Advances in Geosciences 地球科学前沿, 2026, 16(4), 548-557
Published Online April 2026 in Hans. https://www.hanspub.org/journal/ag
https://doi.org/10.12677/ag.2026.164049

Hans汉斯

基于Cython混合编程的电阻率法正演优化

李文海，陈汉波*，吴彬海，李明晟，王立辉

桂林理工大学地球科学学院，广西 桂林

收稿日期：2026年3月20日；录用日期：2026年4月15日；发布日期：2026年4月27日

摘　要

本文以纯Python实现的电阻率法有限元正演算法为研究对象，开展了一套从性能剖析到靶向优化的混合编程实践研究。通过性能剖析发现，在中小规模网格下，矩阵组装耗时占总计算时间的75%以上，是程序的主要性能瓶颈。针对这一瓶颈，对全局刚度矩阵组装(含单元计算)模块进行Cython静态类型重写，显著提升了代码执行效率。实验结果表明，优化后的算法在保证计算精度的前提下，矩阵组装模块获得最高62倍的加速比，将Python解释器在热点函数中的开销降至可忽略水平。在典型测试模型下，矩阵组装模块获得53倍加速比，程序整体执行效率提升约6.4倍。

关键词

有限单元法，混合编程，Cython优化，性能分析

Resistivity Forward Modeling Optimization Based on Cython Hybrid Programming

Wenhai Li, Hanbo Chen*, Binhai Wu, Mingsheng Li, Lihui Wang

College of Earth Science, Guilin University of Technology, Guilin Guangxi

Received: March 20, 2026; accepted: April 15, 2026; published: April 27, 2026

Abstract

This paper presents a case study of hybrid programming for a pure Python-implemented finite element forward modeling algorithm for the resistivity method, spanning from performance profiling to targeted optimization. Through performance profiling, it was found that for small to medium-scale meshes, the matrix assembly process accounts for over 75% of the total computation time, making it the primary performance bottleneck of the program. To address this, the global stiffness matrix assembly module (including element calculations) was rewritten with Cython static typing,

*通讯作者。

文章引用: 李文海, 陈汉波, 吴彬海, 李明晟, 王立辉. 基于 Cython 混合编程的电阻率法正演优化[J]. 地球科学前沿, 2026, 16(4): 548-557. DOI: 10.12677/ag.2026.164049

significantly enhancing code execution efficiency. Experimental results demonstrate that the optimized algorithm achieves a speedup of up to 62x for the matrix assembly module while maintaining computational accuracy, reducing the overhead of the Python interpreter in hot functions to negligible levels. Under the typical test model, the matrix assembly module achieves a 53x speedup, and the overall program execution efficiency improves by approximately 6.4x.

Keywords

Finite Element Method, Hybrid Programming, Cython Optimization, Performance Analysis

1. 引言

电阻率法正演是地球物理精确反演和解释的重要基础[1]，但在大规模模型条件下，其数值计算往往面临计算规模大、耗时长等现实挑战[2]。在算法实现层面，开发效率与执行效率之间长期存在难以兼顾的矛盾。传统高性能计算多采用 C/C++、Fortran 等编译型语言，虽可获得较高的执行效率，但开发周期长、调试成本高，不利于算法的快速迭代与验证[3]。而 Python 凭借其简洁的语法和成熟的科学计算生态，已成为数值算法原型开发的重要工具[4]，但其解释执行和动态类型机制导致核心计算环节(如单元遍历和矩阵组装)效率低下，成为制约其工程应用的主要瓶颈。

Cython 作为一种混合编程工具，为缓解上述问题提供了一种可行途径。通过在 Python 代码中引入静态类型声明并编译为 C 扩展模块，Cython 能够在保留 Python 开发效率的同时显著降低解释执行开销[5]。基于此，本文以一套纯 Python 实现的电阻率法有限元正演程序为研究对象[6]，围绕其计算效率问题开展针对性优化研究，探索在保持原有算法结构与数值精度不变的前提下，提高正演程序整体执行效率的实现路径。

