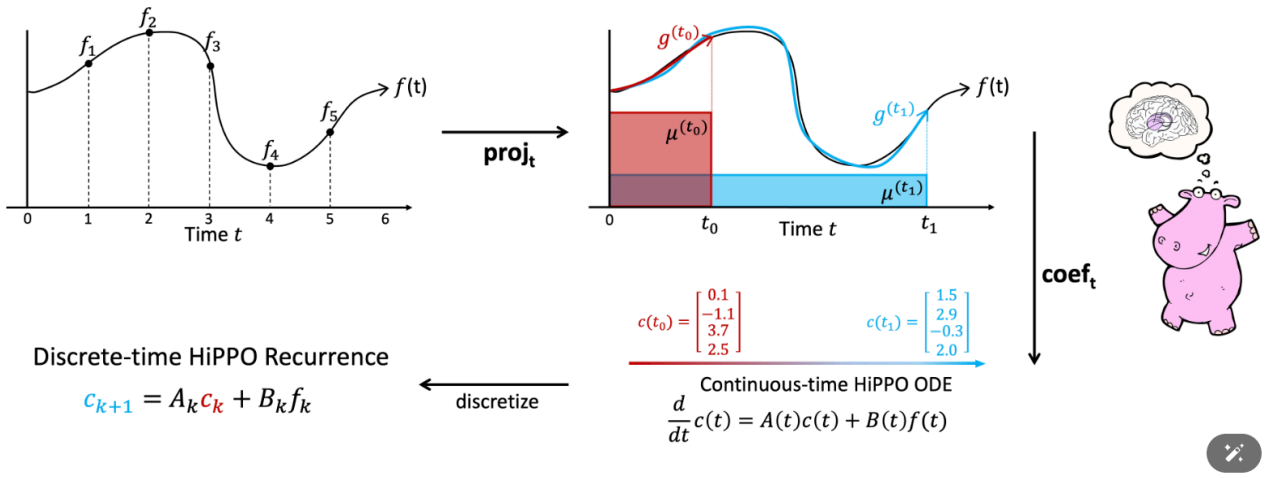
The Annotated S4 - Implementation S4

背景：SSM的性能问题



之前的研究发现，基础的 SSM 在实际中表现非常差。直观上，一个可能的解释是它们在序列长度增加时会遭遇梯度呈指数级缩放的问题（即梯度消失/爆炸问题）。

使用 HiPPO 处理长程依赖

为了应对这一问题，之前的研究提出了 **HiPPO（High-order Polynomial Projection Operators）** 理论，用于连续时间记忆。

- **HiPPO的核心思想**：
  HiPPO 定义了一类特殊的矩阵 $A \in \mathbb{R}^{N\times N}$，当将这些矩阵引入到模型中时，可以让状态 $x(t)$ 记住输入 $u(t)$ 的历史信息。
- **HiPPO矩阵的定义**：
  在这类矩阵中，最重要的是 **HiPPO矩阵**，其定义如下：

$$A_{nk} = \begin{cases} \sqrt{(2n+1)(2k+1)} & \text{if } n > k \\ n+1 & \text{if } n = k \\ 0 & \text{if } n < k \end{cases} \tag{1}$$

- n*：状态索引，表示当前需要更新的状态分量。

- k：输入历史索引，表示输入历史中影响状态的部分。

HiPPO的性能提升

- *以往的研究发现，仅仅将SSM 中的随机矩阵 A 替换为 HiPPO 矩阵，就能显著提升模型性能。*
- *在 **Sequential MNIST 分类基准测试** 中，性能从 **60%** 提升到了 **98%**。*

HiPPO矩阵的**设计理念**在于**生成一个能够记忆历史的隐状态**。对于我们的目的，我们主要需要知道以下两点：

1. 我们只需要计算它一次。
2. 它具有一个简洁优美的结构（我们将在第二部分中利用这一点）。

不深入讨论与ODE相关的数学内容，**主要的结论是：这个矩阵旨在将过去的历史压缩成一个状态，该状态包含足够的信息以近似重构历史。**

```
def make_HiPPO(N):
    P = np.sqrt(1 + 2 * np.arange(N))
    A = P[:, np.newaxis] * P[np.newaxis, :]
    A = np.tril(A) - np.diag(np.arange(N))
    return -A
```

深入一点解释，这个矩阵的直观意义在于它生成了一个能够记忆其历史的隐藏状态。它通过跟踪勒让德多项式的系数来实现这一点。这些系数使其能够近似所有的先前历史。让我们来看一个例子：

