arXiv:2602.05255v1 [cond-mat.str-el] 5 Feb 2026

lrux：在 JAX 中快速进行行列式和 Pfaffian 的低秩更新

Ao Chen[1,2,*] 和 Christopher Roth[2]
[1] *Division of Chemistry and Chemical Engineering, California Institute of Technology, Pasadena, California 91125, USA*
[2] *Center for Computational Quantum Physics, Flatiron Institute, New York 10010, USA*

我们介绍 `lrux`，这是一款基于 JAX 的软件包，用于快速进行行列式和 Pfaffian 的低秩更新，旨在解决各种量子蒙特卡罗（QMC）算法中的主要计算瓶颈。该软件包实现了高效的低秩更新，将连续波函数求值的成本从 $\mathcal{O}(n^3)$ 降至 $\mathcal{O}(n^2k)$，前提是更新秩 k 小于矩阵的维数 n。软件包支持行列式和 Pfaffian 更新，还提供延迟更新策略，以浮点运算换取现代加速器上更低的内存流量。`lrux` 原生支持 JAX 转换功能，例如 JIT编译、向量化和自动微分，并支持实数和复数数据类型。GPU 基准测试表明，在矩阵较大时，性能最高可提升 $1000\times$ 。`lrux` 能够高效、可扩展地求值反对称波函数，并设计为可直接替换的组件，适用于各种 QMC 工作流程。`lrux` 可在此处获取：https://github.com/ChenAo-Phys/lrux。

I. 引言

量子蒙特卡罗（QMC）方法是模拟无法精确求解的相互作用电子系统的最有力方法之一。它们在准确模拟大型分子的化学性质，以及理解关联材料中电子的集体行为方面发挥着核心作用。量子蒙特卡罗算法种类繁多，包括扩散蒙特卡罗（DMC）等投影方法 [1, 2], 辅助场量子蒙特卡罗（AFQMC） [3–5], 以及变分方法 [6, 7], ，其中包括近期提出的神经量子态（NQS） [8]。尽管算法各不相同，所有量子蒙特卡罗方法都依赖于反复从多电子波函数中采样，

$$\psi(\mathbf{x}_1, \mathbf{x}_2 .... \mathbf{x}_n), \tag{1}$$

或从波函数之间的比值或重叠等密切相关的量中采样。

电子的一个定义性特征是其在粒子交换下具有费米子反对称性， $\psi(\mathbf{x}_1, \mathbf{x}_2) = -\psi(\mathbf{x}_2, \mathbf{x}_1)$，这意味着多电子波函数按照对称群的交替表示进行变换。对于无相互作用电子，最简表示是斯莱特行列式，其计算复杂度为 $\mathcal{O}(n^3)$，适用于 n 个电子。另一种更一般的反对称形式称为 Pfaffian，它能自然地表示配对相 [9–11], ，其缩放关系同样为 $\mathcal{O}(n^3)$，但通常具有更大的前置因子。因此，高效计算行列式和 Pfaffian 是将量子蒙特卡罗计算扩展到大型系统时的关键瓶颈。

在许多量子蒙特卡罗算法中，相继的构型仅在少量轨道的占据或位置上有所不同。在这种情况下，行列式和 Pfaffian 可以使用低秩更新（LRU）技术进行更新，从而将计算成本降至 $\mathcal{O}(n^2k)$，其中 k 为更新的秩。因此，此类更新可为蒙特卡罗采样过程中波函数比值和重叠的计算带来大幅加速。此外，只要轨道的神经网络回流变换本身允许低秩表示，LRU 便可应用于费米子神经量子态 [10–16] （如 [10–12] 中所做的那样）。在这种情况下，有效更新秩由与轨道变化和回流变换相关的秩之和给出。

在本工作中，我们介绍 lrux，这是一个基于 JAX 的软件包，实现了行列式和 Pfaffian 的低秩更新，旨在无缝集成到广泛的量子蒙特卡罗算法中。JAX [17] 支持自动并行化和即时（JIT）编译，使 lrux 能够高效利用现代 GPU 架构，其中稠密线性代数运算可比 CPU 实现实现数量级的加速。此外，lrux 支持延迟更新，以增加浮点运算为代价换取减少内存搬移，从而允许用户针对其硬件优化性能。

`lrux` 可通过以下地址获取：https://github.com/ChenAo-Phys/lrux，并可使用以下命令通过 pip 安装 lrux。撰写本文时，当前版本为 0.1.2，兼容 `jax>=0.4.4`。

* chenao.phys@gmail.com

我们提供了一些示例，展示如何在实际的量子蒙特卡罗工作流程中使用 lrux，并介绍了推荐的环境设置和依赖项。特别建议启用双精度算术，以减少数值误差累积，并确保大规模模拟的稳定性。本文中的所有示例代码都应在开头添加以下代码。

```
导入 jax

# 在 JAX 中启用双精度jax.config.update("jax_enable_
x64",True)

导入 jax.numpy as jnp导入
jax.random as jr导入 lrux


键  =jr.key(42)

定义 get_key():""" 用于生成 jax 密钥的函数。不适用
于一般用途。 """全局 KEYKEY,new_key  =jr.split
(KEY,2)返回 new_key
```

II. 行列式

A. 行列式的低秩更新

考虑对 $n \times n$ 矩阵 $\mathbf{A}_0 \to \mathbf{A}_1 \to \ldots \to \mathbf{A}_t \to \ldots$进行连续的低秩更新。每一步的低秩更新都可以表示为 $\mathbf{A}_t = \mathbf{A}_{t-1} + \mathbf{v}_t \mathbf{u}_t^T$，其中 $\mathbf{u}_t$ 和 $\mathbf{v}_t$ 是满足 $k \ll n$ 的 $n \times k$ 矩阵。利用矩阵行列式引理，我们有

$$\det(\mathbf{A}_t) = \det(\mathbf{A}_{t-1} + \mathbf{v}_t \mathbf{u}_t^T) = \det(\mathbf{A}_{t-1}) \det(\mathbf{1} + \mathbf{u}_t^T \mathbf{A}_{t-1}^{-1} \mathbf{v}_t), \tag{2}$$