2. 基于 Green 函数的正演方法及其计算瓶颈分析

2.1. Green 函数法理论基础

利用有限单元法开展电阻率法正演，直接求解总电位的传统方法(总场法)面临源点奇异性的数值困难[7]。为解决此问题，Green 函数法被提出并证明是一种高精度的替代方案，该方法的核心在于引入满足 Poisson 方程 $\nabla^2 G = -\delta$ 的 Green 函数[8]，将含奇异源项的边界问题转化为一个更适于数值求解的等价形式。

对于位于地表 A， B 两点的双点电流源，传统的总场法边值问题[9]可表述为

$$\begin{cases} \nabla\left(\sigma\nabla V\right) = -2I\delta\left(A\right) + 2I\delta\left(B\right), & \in \Omega \\ \dfrac{\partial v}{\partial n} = 0, & \in \Gamma_s \\ \dfrac{\partial v}{\partial n} + \dfrac{1}{r_B - r_A}\left(\dfrac{r_B\cos\left(r_A, n\right)}{r_A} - \dfrac{r_A\cos\left(r_B, n\right)}{r_B}\right)v = 0, & \in \Gamma_\infty \end{cases} \tag{1}$$

其中，σ 是介质的电导率，I 是供电电流，$\delta(A)$和$\delta(B)$分别表示以电源点 A 和 B 为中心的 δ 函数，Γ_s 和Γ_∞ 分别表示地表边界和无穷远边界，n 为边界的外法向，r_A，r_B 分别是测点至电源点 A，B 的距离。

在 Green 函数法[10]中，引入函数 $G(r)=\frac{1}{4\pi}\left(\frac{1}{r_A}-\frac{1}{r_B}\right)$，并利用其性质 $\nabla^2 G=-\delta(A)+\delta(B)$，极大地削弱点电流源邻近位函数的奇异性，得到如下等效边值问题

$$\begin{cases}\nabla(\sigma\nabla V)=2I\nabla^2 G, & \in\Omega\\ \dfrac{\partial v}{\partial n}=0, & \in\Gamma_s\\ \dfrac{\partial v}{\partial n}+\dfrac{1}{r_B-r_A}\left(\dfrac{r_B\cos(r_A,n)}{r_A}-\dfrac{r_A\cos(r_B,n)}{r_B}\right)v=0, & \in\Gamma_\infty\end{cases}\tag{2}$$

利用有限单元法计算方程(2)所得到的结果其精度足以与异常电位法相媲美。本文的正演算法即基于此 Green 函数法构建。

基于上述边值问题式(2)，采用规则六面体对求解区域进行空间离散，并利用有限单元法进行数值求解。其详细推导过程、单元矩阵计算公式及完整的算法框架，可参见文献[10]。该文已验证此方法的计算精度。

2.2. 纯 Python 算法性能瓶颈分析

然而，在纯 Python 环境中直接实现“该文”所述算法框架时，其数值计算效率面临严峻瓶颈。尽管该算法采用解析型单元刚度矩阵公式(避免了传统有限元中的高斯数值积分)，但其仍需对海量单元进行循环遍历以计算并组装全局矩阵。这一过程中涉及的频繁 Python 对象操作与稀疏矩阵索引管理，在 Python 解释器环境下效率低下，成为制约算法应用于大规模三维模型的主要性能瓶颈。

为定量识别并精确量化上述瓶颈，我们基于 Python 实现了完整的 Green 函数法三维正演程序，并采用 cProfile 模块进行精细化性能剖析。测试选用一个典型的三层层状地电模型，其网格剖分规模为 $n_x\times n_y\times n_z=93\times61\times33$。计算平台配置为 AMD 7500F 处理器(6 核心 12 线程)，32 GB 内存，运行 Python3.11、NumPy2.41、SciPy1.16 以及 Pypardiso0.4.7 环境。

表 1 展示了该模型单次计算中耗时最长的函数，结果显示，程序的计算耗时高度集中于少数几个函数。

Table 1. Key function performance profiling of pure Python version
表 1. 纯 Python 版本关键函数性能剖析

耗时排名	函数层级	累计耗时	耗时占比
1	矩阵组装 B	11.47	39.8
2	矩阵组装 A	11.31	39.3
3	方程组求解	6.04	20.9
4	总计	28.82	100