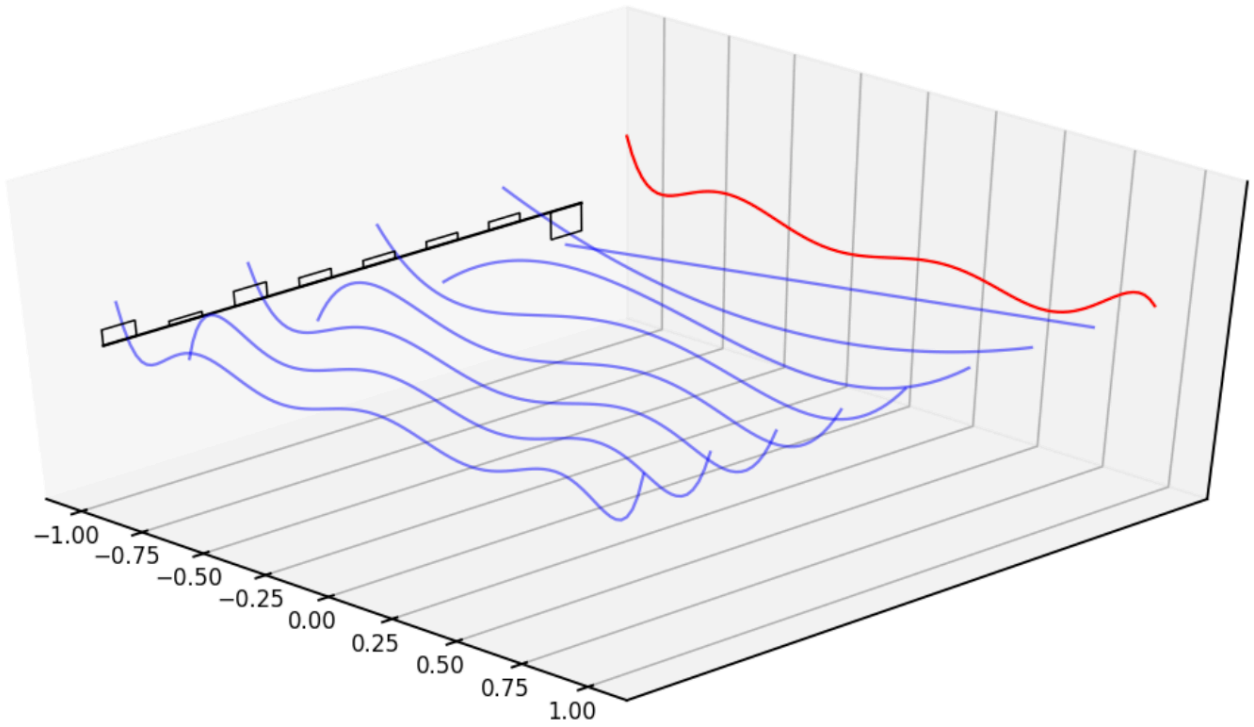
```
def example_legendre(N=8):
    # Random hidden state as coefficients
    import numpy as np
    import numpy.polynomial.legendre

    x = (np.random.rand(N) - 0.5) * 2
    t = np.linspace(-1, 1, 100)
    f = numpy.polynomial.legendre.Legendre(x)(t)

    # Plot
    import matplotlib.pyplot as plt
    import seaborn

    seaborn.set_context("talk")
    fig = plt.figure(figsize=(20, 10))
    ax = fig.gca(projection="3d")
    ax.plot(
        np.linspace(-25, (N - 1) * 100 + 25, 100),
        [0] * 100,
        zs=-1,
```

```
        zdir="x",
        color="black",
    )
    ax.plot(t, f, zs=N * 100, zdir="y", c="r")
    for i in range(N):
        coef = [0] * N
        coef[N - i - 1] = 1
        ax.set_zlim(-4, 4)
        ax.set_yticks([])
        ax.set_zticks([])
        # Plot basis function.
        f = numpy.polynomial.legendre.Legendre(coef)(t)
        ax.bar(
            [100 * i],
            [x[i]],
            zs=-1,
            zdir="x",
            label="x%d" % i,
            color="brown",
            fill=False,
            width=50,
        )
        ax.plot(t, f, zs=100 * i, zdir="y", c="b", alpha=0.5)
    ax.view_init(elev=40.0, azim=-45)
    fig.savefig("images/leg.png")
if False:
    example_legendre()
```



红线表示我们正在近似的曲线，而黑色的条形表示隐藏状态的值。每个值都是勒让德级数中一个元素的系数，这些元素用蓝色函数表示。直观上，HiPPO 矩阵在每一步都会更新这些系数。

*警告：这一部分包含大量数学内容。简单来说，这部分主要是找到一种方法，可以**非常快速地计算第 1 部分中提到的“HiPPO 类”矩阵的滤波器**。如果你感兴趣，这些细节非常有趣。如果不感兴趣，可以直接跳到第 3 部分，了解一些酷炫的应用，比如 MNIST 的补全。*

为了铺垫背景，回忆一下 S4 与普通 SSM 的两个主要区别：

1. **第一个区别**：解决了建模中的一个挑战——长程依赖问题。
   通过使用前一部分定义的 A 矩阵的特殊公式来实现。这些特殊的 SSM 在 S4 的前身工作中已经被研究过。
2. **第二个主要特性**：S4 通过引入一种特殊的表示方法和算法，解决了 SSM 的计算挑战，从而能够高效地处理这个矩阵！

计算离散时间 SSM 的根本瓶颈在于它需要反复进行矩阵 $\overline{A}$ 的乘法。例如，使用简单方法计算时，需要进行 L 次连续的 $\overline{A}$ 矩阵乘法，这会导致以下复杂度：

- ***时间复杂度**：$O(N^2L)$，其中 N 是矩阵的维度，L 是乘法次数。*
- ***空间复杂度**：$O(NL)$，因为需要存储中间结果。*

具体来说，回顾一下这里的函数：

```
def K_conv(Ab, Bb, Cb, L):
    return np.array(
        [(Cb @ matrix_power(Ab, l) @ Bb).reshape() for l in range(L)]
    )
```

S4 的贡献在于提供了一种稳定的方法来加速这一特定操作。为此，我们将重点关注一种具有特殊结构的 SSM：即在复数空间中具有 **对角加低秩（Diagonal Plus Low-Rank, DPLR）** 结构的情况。

一个 DPLR 形式的 SSM 可以表示为：

$$(\Lambda - PQ^*, B, C) \tag{3}$$

其中：

- Λ 是**对角矩阵**：只有对角线上的元素不为零，其余位置全是零。例如：

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \tag{4}$$

  这样的矩阵计算起来很快。

- **低秩矩阵**：可以用很少的数值来描述的矩阵。例如，一个秩为 1 的矩阵可以写成两个向量的乘积：

$$PQ^* \tag{5}$$

  其中 P 和 Q 是列向量，Q^* 是 Q 的转置共轭。

  - **共轭**：把矩阵中的复数取共轭（实数部分不变，虚数部分取负）。例如：

$$z = a + bi \quad \Rightarrow \quad z^* = a - bi \tag{6}$$

  - **转置共轭**：先转置，再取共轭。如果矩阵是复数矩阵，转置共轭的过程可以写作：

$$Q^* = (Q^T)^{\text{共轭}} \tag{7}$$

- $P, Q, B, C \in \mathbb{C}^{N \times 1}$，即这些矩阵是列向量。

我们可以不失一般性地假设秩为 1（rank = 1），也就是说，这些矩阵可以看作是向量。

在这个 DPLR 假设下，S4 通过以下三步克服了速度瓶颈：

1. 避免直接计算 K

在许多情况下，直接计算矩阵 K 的幂次或特征值（spectrum）可能非常复杂，尤其当矩阵的维度很大时。这种计算会导致高昂的时间和空间成本。

*为了解决这个问题，我们使用**截断生成函数（truncated generating function）**来间接计算 K 的谱。*

- ***生成函数是什么？***
  生成函数是一种数学工具，可以将矩阵的行为用一个函数描述出来。通过分析这个函数，我们可以提取矩阵的特征值信息，而不需要直接操作矩阵本身。

- ***为什么用矩阵求逆而不是幂运算?***
  *矩阵的幂运算（比如 K^n）通常比矩阵求逆复杂得多。通过转换为矩阵求逆操作，我们可以利用更高效的数值算法，**尤其是当矩阵具有特殊结构（如对角矩阵或低秩矩阵）时。***