两个行列式的比值为

$$r_t = \frac{\det \mathbf{A}_t}{\det \mathbf{A}_{t-1}} = \det \mathbf{R}_t, \tag{3}$$

其中

$$\mathbf{R}_t = \mathbf{1} + \mathbf{u}_t^T \mathbf{A}_{t-1}^{-1} \mathbf{v}_t \tag{4}$$

是一个 $k \times k$ 矩阵。若 $\det \mathbf{A}_{t-1}$ 和 $\mathbf{A}_{t-1}^{-1}$ 已计算并存储在内存中，则可以以较低的复杂度 $\mathcal{O}(n^2 k)$计算 $\det \mathbf{A}_1$，而非使用原始的行列式计算复杂度 $\mathcal{O}(n^3)$。此处提供一段示例代码，用于计算秩为 1（k=1）的更新。

```
n  = 4k  = 1A  =jr.normal(get_key(),(n,
n))Ainv  =jnp.linalg.inv(A)detA  =jnp.
linalg.det(A)# 随机更新u  =jr.normal(get_key
(),(n,k))v  =jr.normal(get_key(),(n,k))r  =
lrux.det_lru(Ainv,u,v)detA  *=rdet_exact  =
jnp.linalg.det(A  +v@u.T)assert jnp.isclose
(detA,det_exact)
```

在许多应用中，例如在蒙特卡洛链中持续计算波函数的比值，需要连续执行多次低秩更新。这要求跟踪矩阵逆 $\mathbf{A}_t^{-1}$，它可以通过 Sherman–Morrison 公式进行更新，

$$\mathbf{A}_t^{-1} = (\mathbf{A}_{t-1} + \mathbf{v}_t \mathbf{u}_t^T)^{-1} = \mathbf{A}_{t-1}^{-1} - \mathbf{A}_{t-1}^{-1} \mathbf{v}_t \mathbf{R}_t^{-1} \mathbf{u}_t^T \mathbf{A}_{t-1}^{-1}, \tag{5}$$

其复杂度为 $\mathcal{O}(n^2 k)$，而从头计算逆矩阵需要 $\mathcal{O}(n^3)$。通过反复计算波函数之间的比值并更新存储的逆矩阵，可以连续进行低秩更新。此算法存储 $\mathbf{A}_t^{-1}$ 的内存复杂度为 $\mathcal{O}(n^2)$。下面展示连续更新的示例代码，仍以秩一行更新为例。

```
n = 4
k = 1

一个 =jr.正态分布（get_key(), (n, n)）Ainv =
jnp.linalg.inv(A) detA =jnp.linalg.det(A)


det_lru_fn =jax.即时编译(lrux.det_lru, static_argnums=3, donate_argnums=0)

# 随机更新
u1 = jr.正态分布(get_key(), (n, k))
v1 = jr.正态分布(get_key(), (n, k))
r, Ainv =det_lru_函数（Ainv, u1, v1, 返回_更新=True)
detA *=r

# 随机更新
u2 =jr.正态分布(get_key(), (n, k))
v2 = jr.正态分布(get_key(), (n, k))
r, Ainv =det_lru_函数(Ainv, u2, v2, return_更新=True)
detA *=r

行列式_精确 =jnp.linalg.det(A +v1@u1.T +v2@u2.T)断言 jnp.
isclose(detA, det_精确)
```

公式（3）的数值稳定性依赖于 $\mathbf{A}_{t-1}^{-1}$ 的非奇异性，即 $\det\mathbf{A}_{t-1} \neq 0$；否则无法计算该比值$r_t$。同样，公式（5）要求 $\mathbf{R}_t$ 和 $\det\mathbf{R}_t \neq 0$ 均为非奇异，否则 $\mathbf{A}_t$ 将是奇异且不可逆的。因此，我们建议用户启用双精度，以避免实践中偶发的数值（近）奇异性。

B. 矩阵更新

下面我们将展示，包括量子蒙特卡罗中常见的单行和单列更新操作在内的许多矩阵更新，都可以写成统一的低秩表达式 $\mathbf{A}_1 - \mathbf{A}_0 = \mathbf{v}\mathbf{u}^T$。这里列出低秩更新的几个特殊情况，以及在 lrux 中用于表示 $\mathbf{u}$ 和$\mathbf{v}$ 的相应代码。

- **秩一行更新：**

$$\mathbf{A}_1 - \mathbf{A}_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ u_0 & u_1 & u_2 & u_3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} (u_0, u_1, u_2, u_3) = \mathbf{v}\mathbf{u}^T. \tag{6}$$

```
u =jnp.数组([u0, u1, u2, u3])
v = 1# 位置 1 处非零的独热向量
```

- **秩一列更新：**

$$\mathbf{A}_1 - \mathbf{A}_0 = \begin{pmatrix} 0 & 0 & v_0 & 0 \\ 0 & 0 & v_1 & 0 \\ 0 & 0 & v_2 & 0 \\ 0 & 0 & v_3 & 0 \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} (0, 0, 1, 0) = \mathbf{v}\mathbf{u}^T. \tag{7}$$

```
u = 2# 在位置 2 非零的独热向量v =jnp.array([v0,v1,v2,v
3])
```

- **秩-2 行更新：**

$$\mathbf{A}_1 - \mathbf{A}_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ u_{00} & u_{01} & u_{02} & u_{03} \\ 0 & 0 & 0 & 0 \\ u_{10} & u_{11} & u_{12} & u_{13} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} u_{00} & u_{01} & u_{02} & u_{03} \\ u_{10} & u_{11} & u_{12} & u_{13} \end{pmatrix} = \mathbf{v}\mathbf{u}^T. \tag{8}$$

```
u =jnp.数组([[u00,u01,u02,u03], [u10,u11,u12,u13]).T

# 两个拼接的独热向量在位置 1 和 3 处非零v =jnp.array([1, 3])
```