分析表 1 可以得知，全局矩阵的组装，占据总计算时间的 79.1%，是程序的主要性能瓶颈。通过细粒度性能剖析进一步发现，在上述组装函数内部，单元计算耗时约占 24.3%。这一占比揭示了一个关键事实：性能瓶颈主要不在于计算本身，而在于 Python 层面的数据组织与管理开销。如图 1 所示，这些开销

主要体现为：(1) 为每个单元创建 Python 列表对象以存储节点坐标；(2) 执行约 4297 万次 list.append()操作构建稀疏矩阵三元组；(3) 频繁的 Python-C 数据转换。这些操作在解释器中的执行效率极低，但均具备通过静态编译与并行化优化的潜力。

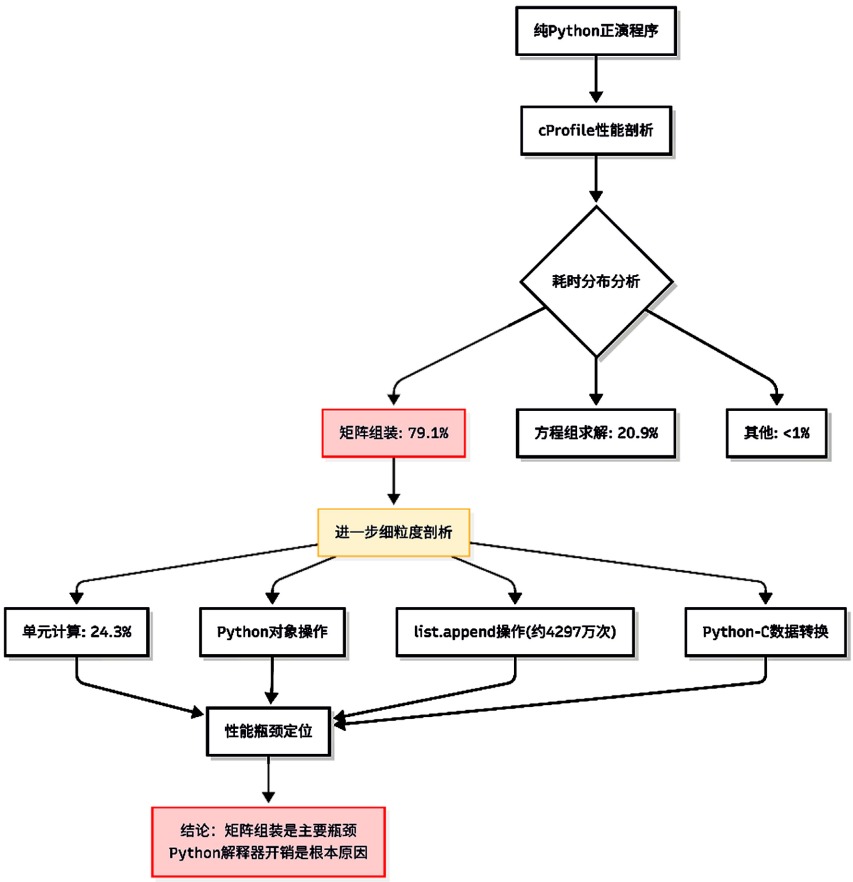

Figure 1. Performance profiling flowchart
图 1. 性能剖析流程图

3. 基于 Cython 的关键模块优化方法

3.1. 优化总体架构

基于第 2.2 节的性能剖析结果，本文提出靶向式混合编程优化策略。其核心思想是：保持算法高层逻辑与调用接口不变，仅对识别出的性能热点函数进行底层重构，在保证数值精度和代码可维护性的前提下，最大限度地提升执行效率。

在技术选型上，我们评估了主流的 Python 高性能计算方案。包括 Numba、PyBind11 和 Cython，从开发效率、性能潜力、渐进式优化能力三个维度进行综合权衡。

Numba 作为即时编译方案，能通过装饰器自动将 Python 函数编译为机器码，开发最为便捷[11]。然而，Numba 对复杂控制流和稀疏矩阵操作的支持有限，难以优化本算法中边界条件密集的 k2e 函数及大规模稀疏矩阵组装过程。相较之下，Cython 允许开发者精确控制内存布局和类型系统，更适合本算法中矩阵组装模块的优化需求。