2. 对角矩阵的等价性

***对于对角矩阵，计算特征值的过程可以大大简化**。我们证明了，对角矩阵的情况等价于计算一个 **Cauchy 核**，其形式为：*

$$\frac{1}{\omega_j - \zeta_k} \tag{8}$$

- ***Cauchy 核的意义***
  这个公式描述了两个变量 ω_j 和 ζ_k 之间的关系。它的分母表示两个变量之间的差距，而分子是常数 1。这个核在许多数学和物理问题中都非常重要，因为它可以捕捉到矩阵的特征值分布。
- ***为什么对角矩阵这么特别?***
  对角矩阵的特点是，只有对角线上的元素是非零的，这使得矩阵的求逆操作和特征值分解变得十分简单。通过引入 Cauchy 核，我们可以直接利用对角矩阵的这一性质，避免冗长的计算。

3. 低秩修正方法

*在实际应用中，**矩阵 K 通常不是简单的对角矩阵，而是包含一些低秩修正项（low-rank terms）**。这些低秩修正项使得计算变得更加复杂，因此我们需要一种方法来简化它们的处理。*

- ***Woodbury 恒等式的作用***
  Woodbury 恒等式是一种数学技巧，用于快速计算低秩修正矩阵的逆。其公式为：

  $$(\Lambda + PQ^*)^{-1} = \Lambda^{-1} - \Lambda^{-1}P(I + Q^*\Lambda^{-1}P)^{-1}Q^*\Lambda^{-1} \tag{9}$$

  其中：
  - *Λ 是一个对角矩阵（或更一般的矩阵）。*
  - *P 和 Q 是低秩矩阵。*

  通过这个公式，我们可以将一个复杂矩阵的求逆过程分解为对角矩阵 Λ 的求逆和一些简单的矩阵乘法。
- ***如何简化到对角矩阵的情况?***
  *Woodbury 恒等式的核心思想是，**将复杂矩阵的逆运算转换为对角矩阵的逆运算**。这使得我们能够充分利用对角矩阵的高效计算特性，同时处理低秩修正项的影响。*

通过上述方法，我们将复杂的矩阵计算问题分解为以下几个步骤：

1. 使用生成函数代替直接计算矩阵 K 的幂或特征值。
2. 利用对角矩阵的特殊性质，通过 Cauchy 核快速提取特征值信息。
3. 使用 Woodbury 恒等式处理低秩修正项，将问题最终简化为对角矩阵的求逆操作。

第 1 步：SSM 生成函数

根据前文，得知，生成函数的好处在于：

1. **压缩信息**：将一个复杂的矩阵序列（可能包含无穷多项）压缩成一个函数，便于分析。
2. **高效计算**：通过截断和特定算法，可以更快地计算矩阵的行为，而不需要直接操作大矩阵。
3. **频率分析**：通过改变 z 的值，可以研究系统在不同频率或时间点的特性。

这一步的主要目的是**从计算序列切换到计算其生成函数**。

- 序列计算公式：

$$K_L = \left[\bar{C}\bar{B}, \bar{C}\bar{A}\bar{B}, \bar{C}\bar{A}^2\bar{B}, \ldots, \bar{C}\bar{A}^{L-1}\bar{B}\right] \tag{10}$$

- 生成函数计算公式：

$$\hat{K}_L(z) = \sum_{i=0}^{L-1} \bar{C}\bar{A}^i\bar{B}z^i \tag{11}$$

- 转换过程：

$$\left[\bar{C}\bar{B}, \bar{C}\bar{A}\bar{B}, \ldots\right] \quad \longrightarrow \quad \sum_{i=0}^{L-1} \bar{C}\bar{A}^i\bar{B}z^i \tag{12}$$

生成函数通过引入 z 将矩阵序列压缩为一个函数，便于分析和高效计算，同时支持频率分析（通过调整 z 的值）。下面是详细的解释：

为了解决计算 $\bar{A}$ 的幂的问题，我们引入了另一种技术。我们不是直接计算 SSM 卷积滤波器 $\bar{K}$，而是对其系数引入生成函数并计算其评估值。

在节点 z 处，截断为 L 的 SSM 生成函数定义为：

$$\hat{K}_L(z; \bar{A}, \bar{B}, \bar{C}) \in \mathbb{C} := \sum_{i=0}^{L-1} \bar{C}\bar{A}^i\bar{B}z^i \tag{13}$$

*它是用来描述矩阵序列行为的工具。其核心思想用来**将序列（比如信号或矩阵幂）压缩到一个函数**中。这个函数可以在不同的 z 值下被评估，从而提供关于序列的全面信息。*

- $\bar{A}$：*矩阵，表示系统的核心动态行为。*
- $\bar{B}$：*向量，表示输入信号如何影响系统。*
- $\bar{C}$：*向量，表示如何从系统中提取输出信号。*
- z：***一个复数变量，表示生成函数的评估点**。它可以看作一个参数，用来分析系统在不同频率或时刻的行为。*
- L：*截断点，表示我们只考虑矩阵幂的前 L 项，而不是所有项。这是为了限制计算复杂度。*

- *在这里，$\hat{K}_L(z;\bar{A},\bar{B},\bar{C})$ 是一个生成函数，它的值是通过对矩阵序列的加权求和得到的：*
  - *$\bar{A}^i$：表示矩阵 $\bar{A}$ 的第 i 次幂，描述系统运行到第 i 个时间步的状态。*
  - *$\bar{C}\bar{A}^i\bar{B}$：表示第 i 个时间步系统的输出信号。*
  - *z^i：为每一项引入的权重，表示不同时间步的贡献可以根据 z 的值进行调整。*
- *截断点 L 限制了求和的项数，只计算前 L 项。这是因为在实际应用中，矩阵幂的高次项可能对结果的影响较小，同时计算成本较高。*

```
def K_gen_simple(Ab, Bb, Cb, L):
    K = K_conv(Ab, Bb, Cb, L)

    def gen(z):
        return np.sum(K * (z ** np.arange(L)))

    return gen
```

