_j\rangle$ 为控制位，作用受控的 σ_y-旋转. 为了实现这一过程，我们引入一个初始态为 $|0\rangle$ 的辅助寄存器作为目标位. 旋转矩阵可以表为：

$$R_y(2\theta)=e^{-\mathrm{i}\theta\sigma_y}=\begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix},\qquad \theta=\arccos\left(C/\widetilde{\lambda}_j\right) \tag{2.7}$$

其中 C 是归一化常数. 受控旋转的结果为：

$$\sum_{j=1}^{N}\beta_j|\widetilde{\lambda}_j\rangle|u_j\rangle\left(\sqrt{1-\frac{C^2}{\widetilde{\lambda}_j^2}}|0\rangle+\frac{C}{\widetilde{\lambda}_j}|1\rangle\right) \tag{2.8}$$

此时我们进行逆 QPE 操作，得到：

$$|0\rangle^{\otimes n_l}\otimes\sum_{j=1}^{N}\beta_j|u_j\rangle\left(\sqrt{1-\frac{C^2}{\widetilde{\lambda}_j^2}}|0\rangle+\frac{C}{\widetilde{\lambda}_j}|1\rangle\right) \tag{2.9}$$

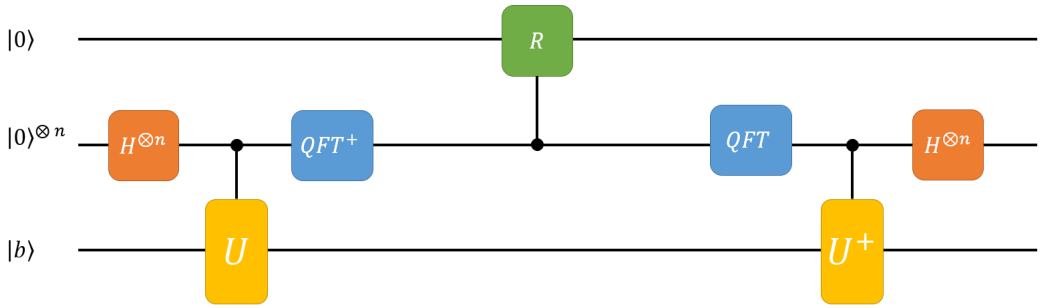


图 1. HHL 算法的基本流程图：预先制备好输入状态 $|b\rangle$；通过先后作用 $H^{\otimes n}, controlled-U$ 和 $QFT^{\dagger}$ 实现 QPE 操作，然后进行受控旋转，再作用第一步的逆操作 $QPE^{\dagger}$，最后测量并输出状态 $|x\rangle$.

最后对辅助位量子比特展开测量，我们得到 $|1\rangle$ 的概率为：

$$p_1 = C^2 \sum_{j=1}^{N} \frac{|\beta_j|^2}{\widetilde{\lambda}_j^2} \tag{2.10}$$

而测量后的量子态变为：

$$\frac{1}{\sqrt{\sum_{j=1}^{N}(\beta_j/\lambda_j)^2}} |0\rangle^{\otimes n_l} \left(\sum_{j=1}^{N} \frac{\beta_j}{\lambda_j} |u_j\rangle \right) |1\rangle \tag{2.11}$$

可见此时三个量子寄存器中不再存在纠缠，其中寄存器 n_b 所处于状态正是我们期待的 $|x\rangle$（忽略掉一个归一化系数）.

以上在 A 为 Hermitian 矩阵的条件下说明了 HHL 算法的有效性. 而当 A 非 Hermitian 时，只需要构造：

$$A'\boldsymbol{x}' = \boldsymbol{b}'; \qquad A' = \begin{pmatrix} 0 & A \\ A^{\dagger} & 0 \end{pmatrix}, \quad \boldsymbol{b}' = \begin{pmatrix} \boldsymbol{b} \\ 0 \end{pmatrix}, \quad \boldsymbol{x}' = \begin{pmatrix} 0 \\ \boldsymbol{x} \end{pmatrix} \tag{2.12}$$

即可将问题化为我们上文已经讨论过的形式，从而 HHL 算法仍然有效.

最后需要指出的是：HHL 算法帮助我们更快地找到了解向量，但代价是它处于量子态，因此我们无法直接获知解的完整信息，而测量并查找其分量需要花费额外的资源.

2. 算法细节

我们先来严格描述 HHL 算法，然后再逐步分析. HHL 算法的过程可由如下伪代码表示：

Algorithm 1 HHL 算法解决 QLSP 问题

输入： 初始态 $|b\rangle$，矩阵 A（元素通过 Oracle 访问），参数 $t_0 = \mathcal{O}(\kappa/\epsilon)$，$T = \widetilde{\mathcal{O}}(\log(N)s^2 t_0)$，$\epsilon$ 为期望精度.

1: 制备输入态 $|\Psi_0\rangle^C \otimes |b\rangle^I$，其中 $|\Psi_0\rangle = \sqrt{\frac{2}{T}} \sum_{\tau=0}^{T-1} \sin \frac{\pi(\tau+\frac{1}{2})}{T} |\tau\rangle^C$;

2: 对输入态应用条件 Hamilton 演化：$\sum_{\tau=0}^{T-1} |\tau\rangle\langle\tau|^C \otimes e^{iA\tau t_0/T}$;

3: 对寄存器 C 应用量子 Fourier 变换，新的基态记为 $|k\rangle$，$k \in \{0, \ldots, T-1\}$，定义 $\tilde{\lambda} := 2\pi k/t_0$.;

4: 附加辅助寄存器 S，并以 C 为控制执行受控旋转，将 $|\tilde{\lambda}\rangle$ 映射为 $|h(\tilde{\lambda})\rangle$;

5: 对寄存器 C 中的“垃圾”信息进行逆 QPE 操作;

6: 测量寄存器 S，结果为“well”时返回寄存器 I，否则返回步骤 1;

7: 在 $\mathcal{A}_{\mathrm{HHL}}(|b\rangle, A, t_0, T, \epsilon)$ 上执行 $\mathcal{O}(\kappa)$ 轮**振幅放大**操作;

输出： 状态 $|\widetilde{x}\rangle$，满足 $\| |\widetilde{x}\rangle - |x\rangle \|_2 \leq \epsilon$.