- **行列同时更新：**

$$\mathbf{A}_1 - \mathbf{A}_0 = \begin{pmatrix} 0 & 0 & v_0 & 0 \\ u_0 & u_1 & u_2 + v_1 & u_3 \\ 0 & 0 & v_2 & 0 \\ 0 & 0 & v_3 & 0 \end{pmatrix} = \begin{pmatrix} 0 & v_0 \\ 1 & v_1 \\ 0 & v_2 \\ 0 & v_3 \end{pmatrix} \begin{pmatrix} u_0 & u_1 & u_2 & u_3 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \mathbf{v}\mathbf{u}^T. \tag{9}$$

```
xu =jnp.array([u0,u1,u2,u3])eu = 2# 稠密表示与独热表示的拼
接u =(xu,eu)xv =jnp.array([v0,v1,v2,v3])ev = 1# 稠密表
示与独热表示的拼接v =(xv,ev)
```

我们建议用户尽可能使用独热表示来表示 $\mathbf{u}$和 $\mathbf{v}$ ，因为这有时能大幅加快计算。例如，计算 $\mathbf{u}^T\mathbf{M}\mathbf{v}$ 时，如果 $\mathbf{u}$ 和 $\mathbf{v}$ 如公式（6）所示，那么朴素的稠密运算 `u @ M @ v dense` 的 $\mathcal{O}(n^2)$复杂度为，而独热运算`u @ M[:, v onehot]` 的复杂度仅为 $\mathcal{O}(n)$。

C. 延迟更新

LRU 的时间瓶颈来自 $\mathbf{A}_{t-1}^{-1}\mathbf{v}_t\mathbf{R}_t^{-1}\mathbf{u}_t^T\mathbf{A}_{t-1}^{-1}$ 方程（5）中的计算，这不可避免地涉及 $n \times k$ 和 $k \times n$ 矩阵的矩阵乘积。虽然浮点运算复杂度 $\mathcal{O}(n^2k$与算法的其他部分相同，但对于较小的 k，计算通常受内存带宽限制。在这种情况下，可以采用延迟更新策略来加速计算。在每个时间步 t，我们定义

$$\mathbf{a}_t = \mathbf{A}_{t-1}^{-1}\mathbf{v}_t, \tag{10}$$

$$\mathbf{b}_t = (\mathbf{A}_{t-1}^{-1})^T\mathbf{u}_t(\mathbf{R}_t^{-1})^T, \tag{11}$$

使得

$$\mathbf{A}_\tau^{-1} = \mathbf{A}_{\tau-1}^{-1} - \mathbf{a}_\tau\mathbf{b}_\tau^T = \mathbf{A}_0^{-1} - \sum_{t=1}^{\tau}\mathbf{a}_t\mathbf{b}_t^T. \tag{12}$$

因此，无需按照公式（$\mathbf{A}_\tau^{-1}$5）在每一步显式计算，也可以改为存储 $\mathbf{A}_0^{-1}$、$\mathbf{a}_t$和 $\mathbf{b}_t$，这些信息足以重构 $\mathbf{A}_\tau^{-1}$。然后结合公式（4）和公式（12），即可计算$\mathbf{R}_\tau$ 如下：

$$\mathbf{R}_\tau = \mathbf{1} + \mathbf{u}_\tau^T\mathbf{A}_0^{-1}\mathbf{v}_\tau - \sum_{t=1}^{\tau-1}(\mathbf{u}_\tau^T\mathbf{a}_t)(\mathbf{b}_t^T\mathbf{v}_\tau) \tag{13}$$

其 $\mathcal{O}(n^2k + \tau nk^2)$ 复杂度，且避免了内存受限计算 $\mathbf{a}_t\mathbf{b}_t^T$。类似地，可以结合公式（10）、公式（11）和公式（12）得到

$$\mathbf{a}_\tau = \mathbf{A}_0^{-1}\mathbf{v}_\tau - \sum_{t=1}^{\tau-1} \mathbf{a}_t(\mathbf{b}_t^T\mathbf{v}_\tau), \tag{14}$$

$$\mathbf{b}_\tau = \left[(\mathbf{A}_0^{-1})^T\mathbf{u}_\tau - \sum_{t=1}^{\tau-1} \mathbf{b}_t(\mathbf{a}_t^T\mathbf{u}_\tau)\right](\mathbf{R}_\tau^{-1})^T \tag{15}$$

同样具有 $\mathcal{O}(n^2k + \tau nk^2)$ 复杂度，并再次避免了内存受限计算 $\mathbf{a}_t\mathbf{b}_t^T$。存储 $\mathbf{A}_0^{-1}$、$\mathbf{a}_t$ 和 $\mathbf{b}_t$ 所需的内存复杂度为 $\mathcal{O}(n^2 + \tau nk)$。为避免 τ 无限增长，可以设置上界 T，并在满足 $\tau = T$ 时使用公式 (12) 重建完整矩阵 $\mathbf{A}_\tau^{-1}$。然后将此 $\mathbf{A}_\tau^{-1}$ 设为新的 $\mathbf{A}_0^{-1}$，并将 $\mathbf{a}_t$ 和 $\mathbf{b}_t$ 设为 0，以开始新一轮延迟更新。通过控制 $\tau \leq T < n/k$，延迟更新的总体复杂度保持为 $\mathcal{O}(n^2k)$。通常选择 $T \approx n/10k$，但确切的最优值很大程度上取决于硬件和矩阵大小。下面给出了延迟更新的示例代码以供参考，其中设置了 $T = 4$。

```
n = 10max_delay = 4A =jr.normal(get_key(), (n, n))carrier =lrux.init_det_carrier(A, max_delay)
detA =jnp.linalg.det(A)det_lru_fn =jax.jit(lrux.det_lru_delayed, static_argnums=(3, 4), donate_
argnums=0)for i in range(20): current_delay =i % max_delay # 随机更新 u =jr.normal(get_key(), (n, ))
# 选择一个随机行 v =jr.randint(get_key(), shape=(1, ), minval=0, maxval=n)# 独热表示 r, carrier =
det_lru_fn(carrier, u, v, True, current_delay)detA *=r # 验证低秩更新结果 A =A.at[v].add(u)detA_exact
=jnp.linalg.det(A)assert jnp.isclose(detA, detA_exact)
```

III. PFAFFIAN

Pfaffian 将反对称矩阵映射为一个数，该数在交换列或行时具有反对称性，这常用于描述费米子之间的配对。有关 Pfaffian 的定义及其重要性质，请参阅附录 A 。