- PyBind11 作为 C++与 Python 的绑定工具，能够将 C++编写的核心函数导出为 Python 模块[12]，理论

上可获得极致性能。但是该方案要求开发者具备深入的 C++知识，需要编写独立的 C++扩展模块并配置复杂的编译环境，与 Python 科学计算生态的集成成本较高，不符合本项目“渐进式优化”的需求。

- Cython 因其在静态编译优化与 Python 生态无缝集成之间的独特平衡而被最终采用。Cython 允许在 Python 代码中逐步引入静态类型声明，通过 cdef 关键字将热点函数编译为 C 扩展模块，在不改变原有调用接口的前提下显著降低解释器开销。相较于 Numba 和 PyBind11，Cython 提供了从解释执行到编译执行的平滑迁移路径，符合渐进式需求。

3.2. 热点函数重构

基于 3.1 节的总体架构，本节对识别出来的两个核心性能热点，单元刚度矩阵计算和矩阵组装内存分配进行具体的 Cython 重构。

3.2.1. 单元刚度矩阵计算的 Cython 重构

单元刚度矩阵计算函数 KE.k1e 是整个有限元正演中最基础的数值计算单元，在典型网格规模(93 × 61 × 33)下被调用约 36 万次。该函数根据单元 8 个节点的坐标和电导率，计算 8 × 8 的单元刚度矩阵。性能剖析表明，该函数的主要开销来自 Python 层的对象操作：(1) 每次调用需从 Python 列表读取 8 个节点的坐标，涉及 24 次 Python 对象访问；(2) abs 等数学函数调用在 Python 层执行；(3) np.array 构造导致临时数组创建。这些 Python 层操作占该函数总执行时间的 75%以上，而真正的浮点运算占比不足 25%。

针对上述问题，我们将该函数重构为 C 级别的纯数值计算函数。重构方案包括：

(1) 静态类型声明：使用 Cython 的 cdef 关键字声明所有变量为静态类型，消除 Python 动态类型检查的开销。

(2) C 数组存储节点坐标：将 Python 列表传入的节点坐标转换为 C 语言二维数组，后续所有坐标访问均通过 C 数组直接寻址，避免 Python 对象访问。

(3) C 数学函数替代：将 Python 的 abs 函数替换为 C 标准库的 fabs 函数，函数调用开销从 Python 级别降至 C 级别。

(4) 释放全局解释器锁：使用 nogil 声明，使该函数在执行期间释放 GIL，为后续多线程并行调用预留接口。

优化后的函数将 36 万次调用完全置于 C 级别执行，Python 解释器仅在函数调用入口处产生一次开销，内部所有运算均在 C 环境中完成。这一改进从根本上消除了原版本中的 Python 对象操作开销。

3.2.2. 边界条件函数 k2e 的向量化优化

边界条件处理函数 KE.k2e 涉及 6 种不同类型的边界面元(对应单元的不同面)，用于计算边界条件对单元刚度矩阵的贡献。原实现存在以下性能问题：(1) 每次调用均需重新计算单元尺寸和公共系数；(2) 6 种面类型的处理采用 Python 条件分支，每次调用均需进行多次分支判断。

针对上述问题，我们采用以下优化策略：

(1) 公共计算外提：将单元尺寸的计算从各分支中提取到函数入口处，所有面类型共享计算结果，避免重复计算。

(2) C 级别条件分支：将 6 种面类型的计算统一实现在 C 函数中，利用 C 语言的 if-else 结构替代 Python 的条件分支，消除 Python 解释器的分支预测开销。

(3) 合并内存访问：将面中心坐标计算与矩阵填充合并，减少中间变量的创建。

优化后的 k2e 函数与 k1e 函数一样，完全在 C 级别执行，每单元调用次数与边界条件数量相关(通常

2~4 次)，整体计算效率提升显著。

3.2.3. 矩阵组装的内存分配优化

性能剖析显示，原 Python 版本在矩阵组装过程中执行了约 4297 万次 list.append()操作，用于构建稀疏矩阵的三元组(行索引、列索引、非零元素值)。这些三元组最终用于创建 SciPy 的 CSC 格式稀疏矩阵。原实现采用 Python 列表动态追加元素的方式：