算法的第一步是制备好初始态 $|b\rangle^I$，并输入寄存器 I 中. 简单起见，假设存在酉算子 B 和初始状态 $|\text{initial}\rangle$ 可以实现完美精度的制备 $B|\text{initial}\rangle^I = |b\rangle^I$，需要的资源为 T_B 个门. 这里 B 是 qRAM Oracle，T_B 是 $|b\rangle$ 维度的多对数函数.

而时钟寄存器 C 则应该被初始化为：

$$|\Psi_0\rangle = \sqrt{\frac{2}{T}} \sum_{\tau=0}^{T-1} \sin\frac{\pi(\tau+\frac{1}{2})}{T} |\tau\rangle^C \tag{2.13}$$

该状态可在时间 $\text{poly}(\log(T/\epsilon_\Psi))$ 内以误差 ϵ_Ψ 制备 [4]. 时间 $T = \tilde{\mathcal{O}}(\log(N)s^2t_0)$ 对应于模拟 e^{iAt} 所需的计算步骤数，其中 t_0/T 是模拟的步长.

接下来，对输入态作用条件 Hamilton 演化 $\sum_{\tau=0}^{T-1} |\tau\rangle\langle\tau|^C \otimes e^{iA\tau t_0/T}$，得到：

$$\sqrt{\frac{2}{T}} \sum_{j=1}^{N} \beta_j \left(\sum_{\tau=0}^{T-1} e^{\mathrm{i}\frac{\lambda_j t_0 \tau}{T}} \sin\frac{\pi(\tau+\frac{1}{2})}{T} |\tau\rangle^C \right) |u_j\rangle^I \tag{2.14}$$

此时作量子 Fourier 变换：将式 (2.14) 括号内的部分与 Fourier 基态 $\frac{1}{\sqrt{T}} \sum_{k=0}^{T-1} e^{-\mathrm{i}2\pi k\tau/T}, |k\rangle$ 取内积，即可将即可将寄存器 C 变换为 Fourier 基 $|k\rangle$ 下的状态：

$$\sum_{j=1}^{N} \beta_j \sum_{k=0}^{T-1} \underbrace{\left(\frac{\sqrt{2}}{T} \sum_{\tau=0}^{T-1} e^{\mathrm{i}\frac{\pi}{T}(\lambda_j t_0 - 2\pi k)} \sin\frac{\pi(\tau+\frac{1}{2})}{T} \right)}_{\alpha_{k|j}} |k\rangle^C |u_j\rangle^I \tag{2.15}$$

定义 $\widetilde{\lambda}_k := 2\pi k/t_0$ 重新标记基础态 $|k\rangle$，得到：

$$\sum_{j=1}^{N} \beta_j \sum_{k=0}^{T-1} \alpha_{k|j} \left|\widetilde{\lambda}_k\right\rangle^C |u_j\rangle^I . \tag{2.16}$$

其中系数 $\alpha_{k|j}$ 定义为:

$$\alpha_{k|j} = \frac{\sqrt{2}}{T} \sum_{\tau=0}^{T-1} e^{\mathrm{i}\frac{\pi}{T}(\lambda_j t_0 - 2\pi k)} \sin\frac{\pi(\tau+\frac{1}{2})}{T} \tag{2.17}$$

可以验证，若定义 $\delta := \lambda_j t_0 - 2\pi k$，则当 $|k - \lambda_j t_0/2\pi| \geq 1$ 时 $\alpha_{k|j}$ 存在上界：[1]

$$\left|\alpha_{k|j}\right|^2 \leq \frac{64\pi^2}{\delta^2} \tag{2.18}$$

可见 $\left|\alpha_{k|j}\right|$ 仅在 $\lambda_j \approx \frac{2\pi k}{t_0}$ 时取到较大值.

以上步骤对应于我们前文提及的 QPE 操作，然而其细节有很大的不同. 本质上来说，**这是因为我们采用了 Hamilton 模拟技术来实现 $controlled - U$ 的操作，即可以认为 HHL 中实际运用的是标准 QPE 技术的一种变体.**

具体来说,标准 QPE 的执行依赖于 QFT 和受控 U^{2^k} 操作,需要一系列精确的幂次酉算子 $(U, U^2, U^4, \ldots)$. 而在 HHL 算法中，我们通过 Hamilton 模拟 $e^{\mathrm{i}A\tau t_0/T}$ 替代了酉算子 U，并引入时钟寄存器 $|\tau\rangle^C$ 和条件演化操作 $\sum_\tau |\tau\rangle\langle\tau|^C \otimes e^{iA\tau t_0/T}$，将时间步长离散化. 此时，我们依赖几何序列求和来分析相位误差，而非直接测量寄存器 C 的二进制位.

因此，HHL 中“QPE”操作的误差来源于步长 t_0/T 的离散取值；在后续过程中，滤波函数对特征值的截断（见后文）也会对特征值的估计产生影响.