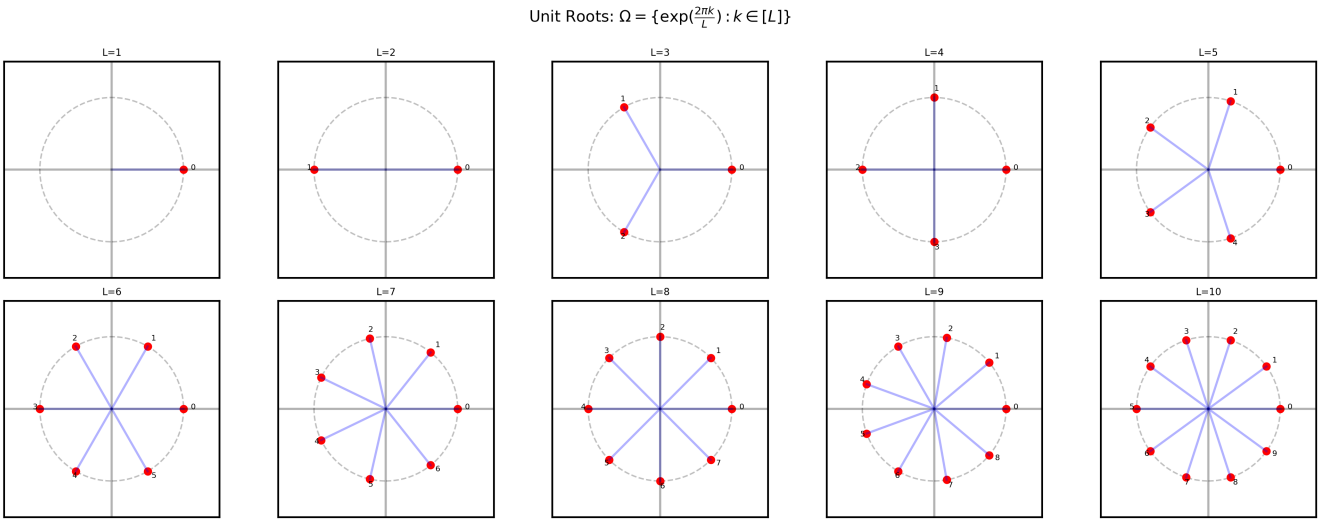
生成函数本质上将 SSM 卷积滤波器从**时域**转换到**频域**。在控制工程文献中，这种变换被称为 **Z 变换**（差一个负号）。重要的是，这种变换**保留了相同的信息**，并且可以从中恢复出所需的 SSM 卷积滤波器。下面**解释恢复过程：**

***一旦我们知道序列的 Z 变换**，就可以通过在**单位根**处对其进行评估来**获得滤波器的离散傅里叶变换（DFT）**。单位根的集合定义为：*

$$\Omega = \left\{ \exp\left(\frac{2\pi k}{L}\right) : k \in [L] \right\} \tag{14}$$

这里：

- $\exp\left(\frac{2\pi k}{L}\right)$：*表示单位圆上的第 k 个单位根。*
- L：*表示截断点，即序列的长度。*



*通过对 Z 变换在这些单位根处（如上图所示，L=10）的评估，我们可以应用**逆傅里叶变换**（Inverse Fourier Transform, IFFT）来稳定地**恢复滤波器**。这可以通过快速傅里叶变换（FFT）实现，其计算复杂度为：*

$$O(L \log L) \tag{15}$$

从生成函数恢复到原始的计算序列，关键点如下：

- **生成函数**将时域序列映射到频域（Z 变换）。
- **单位根评估**提供了离散傅里叶变换（DFT）。
- **逆傅里叶变换**（通过 FFT）可以高效、稳定地恢复原始滤波器。

```
def conv_from_gen(gen, L):
    # Evaluate at roots of unity
    # Generating function is (-)z-transform, so we evaluate at (-)root
    Omega_L = np.exp((-2j * np.pi) * (np.arange(L) / L))
    atRoots = jax.vmap(gen)(Omega_L)
    # Inverse FFT
    out = np.fft.ifft(atRoots, L).reshape(L)
    return out.real
```

更重要的是：生成函数中的矩阵幂可以替换为矩阵的逆！生成函数可以通过以下公式表示：

$$\hat{K}_L(z) = \sum_{i=0}^{L-1} \bar{C}\bar{A}^i\bar{B}z^i \tag{16}$$

通过对公式进行变换，可以将矩阵幂替换为矩阵的逆：

$$\hat{K}_L(z) = \bar{C}\left(I - \bar{A}^L z^L\right)\left(I - \bar{A}z\right)^{-1}\bar{B} \tag{17}$$

对于所有 $z \in \Omega_L$

$$z^L = \left(\exp\left(\frac{2\pi i k}{L}\right)\right)^L = \exp\left(2\pi i k\right) = 1 \tag{18}$$

我们有 $z^L = 1$, 因此可以将该项移除。最终公式化简为：

$$\hat{K}_L(z) = \tilde{C}\left(I - \bar{A}z\right)^{-1}\bar{B} \tag{19}$$

其中：

- $\tilde{C}$：是将常数项 $\bar{C}(I - \bar{A}^L)$ 合并后的新系数。

到目前为止, 进一步优化，使得不需要调用`K_conv`

- 通过引入矩阵逆，生成函数的计算得到了简化。

- 单位根的特性进一步减少了公式中的复杂项。
- 该方法在计算效率和稳定性上更具优势。

```
def K_gen_inverse(Ab, Bb, Cb, L):
    I = np.eye(Ab.shape[0])
    Ab_L = matrix_power(Ab, L)
    Ct = Cb @ (I - Ab_L)
    return lambda z: (Ct.conj() @ inv(I - Ab * z) @ Bb).reshape()
```

但它确实输出相同的值

```
def test_gen_inverse(L=16, N=4):
    ssm = random_SSM(rng, N)
    ssm = discretize(*ssm, 1.0 / L)
    K_original = K_conv(*ssm, L=L)

    K_restore = conv_from_gen(K_gen_inverse(*ssm, L=L), L)
    assert np.allclose(K_restore, K_original)
```

总结： 第一步我们实现了从**计算序列**到**计算生成函数**的转换转换过程：

$$\left[\bar{C}\bar{B}, \bar{C}\bar{A}\bar{B}, \ldots\right] \quad \longrightarrow \quad \sum_{i=0}^{L-1} \bar{C}\bar{A}^i\bar{B}z^i \tag{20}$$

生成函数通过引入 z 将矩阵序列压缩为一个函数，便于分析和高效计算，同时生成函数中的矩阵幂可以替换为矩阵的逆：

$$\sum_{i=0}^{L-1} \bar{C}\bar{A}^i\bar{B}z^i \quad \longrightarrow \quad \tilde{C}\Big(I - \bar{A}z\Big)^{-1}\bar{B} \tag{21}$$

但由于需要针对每一个单位根 $z \in \Omega_L$ 进行计算，因此矩阵的逆仍需计算 L 次。

第 2 步：对角矩阵简化情形

在这一部分，我们的目标是通过对矩阵 A 进行特殊结构假设，来优化逆矩阵的计算方法，使其更加高效：

我们从之前的公式开始：

$$\tilde{C}\Big(I - \bar{A}\Big)^{-1}\bar{B} \tag{22}$$

这是一个常见的矩阵求逆问题，其中 I 是单位矩阵，$\bar{A}$ 和 $\bar{B}$ 是给定的矩阵。直接计算这个逆矩阵可能非常耗时，尤其是当矩阵的规模很大时。