外层循环遍历全部 36 万个单元、中层循环遍历单元矩阵的 8 行、内层循环遍历下三角的 36 个非零元，这种动态内存分配方式在 Python 解释器中效率极低，每次 append 操作都涉及列表扩容和内存重新分配，且随着列表规模增大，内存拷贝开销呈指数级增长。

针对这一问题，我们采用预分配数组策略进行优化。由于有限元网格的拓扑结构已知，每个单元贡献的非零元数量是固定的(下三角 36 个)，因此可以预先计算出全局稀疏矩阵的最大非零元数量(单元数 × 36)。基于这一先验信息，优化方案包括：

预分配固定大小数组：在组装开始前，一次性分配三个 NumPy 数组，分别存储行索引、列索引和非零元素值，数组大小设为 ne × 36。

索引指针填充：使用整型变量 idx 记录当前填充位置，每处理一个非零元，直接对数组元素赋值，并递增 idx。

C 级别数组访问：通过 Cython 的内存视图(memory view)直接操作 NumPy 数组的底层数据缓冲区，所有数组赋值均在 C 级别完成。

该优化方案将 4297 万次 list.append()操作彻底消除，代之以 C 级别的数组索引赋值。预分配的 NumPy 数组在内存中连续存储，访问效率远高于 Python 列表。同时，配合 3.2.1 和 3.2.2 节的 C 函数，整个矩阵组装过程仅在 Python 层保留最外层的单元循环控制流，核心数值计算和数组填充均在 C 级别执行。

4. 数值实验

4.1. 精度验证

为了验证 Cython 的并行优化不会带来精度上的变化，首先设计一例各向同性半空间地电模型，通过与纯 Python 结果及解析解对比验证精度。具体计算参数为：半空间介质电导率为 0.01 $S\cdot m^{-1}$，供电电流为 10 A，在有限单元法正演中，沿 x、y、z 方向将网格剖分为 50 × 42 × 24。

由表 2 中列出的计算结果可以看出：即便在小极距条件下，利用 Green 函数法得到的点电流源总电位与解析解的计算精度依然相当，而利用 Cython 优化后的算法并未改变精度。

Table 2. Accuracy verification of Cython optimized version
表 2. Cython 优化版本精度验证

供电极距(m)	解析解(V)	纯 Python 版(V)	Cython 优化版(V)
0.0	0.000000	0.127323×10^{-13}	0.127323×10^{-13}
1.0	0.212201×10^{3}	0.212201×10^{3}	0.212201×10^{3}
2.0	0.424409×10^{2}	0.424409×10^{2}	0.424409×10^{2}
3.0	0.181890×10^{2}	0.181890×10^{2}	0.181890×10^{2}
4.0	0.101050×10^{2}	0.101050×10^{2}	0.101050×10^{2}
5.0	0.643048×10^{1}	0.643048×10^{1}	0.643048×10^{1}
6.0	0.445187×10^{1}	0.445187×10^{1}	0.445187×10^{1}

续表

7.0	0.326471×10^{1}	0.326471×10^{1}	0.326471×10^{1}
8.0	0.249654×10^{1}	0.249654×10^{1}	0.249654×10^{1}
9.0	0.197095×10^{1}	0.197095×10^{1}	0.197095×10^{1}
⋮	⋮	⋮	⋮

4.2. 计算性能对比分析

为定量评估 Cython 优化策略的实际加速效果，本节首先针对 2.2 节中用于瓶颈分析的地电模型进行详细的性能剖析，随后选取五组不同规模的网格验证优化方法的普适性。

4.2.1. 典型模型性能分析

选用 2.2 节中应用的网格规模做详细性能对比。表 3 汇总了纯 Python 版本与 Cython 优化版本在该模型下的核心计算模块耗时，图 2 直观展示了各模块的耗时对比。

Table 3. Comparison of core computation module time consumption for typical model
表 3. 典型模型核心计算模块耗时对比

函数模块	纯 Python 版(s)	Cython 优化版(s)	加速比
矩阵组装 B	11.47	0.26	44.12
矩阵组装 A	11.31	0.17	66.53
方程组求解	6.04	4.10	1.47
总计	28.82	4.53	6.36