不过需要注意的是，HHL 中的“QPE”操作和标准 QPE 操作**在数学上仍然是等价的**. 两者都是基于 QFT 的原理，制备不同寄存器之间的量子纠缠态，并通过受控门提取相位信息. 事实上，我们当然可以直接采用标准 QPE 的方式来实现 HHL 算法作为简化，但这可能会导致额外的资源消耗.

算法的下一步，是引入一个辅助寄存器 S 用于执行受控反转.

这里需要重点考虑的是算法的**数值稳定性**. 在 HHL 算法的运用过程中，我们希望只翻转矩阵条件“良好”的部分，即特征值位于某个范围内的部分，该范围相对于 $1/\kappa$ 较大（否则 A^{-1} 的返回值将与真实值偏离到不可接受的程度）.

为了实现这一点，我们引入**滤波器函数** $f(\lambda)$ 和 $g(\lambda)$，它们的作用是仅在 A 的良好条件子空间（即由特征值 $\lambda \geq 1/\kappa$ 对应的特征向量张成的子空间）上反转 A，确保 λ 的微小误差不会在 A^{-1} 中引入大的误差.

在 HHL 算法中，受控旋转由形式为 $\theta = \arccos x$ 的角度控制，其中 x 是滤波器函数对 λ 的输出. $\arccos(\cdot)$ 的任何参数都需要在区间 $[-1,1]$ 内，因此滤波器函数的值域必须是 $[-1,1]$。滤波器函数的定义域是 $[\lambda_{min},\lambda_{max}]$. 我们要求在良好条件子空间中，滤波器函数与 $1/\lambda$ 成比例，以实现特征值反转；我们还关注中间特征值，即 $1/\kappa' \leq \lambda \leq 1/\kappa$，在这些情况下需要采取插值行为以提高数值稳定性. 满足所有这些要求的滤波器函数选择之一为：

$$f(\lambda)=\begin{cases}\frac{1}{2\kappa\lambda}, & \lambda \geq 1/\kappa;\\ \frac{1}{2}\sin\left(\frac{\pi}{2}\cdot\frac{\lambda-\frac{1}{\kappa}}{\frac{1}{\kappa}-\frac{1}{\kappa'}}\right), & \frac{1}{\kappa}>\lambda>\frac{1}{\kappa'};\\ 0, & \frac{1}{\kappa'}>\lambda;\end{cases} \qquad g(\lambda)=\begin{cases}0, & \lambda \geq 1/\kappa;\\ \frac{1}{2}\cos\left(\frac{\pi}{2}\cdot\frac{\lambda-\frac{1}{\kappa}}{\frac{1}{\kappa}-\frac{1}{\kappa'}}\right), & \frac{1}{\kappa}>\lambda>\frac{1}{\kappa'};\\ \frac{1}{2}, & \frac{1}{\kappa'}>\lambda.\end{cases} \tag{2.19}$$

注意这一步在算法中引入了 κ 依赖性.

那么，在受控旋转之后，寄存器 S 的状态将变为：

$$|h(\widetilde{\lambda}_k)\rangle^S := \sqrt{1-f(\widetilde{\lambda}_k)^2-g(\widetilde{\lambda}_k)^2}\,|\text{nothing}\rangle^S + f(\widetilde{\lambda}_k)\,|\text{well}\rangle^S + g(\widetilde{\lambda}_k)\,|\text{ill}\rangle^S . \tag{2.20}$$

其中标注“nothing”表示未进行反转，“well”表示已反转，“ill”表示 $|b\rangle$ 处于 A 的不良条件子空间的部分。

应用滤波器函数后，我们执行逆 QPE 过程，同时清除计算过程中的“垃圾”量子位. 到这里为止，我们就完成了一般 HHL 算法的核心步骤.

然而，就像式 (2.10) 展现的那样，如果算法到此为止，我们将不能以 1 的概率在寄存器 C 上测得 $|1\rangle$，从而会导致很多冗余操作无法获得 $|x\rangle$ 的有效估计. 为此，我们需要采取**振幅放大**的技术.[3] 现在，若将到目前为止描述的过程记为 U_{invert}，则将 U_{invert} 应用于 $|b\rangle$ 并测量 S 得到结果“well”时，返回状态 $|\tilde{x}\rangle$，成功的概率将为 $\tilde{p}=\mathcal{O}(1/\kappa^2)$. 运用振幅放大的原理，通过 $\mathcal{O}\left(1/\sqrt{\tilde{p}}\right)=\mathcal{O}(\kappa)$ 次重复，我们即可以实现任意高的成功概率.

之前提到，制备状态 $|b\rangle$ 需要 $\widetilde{O}(T_B)$，运行量子模拟需要 $\widetilde{O}(t_0 s^2 \log N)$。因此，算法的总运行时间为 $\widetilde{O}(\kappa(T_B+t_0 s^2\log N))$，其中因子 κ 是由于幅度放大。由于 $t_0=\mathcal{O}(\kappa/\epsilon)$，运行时间可以表示为 $\widetilde{O}(\kappa T_B+\kappa^2 s^2\log(N)/\epsilon)$.

三 个人分析：初始态制备的优化

关于初始态制备的假设，HHL 算法要求输入态 $|b\rangle$ 能高效制备，然而这在通用场景下并非易事.

一个创新思路是将状态制备与矩阵逆的计算融合为单一量子过程。例如，若 $|b\rangle$ 由某 Hamilton 量 H' 的基态制备，则可设计联合量子线路同时对角化 H' 和 A，通过量子虚时间演化模拟 $e^{-\tau H_{\text{joint}}}|_{\text{init}}$，其稳态即有可能包含 $|A^{-1}b$ 的分量，从而直接输出 $|A^{-1}b$. 这种方法需解决多体 Hamilton 量模拟的复杂度问题，但有望减少预处理步骤.