A. Pfaffian 的低秩更新

考虑两个仅通过低秩更新而不同的 $n \times n$ 反对称矩阵 $\mathbf{A}_t$ 和 $\mathbf{A}_{t-1}$。则有如下通用表达式

$$\mathbf{A}_t - \mathbf{A}_{t-1} = -\mathbf{u}_t\mathbf{J}\mathbf{u}_t^T, \tag{16}$$

其中，$\mathbf{u}$ 是一个 $n \times 2k$ 矩阵，且满足 $k \ll n$，并且

$$\mathbf{J} = \begin{pmatrix} \mathbf{0} & \mathbf{1} \\ -\mathbf{1} & \mathbf{0} \end{pmatrix} \tag{17}$$

是一个 $2k \times 2k$ 斜对称单位矩阵。利用公式（A.8）和$\mathbf{J}^{-1} = -\mathbf{J}$，我们有

$$\mathrm{pf}\mathbf{A}_t = \mathrm{pf}(\mathbf{A}_{t-1} - \mathbf{u}_t\mathbf{J}\mathbf{u}_t^T) = \frac{\mathrm{pf}\mathbf{A}_{t-1}}{\mathrm{pf}\mathbf{J}}\mathrm{pf}(\mathbf{J} + \mathbf{u}_t^T\mathbf{A}_{t-1}\mathbf{u}_t), \tag{18}$$

两个 Pfaffian 之间的比值为

$$r_t = \frac{\mathrm{pf}\mathbf{A}_t}{\mathrm{pf}\mathbf{A}_{t-1}} = \frac{\mathrm{pf}\mathbf{R}_t}{\mathrm{pf}\mathbf{J}}, \tag{19}$$

其中 $\mathrm{pf}\mathbf{J} = (-1)^{k(k-1)/2}$ 可直接由公式（A.6）推导得到，并且

$$\mathbf{R}_t = \mathbf{J} + \mathbf{u}_t^T\mathbf{A}_{t-1}^{-1}\mathbf{u}_t. \tag{20}$$

因此，如果 $\mathrm{pf}\mathbf{A}_t$ 已计算并存储在内存中，则可以用 $\mathcal{O}(n^2k)$ 的时间复杂度计算 $\mathrm{pf}\mathbf{A}_{t-1}$，而不是原始的 Pfaffian 复杂度 $\mathcal{O}(n^3)$。存储$\mathbf{A}_{t-1}^{-1}$所需的内存复杂度为 $\mathcal{O}(n^2)$。下面给出一个 Pfaffian 局部更新的简单示例代码。

```
n  = 4k  = 1A =jr.normal(get_key(),
(n,n))A  = (A -A.T) / 2Ainv =jnp.
linalg.inv(A)pfA =lrux.pf(A)# 随机更新u =
jr.normal(get_key(),(n,2*k))r =lrux.pf_
lru(Ainv,u)pfA *=rJ =lrux.skew_eye(k)pf_
exact =lrux.pf(A-u@J@u.T)assert jnp.
isclose(pfA,pf_exact)
```

要将$\mathbf{A}_{t-1}^{-1}$ 更新为$\mathbf{A}_t^{-1}$，需要利用伍德伯里矩阵恒等式来获 n

$$\mathbf{A}_t^{-1} = (\mathbf{A}_{t-1} - \mathbf{u}_t\mathbf{J}\mathbf{u}_t^T)^{-1} = \mathbf{A}_{t-1}^{-1} + (\mathbf{A}_{t-1}^{-1}\mathbf{u}_t)\mathbf{R}^{-1}(\mathbf{A}_{t-1}^{-1}\mathbf{u}_t)^T \tag{21}$$

其中我们已使用（$\mathbf{A}_{t-1}^{-1}$）$^T = -\mathbf{A}_{t-1}^{-1}$。对$\mathbf{A}^{-1}$ 的此次更新具有 $\mathcal{O}$（n^2k）复杂度。为了在$k = 1$ 情形下更快地计算，可以将 $n \times 2$ 矩阵$\mathbf{u}$ 分解为$\mathbf{u}$ = （$\mathbf{x}$，$\mathbf{y}$），并将式（21）改写为

$$\begin{aligned}\mathbf{A}_t^{-1} &= \mathbf{A}_{t-1}^{-1} + (\mathbf{A}_{t-1}^{-1}\mathbf{x}, \mathbf{A}_{t-1}^{-1}\mathbf{y})\begin{pmatrix} 0 & r \\ -r & 0 \end{pmatrix}^{-1}(\mathbf{A}_{t-1}^{-1}\mathbf{x}, \mathbf{A}_{t-1}^{-1}\mathbf{y})^T \\ &= \mathbf{A}_{t-1}^{-1} + \frac{1}{r}(\mathbf{A}_{t-1}^{-1}\mathbf{y})(\mathbf{A}_{t-1}^{-1}\mathbf{x})^T - \frac{1}{r}(\mathbf{A}_{t-1}^{-1}\mathbf{x})(\mathbf{A}_{t-1}^{-1}\mathbf{y})^T.\end{aligned} \tag{22}$$

带有更新后矩阵逆$\mathbf{A}^{-1}$ 的连续 LRU 的 lrux 代码如下所示 .

```
n = 4
k = 1

A =jr . 正态分布 ( get_key () , (n , n)) A =
(A - A.T) / 2 Ainv =jnp . linalg . inv (A)
pfA =lrux .pf (A)

d et_lru_fn =JAX。即时编译（lrux。Pf_lru，static_argnums =2，donate_argnums =0 ）

# 随机更新
u1 = jr . 正态分布 ( get_key () ,  (n ,  2 * k))
r ,  Ainv =行列式_lru_fn (Ainv , u1 ,  return_更新 =True )
pfA *=r
```

```
# 随机更新u2 =jr。正态分布（get_key（），（n，2 * k））r，
Ainv =行列式_lru_fn（Ainv，u2，return_update =True）PfA *=
r

J =lrux . 反对称_eye(k)pf_精确 =lrux . pf(A-u1@J@u1 .T-u2@J@
u2 .T)断言 jnp . isclose(pfA , pf_精确 )
```