通过一些代数操作，可以将公式扩展为以下形式：

$$\tilde{C}\Big(I-\bar{A}\Big)^{-1}\bar{B}=\frac{1+z}{2\Delta}\tilde{C}\left[2\frac{1+z}{1-z}-\Delta A\right]^{-1}B \tag{23}$$

这里的主要变化是将矩阵求逆公式转换为依赖于参数 z 的形式，其中：

- z 是一个复变量，用于生成函数的分析。
- Δ 是一个常数，用于缩放。

这种表示形式的好处在于，它允许我们使用生成函数的工具来简化计算。

为了进一步简化，我们假设矩阵 A 是一个对角矩阵，用符号 Λ 表示。对角矩阵的特点是它的非对角项全为零，只有对角线上的元素非零，例如：

$$\Lambda=\begin{bmatrix}\lambda_1 & 0 & 0 & \cdots & 0\\ 0 & \lambda_2 & 0 & \cdots & 0\\ 0 & 0 & \lambda_3 & \cdots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & 0 & \cdots & \lambda_n\end{bmatrix} \tag{24}$$

对角矩阵的逆计算非常简单，只需要对每个对角线元素取倒数。

在假设 $A=\Lambda$ 后，作者证明生成函数可以被写成以下形式：

$$\hat{K}_{\Lambda}(z)=c(z)\sum_i\frac{\tilde{C}_iB_i}{g(z)-\Lambda_i} \tag{25}$$

这个公式的含义是：

- 我们将原本复杂的矩阵求逆问题，转化为对每个对角线元素 Λ_i 的逐项计算。
- $\tilde{C}_i$ 和 B_i 分别是矩阵 $\tilde{C}$ 和 B 中对应的分量。
- $g(z)$ 是一个关于 z 的函数，用来描述生成函数的行为。
- $c(z)$ 是一个缩放因子，通常是 z 的简单函数。

最终，我们可以将公式进一步简化为：

$$\hat{K}_{\Lambda}(z)=c(z)\cdot k_{z,\Lambda}(\tilde{C},B) \tag{26}$$

其中 $k_{z,\Lambda}(\tilde{C},B)$ 表示一个加权点积（weighted dot product）。

$$k_{z,\Lambda}(\tilde{C},B)=\sum_i\frac{\tilde{C}_iB_i}{g(z)-\Lambda_i} \tag{27}$$

这意味着：

- 我们不再需要直接计算整个矩阵的逆，而是通过简单的逐项计算完成任务。
- 每一项的计算只涉及标量运算（即对单个数值的加减乘除），这比矩阵运算要快得多。

到目前为止，通过引入对角矩阵假设（$\bar{A} = \Lambda$），我们将复杂的矩阵求逆问题替换为逐项计算的加权点积。这种方法显著降低了计算复杂度，同时保留了生成函数的灵活性和分析能力。

$$\tilde{C}\Big(I - \bar{A}\Big)^{-1}\bar{B} \quad \longrightarrow \quad c(z) \cdot \sum_i \frac{\tilde{C}_i B_i}{g(z) - \Lambda_i} \tag{28}$$

```
def cauchy_dot(v, omega, lambd):
    return (v / (omega - lambd)).sum()
```

尽管对于我们的实现并不重要，但值得注意的是，这是一个Cauchy核，并且是许多其他快速实现的主题。

第3步：对角矩阵加低秩项（Diagonal Plus Low-Rank）

在前两步中，我们假设矩阵 A 是一个对角矩阵 Λ。但在实际应用中，矩阵 A 通常并不是严格的对角矩阵，而是可以分解为对角矩阵加上一个低秩修正项的形式。我们将这种情况表示为：

$$A = \Lambda - PQ^* \tag{29}$$

其中：

- Λ：对角矩阵，表示矩阵的主要部分。
- $P, Q \in \mathbb{C}^{N \times 1}$：列向量，表示低秩修正项。
- Q^*：Q 的共轭转置。

对于这种形式的矩阵 $A = \Lambda - PQ^*$，直接求逆通常比较复杂。但幸运的是，我们可以使用 **Woodbury 公式** 来简化计算：

$$(\Lambda + PQ^*)^{-1} = \Lambda^{-1} - \Lambda^{-1}P\big(1 + Q^*\Lambda^{-1}P\big)^{-1}Q^*\Lambda^{-1} \tag{30}$$

1. Λ^{-1}：对角矩阵的逆，计算简单。
2. P：低秩修正项之一。
3. Q^*：低秩修正项的共轭转置。
4. $1 + Q^*\Lambda^{-1}P$：标量修正项，表示低秩部分对矩阵的影响。

通过 **Woodbury 公式**，我们将复杂的矩阵求逆问题分解为对角矩阵的求逆和低秩修正项的计算，具体推导：

1. **应用 Woodbury 公式**：

   $$(\Lambda + PQ^*)^{-1} = \Lambda^{-1} - \Lambda^{-1}P\big(1 + Q^*\Lambda^{-1}P\big)^{-1}Q^*\Lambda^{-1} \tag{31}$$

   将矩阵求逆分解为对角矩阵的求逆和低秩修正项。

2. **代入生成函数公式**：

   $$\tilde{C}(\Lambda + PQ^*)^{-1}\bar{B} = \tilde{C}\left(\Lambda^{-1} - \Lambda^{-1}P\big(1 + Q^*\Lambda^{-1}P\big)^{-1}Q^*\Lambda^{-1}\right)\bar{B} \tag{32}$$

3. **分解成加权点积形式**：

- 第一项：$\tilde{C}\Lambda^{-1}\bar{B} \to k_{z,\Lambda}(\tilde{C}, B)$
- 第二项：低秩修正部分，逐步展开为：$k_{z,\Lambda}(\tilde{C}, P) \cdot (1 + k_{z,\Lambda}(q^*, P))^{-1} \cdot k_{z,\Lambda}(q^*, B)$

经过推导，我们可以得到**生成函数的最终形式** $\hat{K}_{\text{DPLR}}(z)$：

$$\hat{K}_{\text{DPLR}}(z) = c(z)\left[k_{z,\Lambda}(\tilde{C}, B) - k_{z,\Lambda}(\tilde{C}, P) \cdot (1 + k_{z,\Lambda}(q^*, P))^{-1} \cdot k_{z,\Lambda}(q^*, B)\right] \tag{33}$$