为了进一步探讨状态制备与矩阵求逆的融合策略，我们首先明确问题设定：给定一个需要求解的线性系统 $A\mathbf{x}=\mathbf{b}$，其中 $\mathbf{b}$ 是某哈密顿量 H' 的基态（即 $|b\rangle=|\psi_0'\rangle$，满足 $H'|\psi_0'\rangle=\lambda_0'|\psi_0'\rangle$）。传统 HHL 算法需独立完成 $|b\rangle$ 的制备和 A^{-1} 的作用，而我们的目标是设计一个联合量子过程，直接输出 $|x\rangle\propto A^{-1}|b\rangle$. 下面我们通过尝试来逐步实现这一目标.

1. 尝试构造联合 Hamilton 量

首先，尝试定义联合哈密顿量：

$$H_{\text{joint}} = H' \otimes I + I \otimes A \tag{3.1}$$

其作用在复合系统 $\mathcal{H}_{\text{total}} = \mathcal{H}_{H'} \otimes \mathcal{H}_A$ 上. 假设 H' 和 A 分别具有谱分解：

$$H' = \sum_i \lambda_i' |\psi_i'\rangle\langle\psi_i'|, \quad A = \sum_j \lambda_j |u_j\rangle\langle u_j| \tag{3.2}$$

则 H_{joint} 的本征态为直积态 $|\psi_i'\rangle \otimes |u_j\rangle$，对应本征值 $\lambda_i' + \lambda_j$. 特别地，基态为 $|\psi_0'\rangle \otimes |u_{\min}\rangle$，其中 $|u_{\min}\rangle$ 是 A 的最小本征值态.

现在，我们进行量子虚时间演化（Quantum Imaginary Time Evolution, QITE）操作，通过非酉算子 $e^{-\tau H}$ 将初始态 $|\psi_{\text{init}}\rangle$ 投影至基态. 对于联合系统而言，我们可以模拟：

$$|\psi(\tau)\rangle = \frac{e^{-\tau H_{\text{joint}}}|\psi_{\text{init}}\rangle}{\|e^{-\tau H_{\text{joint}}}|\psi_{\text{init}}\rangle\|_2} \tag{3.3}$$

若初始态为 $|\psi_{\text{init}}\rangle = |\psi_0'\rangle \otimes |\phi\rangle$（其中 $|\phi\rangle$ 任意），当 $\tau \to \infty$ 时，稳态将趋近于 $|\psi_0'\rangle \otimes |u_{\min}\rangle$. 然而，我们需要的是 $A^{-1}|b\rangle$，而非 A 的基础态. 为此，必须调整哈密顿量构造.

考虑定义修正的联合哈密顿量：

$$\tilde{H}_{\text{joint}} = H' \otimes I + I \otimes A^{-1} \tag{3.4}$$

但直接实现 A^{-1} 的哈密顿量似乎也不太现实. 这里，我们不妨利用线性组合式哈密顿量模拟（Hamiltonian Simulation via Linear Combination）：

$$\tilde{H}_{\text{joint}} = H' \otimes I + \gamma (I \otimes A)^{-1} \tag{3.5}$$

其中 γ 为调节参数. 此时通过 Trotter-Suzuki 分解 [4]，我们即可近似实现：

$$e^{-i\tilde{H}_{\text{joint}}t} \approx \left(e^{-iH'\otimes I \cdot t/n} e^{-i\gamma (I\otimes A)^{-1}\cdot t/n}\right)^n \tag{3.6}$$

但 $(I \otimes A)^{-1}$ 的模拟仍需通过 HHL 的子过程，从而将导致循环依赖的出现.

2. 可行的优化方案：直接对角化与量子奇异值变换（QSV）

受 Dervovic D. 等人的书第四章内容 [2] 启发，我们可以利用量子奇异值变换（QSV）同时对 H' 和 A 进行对角化！设 A 的奇异值分解为 $A = \sum_j \sigma_j |u_j\rangle\langle v_j|$，构造块编码（Block Encoding）：

$$U_A = \begin{pmatrix} A/\alpha & \cdot \\ \cdot & \cdot \end{pmatrix} \tag{3.7}$$

其中 $\alpha \geq \|A\|$. 类似地，对 H' 构造 $U_{H'}$。通过 QSV 的多项式变换，设计 $P(A) \approx A^{-1}$ 和 $P(H') \approx |\psi_0'\rangle\langle\psi_0'|$，使得联合操作：

$$U_{\text{joint}} = U_{H'}^{\dagger} U_A^{-1} \tag{3.8}$$

作用于初始态 $|0\rangle|\phi\rangle$ 时，输出态为 $|0\rangle A^{-1}|b\rangle + |\text{err}\rangle$. 具体步骤为：

1. 制备 $|b\rangle$ 的近似态：利用 QSV 多项式 $P(H')$ 逼近投影算子 $|\psi_0'\rangle\langle\psi_0'|$，使得：

$$P(H')|\phi\rangle \approx |\psi_0'\rangle\langle\psi_0'|\phi\rangle \propto |b\rangle \tag{3.9}$$

2. 应用 A^{-1} 的近似：设计另一多项式 $Q(A) \approx A^{-1}$，满足 $Q(\sigma_j) \approx \sigma_j^{-1}$. 通过 QSV 实现：

$$Q(A)|b\rangle \approx \sum_j \sigma_j^{-1} \beta_j |u_j\rangle \propto |x\rangle \tag{3.10}$$

那么，此时的总误差将由以下两部分组成：

$$\||\tilde{x}\rangle - |x\rangle\|_2 \leq \|P(H') - |\psi_0'\rangle\langle\psi_0'|\| \cdot \|Q(A)\| + \|Q(A) - A^{-1}\| \cdot \||b\rangle\|_2 \tag{3.11}$$

若 $P(H')$ 和 $Q(A)$ 分别为 ϵ_1 和 ϵ_2 近似，则总误差为 $O(\epsilon_1 + \epsilon_2)$.