B. 矩阵更新

这里列出了一些在应用中很重要的低秩更新特殊情形，以及用于在 lrux 中表示$\mathbf{u}$ 的相应代码。只要条件允许，仍建议使用向量的独热表示。

- **更新1行和1列**

$$\mathbf{A}_1 - \mathbf{A}_0 = \begin{pmatrix} 0 & -u_0 & 0 & 0 \\ u_0 & 0 & u_2 & u_3 \\ 0 & -u_2 & 0 & 0 \\ 0 & -u_3 & 0 & 0 \end{pmatrix} = - \begin{pmatrix} u_0 & 0 \\ u_1 & 1 \\ u_2 & 0 \\ u_3 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} u_0 & u_1 & u_2 & u_3 \\ 0 & 1 & 0 & 0 \end{pmatrix} = -\mathbf{u}\mathbf{J}\mathbf{u}^T. \tag{23}$$

```
xu =jnp . array ( [u0 , u1 , u2 , u3 ])eu = 1# 稠密表示与独热表示的拼
接u =(xu , eu)
```

- **更新2行和2列**

$$\begin{aligned} \mathbf{A}_1 - \mathbf{A}_0 &= \begin{pmatrix} 0 & -u_{00} & 0 & -u_{10} & 0 & 0 \\ u_{00} & 0 & u_{02} & u_{03} - u_{11} & u_{04} & u_{05} \\ 0 & -u_{02} & 0 & -u_{12} & 0 & 0 \\ u_{10} & u_{11} - u_{03} & u_{12} & 0 & u_{14} & u_{15} \\ 0 & -u_{04} & 0 & -u_{14} & 0 & 0 \\ 0 & -u_{05} & 0 & -u_{15} & 0 & 0 \end{pmatrix} \\ &= - \begin{pmatrix} u_{00} & u_{10} & 0 & 0 \\ u_{01} & u_{11} & 1 & 0 \\ u_{02} & u_{12} & 0 & 0 \\ u_{03} & u_{13} & 0 & 1 \\ u_{04} & u_{14} & 0 & 0 \\ u_{05} & u_{15} & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} u_{00} & u_{01} & u_{02} & u_{03} & u_{04} & u_{05} \\ u_{10} & u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \\ &= -\mathbf{u}\mathbf{J}\mathbf{u}^T. \end{aligned} \tag{24}$$

```
xu =jnp. array ( [[u00 , u01 , u02 , u03 , u04 , u05 ] ,[u10 , u11 , u12 ,
u13 , u14 , u15 ] ,]) .Teu =jnp. array ( [1 , 3])# 密集表示与独热表示
的拼接u =(xu , eu)
```

C. 延迟更新

与行列式状态的 LRU 类似，Pfaffian 的时间瓶颈也来自内存受限计算。在每一步 t，我们定义

$$\mathbf{a}_t = \mathbf{A}_{t-1}^{-1}\mathbf{u}_t, \tag{25}$$

使得

$$\mathbf{A}_\tau^{-1} = \mathbf{A}_{\tau-1}^{-1} + \mathbf{a}_\tau \mathbf{R}_\tau^{-1} \mathbf{a}_\tau^T = \mathbf{A}_0^{-1} + \sum_{t=1}^{\tau} \mathbf{a}_t \mathbf{R}_t^{-1} \mathbf{a}_t^T. \tag{26}$$

与其显式构造$\mathbf{A}_\tau^{-1}$（根据式（21））， 也可以将$\mathbf{A}_0^{-1}$、$\mathbf{a}_t$和$\mathbf{R}_t^{-1}$存储在内存中，它们包含了重建$\mathbf{A}_\tau^{-1}$所需的足够信息。结合式（20）和式（26）， 我们可以通过以下方式计算$\mathbf{R}_\tau$:

$$\mathbf{R}_\tau = \mathbf{J} + \mathbf{u}_\tau^T \mathbf{A}_0^{-1} \mathbf{u}_\tau + \sum_{t=1}^{\tau-1} (\mathbf{u}_\tau^T \mathbf{a}_t) \mathbf{R}_t^{-1} (\mathbf{u}_\tau^T \mathbf{a}_t)^T \tag{27}$$

时间复杂度为 $\mathcal{O}(n^2k + \tau nk^2)$。随后，需要计算 $k \times k$ 矩阵$\mathbf{R}_\tau$的矩阵逆，并将$\mathbf{R}_\tau^{-1}$存储在内存中，以便进行延迟更新。同样，结合式（25）和式（26）， $\mathbf{a}_\tau$可以计算为

$$\mathbf{a}_\tau = \mathbf{A}_0^{-1} \mathbf{u}_t + \sum_{t=1}^{\tau-1} \mathbf{a}_t \mathbf{R}_t^{-1} (\mathbf{u}_\tau^T \mathbf{a}_t)^T \tag{28}$$

时间复杂度同样为 $\mathcal{O}(n^2k+\tau nk^2)$，内存复杂度为 $\mathcal{O}(n^2+\tau nk$，用于存储$\mathbf{A}_0^{-1}$和$\mathbf{a}_t$。与行列式的情况类似，当26达到上界$_\tau^{-1}$时，应利用式（$\mathbf{A}_\tau^{-1}$26）重建$\mathbf{A}_0^{-1}$，并将其设为新的 τ。通常选择 T，但最优值也取决于硬件和矩阵大小。下面给出 Pfaffian 延迟更新的示例代码，其中设定 $T \approx n/20k$。