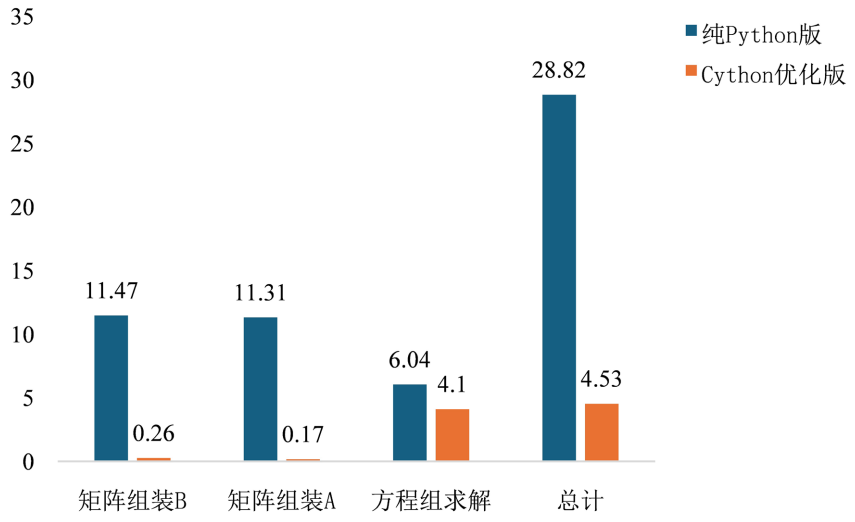

Figure 2. Comparison of core computation module time consumption
图 2. 核心计算模块耗时对比图

从表 3 和图 2 可以看出，优化效果主要体现在矩阵组装模块。其中，矩阵组装 A 的耗时从 11.31 秒降至 0.17 秒，加速比达 66.53 倍；矩阵组装 B 的耗时从 11.47 秒降至 0.26 秒，加速比达 44.12 倍。两者合计，矩阵组装模块总耗时从 22.78 秒降至 0.43 秒，整体加速比达 53.0 倍。相比之下，方程组求解模块

的耗时仅从 6.04 秒降至 4.10 秒，加速比为 1.47 倍，提升幅度有限。

这一结果验证了靶向优化策略的有效性：仅对矩阵组装模块进行 Cython 重构，即可获得超过 50 倍的加速效果。值得注意的是，求解器模块采用的是 Pypardiso——目前 Python 生态中性能领先的稀疏直接求解器之一[13]，其在所有测试中均保持稳定高效的求解性能。

4.2.2. 不同网格规模下的加速效果

为验证优化方法在不同计算规模下的普适性，选取五组具有代表性的网格进行测试，表 4 汇总了五组网格的测试结果，图 3 展示了加速比随网格规模的变化趋势。

Table 4. Matrix optimization performance under different mesh scales
表 4. 不同网格规模下的矩阵优化效果

网格规模	纯 Python 版(s)	Cython 优化版(s)	加速比
63 × 41 × 21	5.79	0.10	57.90
93 × 61 × 33	22.88	0.43	53.21
115 × 71 × 36	34.90	0.56	62.32
141 × 83 × 41	55.50	0.93	59.68
161 × 101 × 51	95.25	1.65	57.73