3. 复杂度讨论

传统 HHL 状态制备复杂度 T_B，矩阵求逆复杂度 $O(\kappa^2 s^2 \log N/\epsilon)$. 而联合 QSV 方法满足：若 H' 和 A 的块编码复杂度为 $O(s')$，则总复杂度取决于多项式次数 d（与 κ、ϵ 相关）. 对于条件数 κ，需 $d = O(\kappa \log(1/\epsilon))$，整体复杂度可能优于分步处理.

综上，通过联合哈密顿量对角化或量子奇异值变换，有望将状态制备与矩阵求逆融合为单一量子过程，但需权衡近似精度与资源开销. 这一方向为减少 HHL 预处理步骤提供了新思路，值得进一步研究.

四 数值计算

作为一个例子，我们来求解 https : //qiskit.org/textbook/ch − applications/hhl_tutorial.html [4] 中给出的简单问题：

$$A\boldsymbol{x} = \boldsymbol{b}; \qquad A = \begin{pmatrix} 1 & -1/3 \\ -1/3 & 1 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{4.1}$$

1. 理论推导

先来从理论上预测算法的运行结果. 不难求出 A 的特征值和相应特征矢量为：

$$\lambda_1 = \frac{2}{3}, \quad \lambda_2 = \frac{4}{3}; \qquad |u_1\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad |u_2\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -1 \end{pmatrix} \tag{4.2}$$

从而 $|b\rangle$ 在该基下的表示是：

$$|b\rangle = |0\rangle = \frac{1}{\sqrt{2}}\left(|u_1\rangle + |u_2\rangle\right) \tag{4.3}$$

不过需要注意的是：**实现 HHL 算法不必预先知道这些信息**.

我们采用 $n_b = 1$ 来寄存 $|b\rangle$，$n_l = 2$ 来寄存 A 特征值的二进制表示. 若选取时间步长 $t = 2\pi \cdot \frac{3}{8}$，则 QPE 给出的估计为：

$$\frac{\lambda_1 t}{2\pi} = \frac{1}{4}, \quad \frac{\lambda_2 t}{2\pi} = \frac{1}{2} \quad \Longrightarrow \quad |\widetilde{\lambda_1}\rangle = |01\rangle_{n_l}, \quad |\widetilde{\lambda_2}\rangle = |10\rangle_{n_l} \tag{4.4}$$

从而算法的第一步给出：

$$|0\rangle^{\otimes n_l}|b\rangle \xrightarrow{QPE} \frac{1}{\sqrt{2}}|01\rangle|u_1\rangle + \frac{1}{\sqrt{2}}|10\rangle|u_2\rangle \tag{4.5}$$

随后，我们进行 $C = \frac{1}{8}$ 的受控旋转，得到：

$$\frac{1}{\sqrt{2}}|01\rangle|u_1\rangle\left(\sqrt{1-\frac{1}{4}}|0\rangle + \frac{1}{2}|1\rangle\right) + \frac{1}{\sqrt{2}}|10\rangle|u_2\rangle\left(\sqrt{1-\frac{1}{16}}|0\rangle + \frac{1}{4}|1\rangle\right) \tag{4.6}$$

最后再作用 $QPE^{\dagger}$，得到：

$$|00\rangle \otimes \left[\frac{1}{\sqrt{2}}|u_1\rangle\left(\sqrt{\frac{3}{4}}|0\rangle + \frac{1}{2}|1\rangle\right) + \frac{1}{\sqrt{2}}|u_2\rangle\left(\sqrt{\frac{15}{16}}|0\rangle + \frac{1}{4}|1\rangle\right)\right] \tag{4.7}$$

对辅助量子比特进行测量，得到的态为 $|1\rangle$ 的概率为：

$$p_1 = \left(\frac{1}{2\sqrt{2}}\right)^2 + \left(\frac{1}{4\sqrt{2}}\right)^2 = \frac{5}{32} \tag{4.8}$$

此时测量后的量子态变为了：

$$|00\rangle \otimes \frac{\frac{1}{2\sqrt{2}}|u_1\rangle| + \frac{1}{4\sqrt{2}}|u_2\rangle}{\sqrt{5/32}} \otimes |1\rangle \tag{4.9}$$

很容易用初等方法求得原方程的解为 $|x\rangle = (9/8,\ 3/8)^T = \frac{3}{4\sqrt{2}}(2|u_1\rangle + |u_2\rangle)$，$\| x \| = 3\sqrt{\frac{5}{32}}$，从而测量后得到的态正可以写为：

$$|00\rangle_{n_l} \otimes \frac{|x\rangle}{\|x\|} \otimes |1\rangle_{n_b} \tag{4.10}$$

2. 代码实现

我们来编程实现上述算法. 作为参照，首先采用经典方法来求解该方程组：

```
import numpy as np

matrix = np.array([[1, -1/3], [-1/3, 1]])
vector = np.array([1, 0])
classical_solution = NumPyLinearSolver().solve(matrix,
                                               vector/np.linalg.norm(vector))
print('classical state:', classical_solution.state)
```