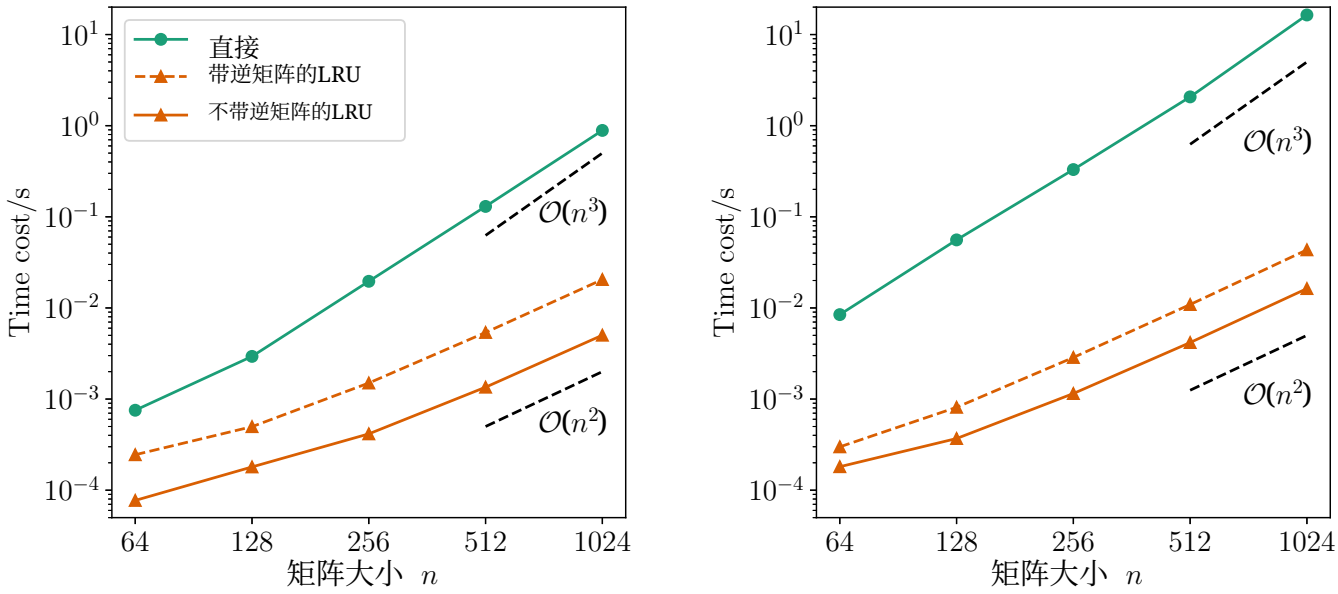
```
n = 10最大_延迟 = 4A =jr . 正态分布 ( get_key () , (n , n)) A = (A - A.T) / 2 载体 =
lrux . init_pf_载体 (A , 最大_延迟 ) pfA =lrux . pf (A) det_lru_fn =jax . 即时编译 (
lrux . pf_lru_延迟 , static_argnums =(2 , 3) , donate_argnums =0) 对于 i in range (20) :
当前_延迟 =i % 最大_延迟 # 随机更新 x =jr . 正态分布 ( get_key () , (n ,) ) # 选择一个随
机行和列 e =jr . randint ( get_key () , shape =(1 ,) , minval =0, maxval =n) u =(x ,
e) # 结合稠密和独热表示 r , 载体 =det_lru_fn ( 载体 , u , True , 当前_延迟 ) pfA *=r # 验
证低秩更新结果 A =A. at [e [0]]. add (x). at [: , e [0]]. add (-x) pfA_精确 =lrux .
pf (A) assert jnp . isclose (pfA , pfA_精确 )
```

IV. 基准测试结果

A. 单步比较

我们首先展示 LRU 在 A100-80GB GPU 上实现的加速效果。为充分利用 GPU 的浮点运算能力，我们测量了通过 jax.vmap 并行计算 1024 个行列式或 Pfaffian 的时间开销，这模拟了现代变分蒙特卡罗中的批量波函数评估。在行列式和 Pfaffian 的测试中，我们选择$k = 1$ 并改变 n， 以展示矩阵尺寸扩展性。

在图1中，我们绘制了不同方法的时间开销随矩阵大小变化的情况 n。虽然对于较小的 n，缩放关系并不明确，但对于较大的 n， 我们可以清楚地看到，直接计算行列式和 Pfaffian 的时间开销呈现 $\mathcal{O}(n^3)$ 缩放关系，



(a) 行列式　　(b) Pfaffian

图1：时间开销的缩放关系。图中对比了直接计算、更新矩阵逆的 LRU，以及不更新矩阵逆的 LRU。虚线表示参考缩放关系 $\mathcal{O}(n^2)$ 和 $\mathcal{O}(n^3)$。

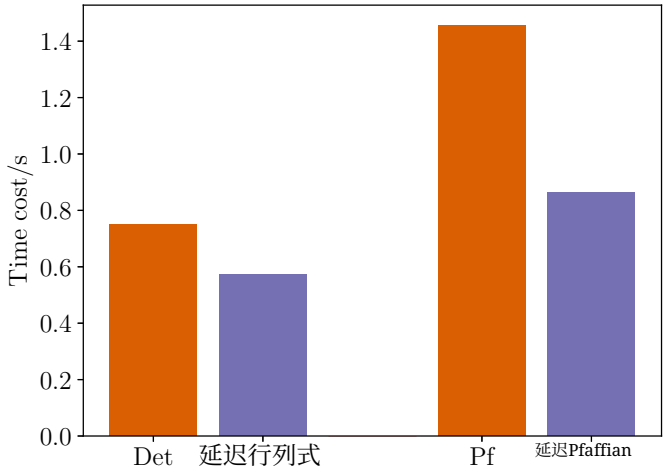


图 2：LRU 使用和不使用延迟更新时的时间开销

而 LRU 仅呈现 $\mathcal{O}(n^2)$。使用矩阵逆更新的 LRU $\mathbf{A}^{-1}$ 比不使用矩阵逆更新的 LRU 时间开销更大，但缩放关系相同。比较所展示的最大矩阵大小 $n = 1024$下的时间开销，LRU 技术使行列式计算加速 $\sim 200\times$，使 Pfaffian 计算加速 $\sim 1000\times$。这些结果凸显了 LRU 在更新行列式和 Pfaffian 时的重要性，也体现了 lrux 的高效性。

B. 延迟更新

接下来，我们比较直接更新矩阵逆的 LRU 与采用延迟更新的 LRU 之间的效率。测试仍在 A100-80GB GPU 上进行。我们选择并行计算 16384 个行列式和 Pfaffian，矩阵大小为 $n = 128$，并执行 n 次连续更新，以比较总时间开销。延迟更新步数的上界 T 通过数值实验设定，以最大限度地提高效率。经过数轮测试后