1. $c(z)$：生成函数的缩放因子，表示整体的权重。
2. $k_{z,\Lambda}(\cdot,\cdot)$：加权点积运算，定义为：

$$k_{z,\Lambda}(X, Y) = \sum_i \frac{X_i Y_i}{g(z) - \Lambda_i} \tag{34}$$

   它是矩阵求逆问题的核心计算部分, 由刚才定义的`cauchy_dot` 计算

3. $\tilde{C}, B, P, Q^*$：分别表示矩阵的不同部分，用于描述矩阵的结构和修正项。

优化思路总结

- 第一步我们实现了从**计算序列**到**计算生成函数**的转换转换过程：

$$\left[\bar{C}\bar{B}, \bar{C}\bar{A}\bar{B}, \ldots\right] \quad \longrightarrow \quad \sum_{i=0}^{L-1} \bar{C}\bar{A}^i\bar{B}z^i \tag{35}$$

  并进一步生成函数通过引入 z 将矩阵序列压缩为一个函数，便于分析和高效计算，同时生成函数中的矩阵幂可以替换为矩阵的逆：

$$\sum_{i=0}^{L-1} \bar{C}\bar{A}^i\bar{B}z^i \quad \longrightarrow \quad \tilde{C}\Big(I - \bar{A}z\Big)^{-1}\bar{B} \tag{36}$$

  但由于需要针对每一个单位根 $z \in \Omega_L$ 进行计算，因此矩阵的逆仍需计算 L 次。

- 第二步通过引入对角矩阵假设（$\bar{A} = \Lambda$），我们将复杂的矩阵求逆问题替换为逐项计算的加权点积

$$\tilde{C}\Big(I - \bar{A}\Big)^{-1}\bar{B} \quad \longrightarrow \quad c(z) \cdot \sum_i \frac{\tilde{C}_i B_i}{g(z) - \Lambda_i} \tag{37}$$

- 第三步通过引入低秩修正项 PQ^*，我们能够在保持计算效率的同时，捕捉复杂矩阵的特性。结合 **Woodbury 公式** 和加权点积的计算方法，我们可以高效地求解生成函数，适用于大规模矩阵和复杂系统的分析。

$$\tilde{C}(\Lambda + PQ^*)^{-1}\bar{B} \quad \longrightarrow \quad c(z) \cdot \left[k_{z,\Lambda}(\tilde{C}, B) - k_{z,\Lambda}(\tilde{C}, P) \cdot (1 + k_{z,\Lambda}(q^*, P))^{-1} \cdot k_{z,\Lambda}(q^*, B)\right] \tag{38}$$

```
def K_gen_DPLR(Lambda, P, Q, B, C, step, unmat=False):
    aterm = (C.conj(), Q.conj())
```

```
    bterm = (B, P)

    def gen(o):
        g = (2.0 / step) * ((1.0 - o) / (1.0 + o))
        c = 2.0 / (1.0 + o)

        def k(a):
            # Checkpoint this calculation for memory efficiency.
            if unmat:
                return jax.remat(cauchy_dot)(a, g, Lambda)
            else:
                return cauchy_dot(a, g, Lambda)

        k00 = k(aterm[0] * bterm[0])
        k01 = k(aterm[0] * bterm[1])
        k10 = k(aterm[1] * bterm[0])
        k11 = k(aterm[1] * bterm[1])
        return c * (k00 - k01 * (1.0 / (1.0 + k11)) * k10)

    return gen
```

最终版本的 `K` 函数，因为 `conv_from_gen` 总是与生成函数（例如 `K_gen_DPLR`）一起调用，我们将它们合并，定义一个专用函数，用于从所有参数直接计算 **DPLR SSM 核**。
（通过减少间接调用的层级，这也可以让 **XLA 编译器** 更容易优化。）

```
@jax.jit
def cauchy(v, omega, lambd):
    """Cauchy matrix multiplication: (n), (l), (n) -> (l)"""
    cauchy_dot = lambda _omega: (v / (_omega - lambd)).sum()
    return jax.vmap(cauchy_dot)(omega)
def kernel_DPLR(Lambda, P, Q, B, C, step, L):
    # Evaluate at roots of unity
    # Generating function is (-)z-transform, so we evaluate at (-)root
    Omega_L = np.exp((-2j * np.pi) * (np.arange(L) / L))

    aterm = (C.conj(), Q.conj())
    bterm = (B, P)

    g = (2.0 / step) * ((1.0 - Omega_L) / (1.0 + Omega_L))
    c = 2.0 / (1.0 + Omega_L)

    # Reduction to core Cauchy kernel
    k00 = cauchy(aterm[0] * bterm[0], g, Lambda)
    k01 = cauchy(aterm[0] * bterm[1], g, Lambda)
```

```
    k10 = cauchy(aterm[1] * bterm[0], g, Lambda)
    k11 = cauchy(aterm[1] * bterm[1], g, Lambda)
    atRoots = c * (k00 - k01 * (1.0 / (1.0 + k11)) * k10)
    out = np.fft.ifft(atRoots, L).reshape(L)
    return out.real
```

现在我们可以检查它是否有效。首先，让我们生成一个随机的对角加低秩（DPLR）矩阵，

```
def random_DPLR(rng, N):
    l_r, p_r, q_r, b_r, c_r = jax.random.split(rng, 5)
    Lambda = jax.random.uniform(l_r, (N,))
    P = jax.random.uniform(p_r, (N,))
    Q = jax.random.uniform(q_r, (N,))
    B = jax.random.uniform(b_r, (N, 1))
    C = jax.random.uniform(c_r, (1, N))
    return Lambda, P, Q, B, C
```

我们可以验证，**DPLR 方法**生成的滤波器与直接计算 A 的结果相同。

```
def test_gen_dplr(L=16, N=4):
    I = np.eye(4)

    # Create a DPLR A matrix and discretize
    Lambda, P, B, _ = make_DPLR_HiPPO(N)
    A = np.diag(Lambda) - P[:, np.newaxis] @ P[:, np.newaxis].conj().T
    _, _, C = random_SSM(rng, N)

    Ab, Bb, Cb = discretize(A, B, C, 1.0 / L)
    a = K_conv(Ab, Bb, Cb.conj(), L=L)

    # Compare to the DPLR generating function approach.
    C = (I - matrix_power(Ab, L)).conj().T @ Cb.ravel()
    b = kernel_DPLR(Lambda, P, P, B, C, step=1.0 / L, L=L)
    assert np.allclose(a.real, b.real)
```