结果如下：

```
classical state: [1.125 0.375]
```

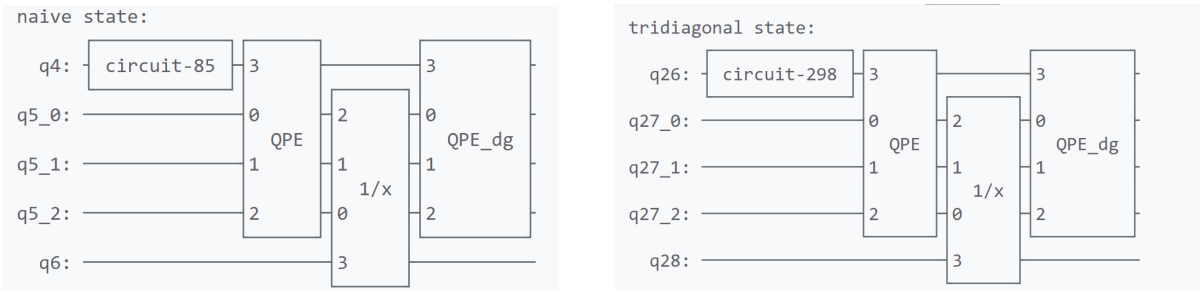
图 2. 经典算法的求解结果.

再来看 HHL 算法的结果. 这里我们将使用 https : //github.com/anedumla/quantum_linear_solvers 提供的包 *linear_solvers*，它针对 HLL 算法的实现定义了一系列模块. 这里 HHL 的算法有两种执行模式，一为默认的求解普通矩阵方程的方法，一为利用 TridiagonalToeplitz 类求解三对角矩阵方程的方法（我们选取的示例恰好满足这个要求）. 我们分别画出两种方法的线路图:

```
from linear_solvers import NumPyLinearSolver, HHL
from linear_solvers.matrices.tridiagonal_toeplitz import TridiagonalToeplitz
tridi_matrix = TridiagonalToeplitz(1, 1, -1 / 3)
tridi_solution = HHL().solve(tridi_matrix, vector)

naive_hhl_solution = HHL().solve(matrix, vector)
print('naive state:')
print(naive_hhl_solution.state)
print('tridiagonal state:')
print(tridi_solution.state)
```

结果如下：



(a) naive state *(b) tridiagonal state*

图 3. 算法的线路图

接下来计算解向量的 Euclidean 范数，并逐个分量地来比较解向量（已做归一化）：

```
print('classical Euclidean norm:', classical_solution.euclidean_norm)
print('naive Euclidean norm:', naive_hhl_solution.euclidean_norm)
print('tridiagonal Euclidean norm:', tridi_solution.euclidean_norm)
```

```
from qiskit.quantum_info import Statevector

naive_sv = Statevector(naive_hhl_solution.state).data
tridi_sv = Statevector(tridi_solution.state).data
naive_full_vector = np.array([naive_sv[16], naive_sv[17]])
tridi_full_vector = np.array([tridi_sv[16], tridi_sv[17]])

def get_solution_vector(solution):
    """Extracts and normalizes simulated state vector
    from LinearSolverResult."""
    solution_vector = Statevector(solution.state).data[16:18].real
    norm = solution.euclidean_norm
    return norm * solution_vector / np.linalg.norm(solution_vector)

print('full naive solution vector:', get_solution_vector(naive_hhl_solution))
print('full tridi solution vector:', get_solution_vector(tridi_solution))
print('classical state:', classical_solution.state)
```

结果如下：

```
classical Euclidean norm: 1.1858541225631423
naive Euclidean norm: 1.18585412256314
tridiagonal Euclidean norm: 1.185854122563139
```

图 4. 解向量范数的结果对比.

```
full naive solution vector: [1.125 0.375]
full tridi solution vector: [1.125 0.375]
classical state: [1.125 0.375]
```

图 5. 解向量各分量的对比.

默认方法得到解是精确的，这并不奇怪；然而 TridiagonalToeplitz 仅能给出 2×2 矩阵系统的精确解，对于更大的系统，它给出的将是近似值.

随着系统规模的增加，默认精确算法所需的量子比特数呈指数增长，而三对角矩阵方法所需的比特数仅仅呈多项式增长. 作为验证，我们来比较不同系统规模下两种方法的量子电路层数：

```
from scipy.sparse import diags
from qiskit import transpile

MAX_QUBITS = 4
a = 1
b = -1/3
i = 1
# calculate the circuit depths for different number of qubits to compare the use
# of resources (WARNING: This will take a while to execute)
naive_depths = []
tridi_depths = []
for n_qubits in range(1, MAX_QUBITS+1):
    matrix = diags([b, a, b],
                   [-1, 0, 1],
                   shape=(2**n_qubits, 2**n_qubits)).toarray()
```

```
    vector = np.array([1] + [0]*(2**n_qubits -1))

    naive_hhl_solution = HHL().solve(matrix, vector)
    tridi_matrix = TridiagonalToeplitz(n_qubits, a, b)
    tridi_solution = HHL().solve(tridi_matrix, vector)

    naive_qc = transpile(naive_hhl_solution.state,
                         basis_gates=['id', 'rz', 'sx', 'x', 'cx'])
    tridi_qc = transpile(tridi_solution.state,
                         basis_gates=['id', 'rz', 'sx', 'x', 'cx'])

    naive_depths.append(naive_qc.depth())
    tridi_depths.append(tridi_qc.depth())
    i +=1
```

```
sizes = [f"{2**n_qubits}×{2**n_qubits}"
        for n_qubits in range(1, MAX_QUBITS+1)]
columns = ['size of the system',
           'quantum_solution depth',
           'tridi_solution depth']
data = np.array([sizes, naive_depths, tridi_depths])
ROW_FORMAT ="{:>23}" * (len(columns) + 2)
for team, row in zip(columns, data):
    print(ROW_FORMAT.format(team, *row))

print('excess:',
      [naive_depths[i] - tridi_depths[i] for i in range(0, len(naive_depths))])
```

结果如下：

```
     size of the system                2×2                 4×4                 8×8                16×16
excess: [-231, -2530, 19253, 355205]
 quantum_solution depth                334                2577               34009               401757
excess: [-231, -2530, 19253, 355205]
   tridi_solution depth                565                5107               14756                46552
excess: [-231, -2530, 19253, 355205]
```

图 6. 两种方法的量子线路层数随系统规模变化的对比.