，我们发现，在我们的环境中，行列式的最优选择是 $T = 16$，Pfaffian 的最优选择是 $T = 4$。

在图2中，我们比较了采用和不采用延迟更新时 LRU 的时间开销。两种方法的效率相当，而延迟更新还能额外提升 20% $\sim$ 40的速度。当用户希望将计算效率推至极限时，这一提升会很有用，尤其是在 LRU 成为整个程序的时间瓶颈时。不过，延迟更新的性能很大程度上取决于 T 用户所用平台上的合适选择。因此，我们建议用户在使用延迟更新时自行进行基准测试。否则，采用直接矩阵求逆更新的 LRU 更为合适。

V. 结论

我们介绍了 lrux，这是一个基于 JAX 的库，可为行列式和 Pfaffian 提供高效的（延迟）低秩更新算法，使常见于量子蒙特卡罗和费米子神经量子态计算中的连续更新能够实现 $\mathcal{O}(n^2k)$ 缩放关系。借助 JAX 的 JIT编译、向量化和加速器支持，lrux 能在现代 GPU 上实现显著加速，同时保持数值稳定性。该软件包提供灵活的接口，支持稠密和独热更新表示，以及实数和复数数据类型。我们期望 lrux 能成为相互作用费米子系统大规模模拟的稳健构建模块，并推动新一代变分波函数和采样算法的开发。

致谢

AC 衷心感谢 Garnet Kin-Lic Chan 和 Markus Heyl 的支持。我们感谢 Anirvan Sengupta、Antoine Georges、Miguel Morales 和 Zhou-Quan Wan 所进行的有益讨论。我们也感谢 Flatiron 研究所提供的支持和计算资源。Flatiron 研究所是 Simons 基金会的一个部门。

附录 A：Pfaffian

1. 定义

Pfaffian 将一个 $2n \times 2n$ 反对称矩阵 $\mathbf{A}$ 映射为一个数 pf$\mathbf{A}$。Pfaffian 的形式定义如下。将所有数字 $\{1, ..., 2n\}$ 划分为 n 对 $\alpha = \{(i_1, j_1) ..... (i_n, j_n)\}$，满足 $i_k < j_k$ 和 $i_1 < i_2 < ... < i_n$，总共有 $(2n-1)!!$ 种可能的划分。然后，元素为 $A_{i,j}$ 的矩阵 $\mathbf{A}$ 的 Pfaffian 定义为

$$\mathrm{pf}\mathbf{A} = \sum_{\alpha} \mathrm{sign}(\alpha) \prod_{k=1}^{n} A_{i_k, j_k}, \tag{A.1}$$

其中 sign(α) 是排列 (i_1, j_1, i_2, j_2, ..., i_n, j_n) 的奇偶性。例如，

$$\mathrm{pf} \begin{pmatrix} 0 & a \\ -a & 0 \end{pmatrix} = a, \tag{A.2}$$

$$\mathrm{pf} \begin{pmatrix} 0 & a & b & c \\ -a & 0 & d & e \\ -b & -d & 0 & f \\ -c & -e & -f & 0 \end{pmatrix} = af - be + cd. \tag{A.3}$$

Pfaffian 的时间复杂度为 $\mathcal{O}(n^3)$，与行列式相同。

在 lrux 中，我们定义了两个函数 pf 和 slogpf，用于计算反对称矩阵的 Pfaffian。这些函数针对 JAX 框架进行了设计，以确保良好的数值稳定性，而非追求最高效率。pf(A) 直接返回（批处理的）Pfaffian 值，而 slogpf 返回一个命名元组对象，其中包含两个值，分别等于 jnp.sign(pf(A)) 和 jnp.log(jnp.abs(pf(A)))。

2. 性质

作为参考，这里列出 Pfaffian 的几条重要性质，不作证明。

$$\mathrm{pf}^2(\mathbf{A}) = \det(\mathbf{A}), \tag{A.4}$$

$$\mathrm{pf}(\mathbf{B}\mathbf{A}\mathbf{B}^T) = \det(\mathbf{B})\mathrm{pf}(\mathbf{A}), \tag{A.5}$$

$$\mathrm{pf}\begin{pmatrix} \mathbf{0} & \mathbf{A} \\ -\mathbf{A}^T & \mathbf{0} \end{pmatrix} = (-1)^{n(n-1)/2}\det(\mathbf{A}), \tag{A.6}$$

$$\mathrm{pf}\begin{pmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}' \end{pmatrix} = \mathrm{pf}(\mathbf{A})\mathrm{pf}(\mathbf{A}'), \tag{A.7}$$

$$\frac{\mathrm{pf}(\mathbf{A}+\mathbf{B}\mathbf{C}\mathbf{B}^T)}{\mathrm{pf}(\mathbf{A})} = \frac{\mathrm{pf}(\mathbf{C}^{-1}+\mathbf{B}^T\mathbf{A}^{-1}\mathbf{B})}{\mathrm{pf}(\mathbf{C}^{-1})}, \tag{A.8}$$

$$\mathrm{pf}\begin{pmatrix} \mathbf{M} & \mathbf{Q} \\ -\mathbf{Q}^T & \mathbf{N} \end{pmatrix} = \mathrm{pf}(\mathbf{M})\,\mathrm{pf}(\mathbf{N}+\mathbf{Q}^T\mathbf{M}^{-1}\mathbf{Q}) = \mathrm{pf}(\mathbf{N})\,\mathrm{pf}(\mathbf{M}+\mathbf{Q}\mathbf{N}^{-1}\mathbf{Q}^T). \tag{A.9}$$

3. 梯度

这里，我们推导 Pfaffian 的梯度。假设在$\mathbf{A}$ 的第 i 行和第 j 列施加变化 $\mathrm{d}x$ 。则

$$(\mathrm{d}\mathbf{A})_{kl} = \delta_{ik}\delta_{jl}\,\mathrm{d}x. \tag{A.10}$$