对角加低秩 RNN

DPLR 分解的一个次要好处是，它使我们能够在不直接求逆矩阵 A 的情况下计算 SSM 的离散化形式。在这里，我们回到论文中查看其推导过程。

回忆一下，离散化计算如下：

$$\bar{A} = (I - \frac{\Delta}{2} \cdot A)^{-1}(I + \frac{\Delta}{2} \cdot A) \tag{39}$$

$$\bar{B} = (I - \frac{\Delta}{2} \cdot A)^{-1} \Delta B \tag{40}$$

我们将分别简化 $\bar{A}$ 的定义中的两个部分。

第一部分简化

$$I + \frac{\Delta}{2} A = I + \frac{\Delta}{2}(\Lambda - PQ^*) \tag{41}$$

$$= \frac{\Delta}{2}\left[\frac{2}{\Delta} I + (\Lambda - PQ^*)\right] \tag{42}$$

$$= \frac{\Delta}{2} A_0 \tag{43}$$

其中，A_0 定义为最终括号中的项。

第二部分简化：虽然通常情况下这个逆项很难处理，但在 ***DPLR*** *的情况下，我们可以使用* ***Woodbury 恒等式****来简化，如下所示：*

$$(I - \frac{\Delta}{2} A)^{-1} = (I - \frac{\Delta}{2}(\Lambda - PQ^*))^{-1} \tag{44}$$

$$= \frac{2}{\Delta}\left[\frac{\Delta}{2} I - \Lambda + PQ^*\right]^{-1} \tag{45}$$

$$= \frac{2}{\Delta}\left[D - DP(1 + Q^* DP)^{-1} Q^* D\right] \tag{46}$$

$$= \frac{2}{\Delta} A_1 \tag{47}$$

其中：

$$D = (\frac{2}{\Delta} - \Lambda)^{-1} \tag{48}$$

A_1 定义为最终括号中的项。

最终，离散时间状态空间模型变为：

$$x_k = \bar{A} x_{k-1} + \bar{B} u_k \tag{49}$$

$$= A_1 A_0 x_{k-1} + 2 A_1 B u_k \tag{50}$$

$$y_k = C x_k \tag{51}$$

通过以上推导，我们可以有效地在 DPLR 框架下简化离散化计算，从而避免直接求逆矩阵 A。

```
def discrete_DPLR(Lambda, P, Q, B, C, step, L):
    # Convert parameters to matrices
    B = B[:, np.newaxis]
    Ct = C[np.newaxis, :]
```

```
    N = Lambda.shape[0]
    A = np.diag(Lambda) - P[:, np.newaxis] @ Q[:, np.newaxis].conj().T
    I = np.eye(N)

    # Forward Euler
    A0 = (2.0 / step) * I + A

    # Backward Euler
    D = np.diag(1.0 / ((2.0 / step) - Lambda))
    Qc = Q.conj().T.reshape(1, -1)
    P2 = P.reshape(-1, 1)
    A1 = D - (D @ P2 * (1.0 / (1 + (Qc @ D @ P2))) * Qc @ D)

    # A bar and B bar
    Ab = A1 @ A0
    Bb = 2 * A1 @ B

    # Recover Cbar from Ct
    Cb = Ct @ inv(I - matrix_power(Ab, L)).conj()
    return Ab, Bb, Cb.conj()
```

将 HiPPO 转换为 DPLR

这种方法主要适用于 **DPLR 矩阵**（对角加低秩矩阵），但我们希望它也能同样适用于 **HiPPO 矩阵**。虽然 HiPPO 矩阵在其当前形式下并不是标准的 DPLR 矩阵，但它有一个非常重要的特性：它属于 **NPLR 矩阵**（正规加低秩矩阵）。那么，NPLR 矩阵又是什么呢?

- **正规矩阵**（Normal Matrices）：正规矩阵是指满足 $AA^* = A^*A$ 的矩阵，其中 A^* 是 A 的共轭转置。一个重要的性质是，正规矩阵总是可以被**酉矩阵**对角化（也就是说，可以通过一个特殊的单位矩阵将其转化为对角矩阵）。
- **低秩矩阵**（Low-Rank Matrices）：低秩矩阵是指其秩（矩阵的非零特征值数量）远小于其维度的矩阵。这种矩阵非常稀疏，计算起来通常更高效。

因此，**NPLR 矩阵**可以理解为一个“正规矩阵”和一个“低秩矩阵”的组合。

DPLR 矩阵是指“对角矩阵加低秩矩阵”，而 NPLR 矩阵是“正规矩阵加低秩矩阵”。由于正规矩阵总是可以通过酉变换对角化，因此从数学上来看，NPLR 矩阵和 DPLR 矩阵在一定意义上是等价的。换句话说，虽然它们的定义不同，但它们都可以被转化为类似的形式，因此在状态空间模型（SSM）中，它们的作用是一样的。

因此，尽管 HiPPO 矩阵不是严格意义上的 DPLR 矩阵，但由于它属于 NPLR 矩阵类别，而 NPLR 矩阵和 DPLR 矩阵在 SSM 模型中的作用本质上是等价的，所以我们可以将 HiPPO 矩阵视为“几乎是 DPLR 矩阵”。这意味着我们可以使用类似的方法来处理 HiPPO 矩阵，从而高效地构建和训练 SSM 网络。

S4 技术可以应用于任何可以分解为 ***Normal Plus Low-Rank (NPLR)*** *形式的矩阵 A。具体来说，矩阵 A 可以表示为：*

$$A = V\Lambda V^* - PQ^\top = V\big(\Lambda - V^*P(V^*Q)^*\big)V^* \tag{52}$$

其中：

- $V \in \mathbb{C}^{N\times N}$ *是一个酉矩阵（单位矩阵），满足* $V^*V = I$。
- Λ *是对角矩阵。*
- $P, Q \in \mathbb{R}^{N\times r}$ *是低秩分解中的两个矩阵，秩* r *远小于矩阵的维度* N。

因此，一个 NPLR 状态空间模型（SSM）在数学上与某种 DPLR 矩阵是酉等价的（unitarily equivalent）。这意味着，从 SSM 的角度来看，NPLR 矩阵可以被处理得像 DPLR 矩阵一样高效。

在 S4 中，我们需要将矩阵 A 替换为一个 HiPPO 矩阵。为此，首先需要将 HiPPO 矩阵表示为“正规矩阵加低秩项”（Normal Plus Low-Rank, NPLR）的形式，然后对其进行对角化以从分解中提取 Λ。

论文附录展示了如何实现这一点：通过将正规部分写为一个**斜对称矩阵**（skew-symmetric matrix，加上一个常数乘以单位矩阵），而斜对称矩阵是一类特殊的正规矩阵。

此外，还有一个额外的简化：实际上存在一种表示形式，可以将低秩部分的两个分量约束为相等，即 $P = Q$。后续研究表明，这种约束对于模型的稳定性非常重要。

```
def make_NPLR_HiPPO(N):
    # Make -HiPPO
    nhippo = make_HiPPO(N)

    # Add in a rank 1 term. Makes it Normal.
    P = np.sqrt(np.arange(N) + 0.5)

    # HiPPO also specifies the B matrix
    B = np.sqrt(2 * np.arange(N) + 1.0)
    return nhippo, P, B
```