回到我们求解方程的主题. 还可以尝试计算解向量的平均分量：

```
from linear_solvers.observables import AbsoluteAverage, MatrixFunctional

NUM_QUBITS = 1
MATRIX_SIZE = 2 ** NUM_QUBITS
# entries of the tridiagonal Toeplitz symmetric matrix
a = 1
b = -1/3

matrix = diags([b, a, b],
               [-1, 0, 1],
               shape=(MATRIX_SIZE, MATRIX_SIZE)).toarray()
vector = np.array([1] + [0]*(MATRIX_SIZE - 1))
tridi_matrix = TridiagonalToeplitz(1, a, b)

average_solution = HHL().solve(tridi_matrix,
```

```
                                vector,
                                AbsoluteAverage())
classical_average = NumPyLinearSolver(
                        ).solve(matrix,
                                 vector / np.linalg.norm(vector),
                                 AbsoluteAverage())

print('quantum average:', average_solution.observable)
print('classical average:', classical_average.observable)
print('quantum circuit results:', average_solution.circuit_results)
```

结果如下：

```
quantum average: 0.7499999999999976
classical average: 0.75
quantum circuit results: (0.4999999999999706+0j)
```

图 7. 解向量平均分量的对比.

还可以提取解向量的更多特征供 HHL 算法和经典算法展开对比，由于空间所限就不一一详述了.

3. 拓展讨论

本人独立地撰写了一套基于 IBM Qiskit 的程序用于实现 HHL 算法，没有额外调用其他平台的包. 所有函数的定义都直观地呈现在了代码中，可以让我们清晰地看到算法的逻辑和步骤.

这里我们不再使用 github 上的例子，而是重新考虑一个简单的问题：

$$A\boldsymbol{x} = \boldsymbol{b}; \qquad A = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 0.8 \\ 0.6 \end{pmatrix} \tag{4.11}$$

与 github 上例子不同的是，这里的相位估计无法做到绝对精确，因此运行结果的准确性高度依赖于参数 C 和 n_l 的选取. 我们通过在不同参数区域妥善选取参数值调试代码，已成功验证 HHL 算法的准确性和有效性，并指出默认算法中进行精确的受控旋转的确需要指数级资源. 作为一个例子，我们选取参数为 $C = 0.01$，$dt = \pi/8$，$n_l = 10$ 来执行计算过程：

```
import numpy as np
from scipy.linalg import expm
from qiskit import QuantumCircuit, transpile
from qiskit.circuit.library import RYGate
from qiskit.providers.basic_provider import BasicSimulator
from qiskit.quantum_info import Operator
import matplotlib.pyplot as plt
import time

def qft(n_qubits):
    circuit = QuantumCircuit(n_qubits)
    circuit.h(range(n_qubits))                  # 初始化钟表的指针
    for i in reversed(range(n_qubits)):
        circuit.barrier(range(n_qubits))
        for j in reversed(range(i)):
            circuit.crx(np.pi / 2 ** (i - j),
                        j,                      # 控制位bit
                        i)                      # 目标位bit
    return circuit
```

```
def qft_dagger(n_qubits):
    return qft(n_qubits).inverse()

def controlled_U(n_qubits, A, dt):
    circuit = QuantumCircuit(n_qubits + 1)
    U = np.matrix(expm(1j * A * dt))
    for i in range(n_qubits):
        sub_circuit = QuantumCircuit(1)
        sub_circuit.name = '$U^{2^' + str(i) + '}$'
        sub_circuit.append(Operator(U ** (2 ** i)), [0])
        sub_gate = sub_circuit.to_gate().control()
        circuit.append(sub_gate, [n_qubits - i - 1, n_qubits])
    return circuit


def controlled_R(n_qubits, C, dt):
    circuit = QuantumCircuit(n_qubits + 1)
    for k in range(1, 2 ** n_qubits):
        for i in range(n_qubits):
            if (k & (1 << i)) == 0:
                circuit.x(1 + i)
        phi = k / (2 ** n_qubits)
        if k > 2 ** (n_qubits - 1):
            phi -= 1.0
        l = 2 * np.pi / dt * phi
        circuit.append(RYGate(2 * np.arcsin(C / l)).control(n_qubits),
                       [(i + 1) % (n_qubits + 1) for i in range(n_qubits + 1)])
        for i in range(n_qubits):
            if (k & (1 << i)) == 0:
                circuit.x(1 + i)
        circuit.barrier(range(n_qubits + 1))
    return circuit

if __name__ == "__main__":
    import scipy.linalg

    dt = np.pi / 8     # time step in e^iHdt
    C = 0.01           # constant in controlled-R
    n_qubits = 10
    A = np.matrix([
        [1,  0],
        [0, -1]
    ])

    # 初始化
    qc = QuantumCircuit(1 + n_qubits + 1, 2)
    qc.ry(2 * np.arcsin(0.6), n_qubits + 1)

    # 量子相位估计
    qc.barrier(range(1 + n_qubits + 1))
    qc.h(range(1, n_qubits + 1))
    qc.append(controlled_U(n_qubits, A, dt), range(1, 1 + n_qubits + 1))
```

```
    qc.append(qft_dagger(n_qubits),range(1, n_qubits + 1))
    qc.barrier(range(1 + n_qubits + 1))

    # 受控旋转
    qc.append(controlled_R(n_qubits, C, dt), range(n_qubits + 1))

    # 逆量子相位估计
    qc.barrier(range(1 + n_qubits + 1))
    qc.append(qft(n_qubits),range(1, n_qubits + 1))
    qc.append(controlled_U(n_qubits, A, dt).inverse(), range(1, 1 + n_qubits + 1))
    qc.h(range(1, n_qubits + 1))
    qc.barrier(range(1 + n_qubits + 1))

    # 测量
    qc.measure(0, 0)
    qc.measure(1 + n_qubits, 1)

    # 绘制电路图
    decomposed_qc = qc.decompose()
    fig = decomposed_qc.draw('mpl', fold=-1, scale=0.8)
    plt.title(f"HHL Quantum Circuit ({n_qubits} Clock Qubits)", fontsize=14)
    plt.savefig('HHL_detailed.png', bbox_inches='tight', dpi=300)
    print("电路图已保存为 HHL_detailed.png")

    # 运行结果
    print("运行量子模拟:")
    start_time = time.time()
    backend = BasicSimulator()
    t_qc = transpile(qc, backend)
    result = backend.run(t_qc, shots=2 ** 21).result()

    end_time = time.time()
    elapsed_time = end_time - start_time
    counts = result.get_counts(t_qc)
    print(counts)
    count_01 = counts.get('01', 0)
    count_11 = counts.get('11', 0)
    total = count_01 + count_11

if total > 0:
    Prob_01 = count_01 / total
    Prob_11 = count_11 / total
    amplitude_01 = np.sqrt(Prob_01)
    amplitude_11 = np.sqrt(Prob_11)
    print("Prob_01:", Prob_01)
    print("Prob_11:", Prob_11)
    print(f"x=({amplitude_01}, {amplitude_11})")
else:
    print("未测量到有效解!")

print(f"运行时间:{elapsed_time:.2f}秒")
```