由于反对称约束，我们将矩阵 $\mathbf{A}$ 反对称化，得到

$$\mathbf{A}' = \frac{\mathbf{A}-\mathbf{A}^T}{2}, \tag{A.11}$$

其变化为

$$(\mathrm{d}\mathbf{A}')_{kl} = \frac{\delta_{ik}\delta_{jl}-\delta_{il}\delta_{jk}}{2}\mathrm{d}x. \tag{A.12}$$

那么我们有 $\mathbf{A}'^T = -\mathbf{A}'$ ，且 $\mathrm{d}\mathbf{A}'^T = -\mathrm{d}\mathbf{A}'$。变化后的 $\mathrm{pf}\mathbf{A}'$ 值为

$$\begin{aligned} \mathrm{pf}(\mathbf{A}'+\mathrm{d}\mathbf{A}') &= \mathrm{pf}\left[\left(\mathbf{1}+\frac{1}{2}\mathbf{A}'^{-1}\mathrm{d}\mathbf{A}'\right)^T\mathbf{A}'\left(\mathbf{1}+\frac{1}{2}\mathbf{A}'^{-1}\mathrm{d}\mathbf{A}'\right)\right]+\mathcal{O}(\mathrm{d}x^2) \\ &= \mathrm{pf}(\mathbf{A}')\det\left(\mathbf{1}+\frac{1}{2}\mathbf{A}'^{-1}\mathrm{d}\mathbf{A}'\right)+\mathcal{O}(\mathrm{d}x^2) \\ &= \mathrm{pf}(\mathbf{A}')\left(1+\frac{1}{2}\mathrm{tr}(\mathbf{A}'^{-1}\mathrm{d}\mathbf{A}')\right)+\mathcal{O}(\mathrm{d}x^2), \end{aligned} \tag{A.13}$$

其中我们利用了$\mathbf{A}'$ 的反对称性以及式（A.5）。利用爱因斯坦求和约定，上述迹可计算为

$$\mathrm{tr}(\mathbf{A}'^{-1}\mathrm{d}\mathbf{A}') = (\mathbf{A}'^{-1})_{lk}(\mathrm{d}\mathbf{A}')_{kl} = (\mathbf{A}'^{-1})_{lk}\frac{\delta_{ik}\delta_{jl}-\delta_{il}\delta_{jk}}{2}\mathrm{d}x = (\mathbf{A}'^{-1})_{ji}\,\mathrm{d}x. \tag{A.14}$$

那么，$\mathrm{pf}\mathbf{A}'$ 的偏导数可计算为

$$\frac{\partial\,\mathrm{pf}\mathbf{A}'}{\partial A_{ij}} = \lim_{\mathrm{d}x\to 0}\frac{\mathrm{pf}(\mathbf{A}'+\mathrm{d}\mathbf{A}')-\mathrm{pf}\mathbf{A}'}{\mathrm{d}x} = \frac{1}{2}\mathrm{pf}(\mathbf{A}')(\mathbf{A}'^{-\mathbf{1}})_{ji}. \tag{A.15}$$

在 lrux中，函数 pf 和 slogpf 的梯度根据公式(A.15)使用 jax.custom jvp进行定制，并且同时支持实数和复数数据类型。

---

[1] J. B. Anderson，薛定¨谔方程的随机游走模拟：H+3，《化学物理学杂志》63，1499（1975）。[2] D. M. Ceperley 和 B. J. Alder，用随机方法得到的电子气基态，《物理评论快报》45，566（1980）。[3] J. E. Hirsch，二维哈伯德模型：数值模拟研究，《物理评论B辑》31，4403（1985）。[4] R. Blankenbecler、D. J. Scalapino 和 R. L. Sugar，耦合玻色子-费米子系统的蒙特卡罗计算。i，《物理评论D辑》24，2278（1981）。[5] S. Zhang、J. Carlson 和 J. E. Gubernatis，费米子基态的受限路径量子蒙特卡罗方法，《物理评论快报》74，3652（1995）。[6] W. L. McMillan，液态 he^4，的基态，《物理评论》138，A442（1965）。[7] W. M. C. Foulkes、L. Mitas、R. J. Needs 和 G. Rajagopal，固体的量子蒙特卡罗模拟，《现代物理评论》73，33（2001）。[8] G. Carleo 和 M. Troyer，用人工神经网络求解量子多体问题，《科学》355，602（2017）。[9] W. T. Lou、H. Sutterud、G. Cassella、W. M. C. Foulkes、J. Knolle、D. Pfau 和 J. S. Spencer，超流体的神经网络波函数，Phys. Rev. X 14，021030（2024）。[10] A. Chen、Z.-Q. Wan、A. Sengupta、A. Georges 和 C. Roth，用于相互作用费米子可扩展模拟的神经网络增强的 Pfaffian 波函数（2025年），arXiv:2507.10705 [cond-mat.str-el]。[11] C. Roth、A. Chen、A. Sengupta 和 A. Georges，神经量子态揭示的二维哈伯德模型中的超导性（2025年），arXiv:2511.07566 [cond-mat.supr-con]。[12] A. Chen，*Large-scale simulation of deep neural quantum states*，博士论文，Universit¨at Augsburg（2025年）。[13] J. R. Moreno、G. Carleo、A. Georges 和 J. Stokes，由神经网络约束隐态得到的费米子波函数，《美国国家科学院院刊》119，e2122059119（2022）。[14] D. Luo 和 B. K. Clark，通过神经网络实现量子多体波函数的回流变换，《物理评论快报》122，226401（2019）。[15] D. Pfau、J. S. Spencer、A. G. D. G. Matthews 和 W. M. C. Foulkes，用深度神经网络从头算求解多电子薛定¨谔方程，Phys. Rev. Res. 2，033429（2020）。[16] J. Hermann、J. Spencer、K. Choo、A. Mezzacapo、W. M. C. Foulkes、D. Pfau、G. Carleo 和 F. No´e，使用神经网络波函数的从头算量子化学，《自然评论：化学》7，692（2023）。[17] J. Bradbury、R. Frostig、P. Hawkins、M. J. Johnson、C. Leary、D. Maclaurin、G. Necula、A. Paszke、J. VanderPlas、S. Wanderman-Milne 和 Q. Zhang，JAX：Python+NumPy 程序的可组合变换。