在提取出矩阵的正规部分后，我们可以对其进行对角化，从而得到 DPLR 的分解形式。由于正规部分实际上是一个**斜对称矩阵**，我们可以分别提取 Λ 的实部和虚部。这种方法具有以下两个重要作用：

1. **更细粒度的控制**
   通过分别控制实部和虚部，我们可以更好地优化模型的稳定性。这种灵活性在状态空间模型中尤为重要，尤其是在长时间序列建模时。
2. **高效的对角化算法**
   这种分解方式允许我们使用更强大的针对 Hermitian 矩阵（厄米矩阵）的对角化算法。事实上，目前 JAX 的实现并不支持在 GPU 上对非 Hermitian 矩阵进行对角化，而 Hermitian 矩阵的对角化在 GPU 上是支持且高效的。这使得这种方法在实际应用中更加实用。

```python
def make_DPLR_HiPPO(N):
    """Diagonalize NPLR representation"""
    A, P, B = make_NPLR_HiPPO(N)

    S = A + P[:, np.newaxis] * P[np.newaxis, :]

    # Check skew symmetry
    S_diag = np.diagonal(S)
    Lambda_real = np.mean(S_diag) * np.ones_like(S_diag)
    # assert np.allclose(Lambda_real, S_diag, atol=1e-3)

    # Diagonalize S to V \Lambda V^*
    Lambda_imag, V = eigh(S * -1j)

    P = V.conj().T @ P
    B = V.conj().T @ B
    return Lambda_real + 1j * Lambda_imag, P, B, V
```

这段代码的目的是将 HiPPO 矩阵转化为 **Diagonal Plus Low-Rank (DPLR)** 的形式。DPLR 是 S4 模型的核心数学基础，因为它可以高效地对矩阵进行分解和操作。

1. **生成 NPLR 表示**

   ```python
   A, P, B = make_NPLR_HiPPO(N)
   ```

   - `make_NPLR_HiPPO(N)` 是一个辅助函数，用于生成 HiPPO 矩阵的 **Normal Plus Low-Rank (NPLR)** 表示：

     $$A + PP^\top \tag{53}$$

     - A：是 HiPPO 矩阵的正规部分，通常是一个斜对称矩阵。
     - P：是低秩部分的向量。
     - B：是输入向量（通常用于控制输入到状态空间模型的信号）。

2. **构造矩阵 S**

   ```python
   S = A + P[:, np.newaxis] * P[np.newaxis, :]
   ```

   - S 是 HiPPO 矩阵的完整形式，由正规部分 A 和低秩部分 $PP^\top$ 组成：

     $$S = A + PP^\top \tag{54}$$

   - 这里，`P[:, np.newaxis]` 和 `P[np.newaxis, :]` 将向量 P 转换为列向量和行向量，从而实现矩阵乘积。

3. **检查斜对称性**

```
S_diag = np.diagonal(S)
Lambda_real = np.mean(S_diag) * np.ones_like(S_diag)
```

- 斜对称矩阵的特性是对角线元素为零。然而，由于数值误差，可能会出现微小偏差。
- 这里通过计算对角线元素的平均值 $\text{mean}(S_{\text{diag}})$ 来近似对角线元素的值，并将其视为 Λ_{real} 的实部。

4. **对 S 进行对角化**

```
Lambda_imag, V = eigh(S * -1j)
```

- S 是一个斜对称矩阵，通过乘以 $-1j$，将其转换为 Hermitian 矩阵（厄米矩阵）。这是因为 Hermitian 矩阵的特征值总是实数，便于高效计算。
- `eigh()` 是一个专门用于 Hermitian 矩阵的对角化函数，返回特征值 Λ_{imag} 和特征向量矩阵 V：

$$S \cdot (-i) = V \Lambda_{\text{imag}} V^* \tag{55}$$

5. **更新低秩部分 P 和输入向量 B**

```
P = V.conj().T @ P
B = V.conj().T @ B
```

- 通过特征向量矩阵 V 将低秩部分 P 和输入向量 B 转换到新的特征空间中：

$$P' = V^* P, \quad B' = V^* B \tag{56}$$

- 这里 V^* 是 V 的共轭转置。

6. **返回结果**

```
return Lambda_real + 1j * Lambda_imag, P, B, V
```

- 返回值包括：
  - $\Lambda = \Lambda_{\text{real}} + i\Lambda_{\text{imag}}$：矩阵的特征值分解结果。
  - P：更新后的低秩部分。
  - B：更新后的输入向量。
  - V：特征向量矩阵。

通过对矩阵进行对角化，我们可以更高效地处理长时间序列建模任务，同时保持数值稳定性。这种方法结合了数学理论与工程实践，是 S4 模型的关键步骤。

最后，进行sanity check 以确保这些恒等式成立，

```
def test_nplr(N=8):
    A2, P, B = make_NPLR_HiPPO(N)
    Lambda, Pc, Bc, V = make_DPLR_HiPPO(N)
    Vc = V.conj().T
    P = P[:, np.newaxis]
    Pc = Pc[:, np.newaxis]
    Lambda = np.diag(Lambda)

    A3 = V @ Lambda @ Vc - (P @ P.T)  # Test NPLR
    A4 = V @ (Lambda - Pc @ Pc.conj().T) @ Vc  # Test DPLR
    assert np.allclose(A2, A3, atol=1e-4, rtol=1e-4)
    assert np.allclose(A2, A4, atol=1e-4, rtol=1e-4)
```

最终检查

测试所有功能是否按计划工作。

```
def test_conversion(N=8, L=16):
    step = 1.0 / L
    # Compute a HiPPO NPLR matrix.
    Lambda, P, B, _ = make_DPLR_HiPPO(N)
    # Random complex Ct
    C = normal(dtype=np.complex64)(rng, (N,))

    # CNN form.
    K = kernel_DPLR(Lambda, P, P, B, C, step, L)

    # RNN form.
    Ab, Bb, Cb = discrete_DPLR(Lambda, P, P, B, C, step, L)
    K2 = K_conv(Ab, Bb, Cb, L=L)
    assert np.allclose(K.real, K2.real, atol=1e-5, rtol=1e-5)

    # Apply CNN
    u = np.arange(L) * 1.0
    y1 = causal_convolution(u, K.real)

    # Apply RNN
    _, y2 = scan_SSM(
        Ab, Bb, Cb, u[:, np.newaxis], np.zeros((N,)).astype(np.complex64)
    )
    assert np.allclose(y1, y2.reshape(-1).real, atol=1e-4, rtol=1e-4)
```