算法具体到门的示意图如下（简洁起见，这里展示 $n_l = 2$ 情形）：

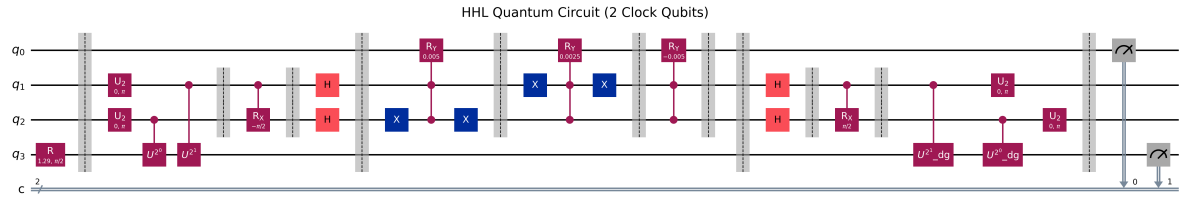


图 8. *HHL* 算法的量子线路图.

运行结果为：

```
运行量子模拟:
{'10': 750492, '00': 1335660, '01': 7037, '11': 3963}
Prob_01: 0.6397272727272727
Prob_11: 0.3602727272727273
x=(0.7998295272914552, 0.6002272296994925)
运行时间: 32.07 秒
```

图 9. *HHL* 算法的运行结果.

可见 $n_l = 10$ 时给出的解向量分量绝对值相当精确.

不过需要注意的是，我们的程序无法判断分量的正负号. 这是因为我们实际测量的是 $|b\rangle$ 寄存器的输出中分别得到 $|0\rangle$ 和 $|1\rangle$ 的概率，也就是 $|x\rangle$ 分量模长的平方，从而无法直接判断 $|0\rangle$ 和 $|1\rangle$ 的相对相位. 可见根本原因仍在于 HHL 算法输出的解处于量子态，我们只能通过经典观测来了解它，这个过程中不可避免地损失了一些信息.

想要确定 $|x\rangle$ 的完整信息，我们可以在测量后对输出的量子态进行进一步的处理，比如引入多个不同可观测量 M_i 计算在解上的期望值 $\langle x|M_i|x\rangle$ 等，但这会增加额外的计算复杂度和资源消耗.

五 总结与展望

HHL 算法在量子数值计算等方面有着广泛的应用价值, 但是目前仍然处在抽象的算法分析与描述阶段. 使用 IBM qiskit 量子计算等平台，可以对 HHL 量子算法的通用量子线路进行设计和仿真. 我们期待未来能有更多关于 HHL 算法理论优化或物理实现的研究，推动量子计算在实际问题求解中的应用.

参考文献

[1] Harrow, A. W., Hassidim, A., & Lloyd, S. (2009). Quantum algorithm for solving linear systems of equations. Physical Review Letters, 103(15), 150502.

[2] Dervovic, D., Herbster, M., Mountney, P., Severini, S., Usher, N., & Wossnig, L. (2018). Quantum linear systems algorithms: a primer. arXiv preprint arXiv:1802.08227.

[3] Mickael A. Nielsen, & Isaac L. Chuang. (2010). Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press.

[4] Grover, L., & Rudolph, T. (2002). Creating superpositions that correspond to effi ciently integrable probability distributions. eprint: arXiv:quantph/0208112v1.

[5] Qiskit Community, HHL Tutorial —Solving Linear Systems of Equations using Quantum Computing.Qiskit Textbook (GitHub), `https://qiskit.org/textbook/ch-applications/hhl_tutorial.html`.