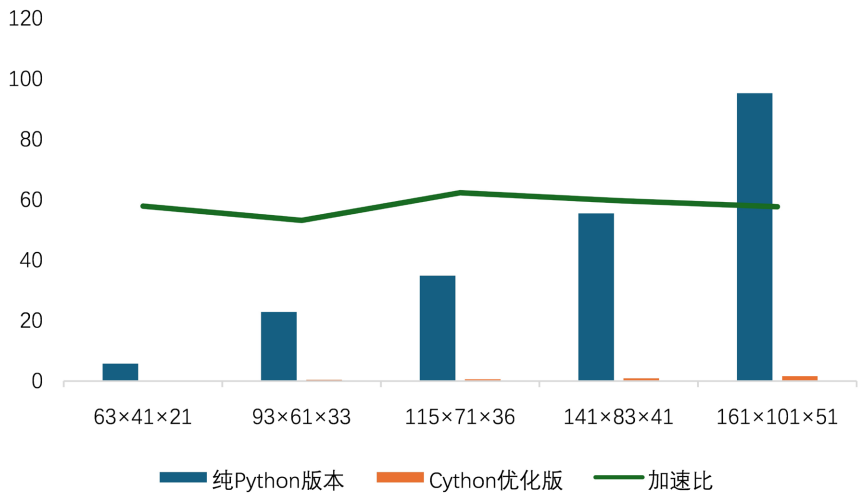

Figure 3. Speedup variation trend with mesh scale
图 3. 加速比随网格规模的变化趋势

从表 4 和图 3 可以看出，矩阵组装模块的加速比在不同网格规模下均保持在较高水平。五组网格的加速比分布在 53~62 倍之间，未出现随规模增大而明显下降的趋势。这表明 Cython 优化方法具有良好的稳定性，其加速效果不受问题规模影响。

为进一步分析优化后各模块的耗时分布，表 5 展示了五组网格规模下矩阵组装与方程组求解的耗时对比。可以看出，随着网格规模增大，方程组求解耗时的增长速度明显快于矩阵组装模块。当网格规模从 63 × 41 × 21 增至 161 × 101 × 51 时，求解器耗时从 0.60 秒增至 66.07 秒(增长 110 倍)，而矩阵组装耗时仅从 0.10 秒增至 1.65 秒(增长 16.5 倍)。求解器耗时占比从 85.7%上升至 97.6%，表明在超大规模型问题中，求解器已成为决定性性能瓶颈。

Table 5. Time consumption comparison of matrix assembly and solver under different mesh scales
表 5. 不同网格规模下矩阵组装与方程组求解耗时对比

网格规模	矩阵组装耗时(s)	方程组求解耗时(s)	求解占比(%)
63 × 41 × 21	0.10	0.60	85.7
93 × 61 × 33	0.43	4.15	90.6
115 × 71 × 36	0.56	7.26	92.8
141 × 83 × 41	0.93	19.57	95.5
161 × 101 × 51	1.65	66.07	97.6

需要指出的是，Pypardiso 求解器在测试平台(AMD 7500F，6 核心 12 线程)上的并行扩展能力有限，主要受限于稀疏矩阵分解的天然串行依赖。随着网格规模增大，求解器耗时呈超线性增长，这一现象进一步验证了求解器已成为新的性能瓶颈，后续工作应重点关注求解器的并行优化。

5. 结论

本文针对纯 Python 实现的电阻率法正演程序计算效率低下的问题，提出了一套基于 Cython 的靶向优化策略。通过对核心计算模块进行性能剖析，精准识别出矩阵组装为程序的主要性能瓶颈，在此基础上，对全局矩阵组装模块(包含单元刚度矩阵计算)进行 Cython 静态类型重写。主要结论如下：

(1) 优化效果显著。在保持数值精度的前提下，矩阵组装模块获得最高 62 倍加速比，五组不同规模网格的平均加速比达 58.2 倍。其中，单元刚度矩阵计算函数的 36 万次 Python 调用被完全消除，list.append 操作从 4297 万次降至 0，Python 解释器在核心计算环节的开销被极大降低。

(2) 瓶颈转移现象。优化后矩阵组装模块耗时占比从 79.1%降至 9.5%，而方程组求解占比从 20.9%升至 90.5%。这一现象表明，当矩阵组装不再是瓶颈后，原本被掩盖的求解器耗时凸显出来，成为新的性能热点。这并非求解器性能不足，恰恰相反，正是因为矩阵组装优化效果显著，才使得求解器的计算占比被“放大”。

(3) 方法普适性强。在五组不同规模网格(单元数从 5 万至 82 万)上的测试结果表明，矩阵组装模块加速比稳定在 53~62 倍之间，平均达 58.2 倍，证明本文提出的 Cython 优化方法具有良好的普适性和可扩展性，可推广至不同尺度的三维正演问题。

(4) 方法局限性。本文的优化策略在层状地电模型和规则网格剖分条件下取得了显著效果，但实验模型相对简单。在实际应用中，各向异性介质等复杂模型会导致单元计算量显著增加(如单元矩阵从 8 × 8 扩展为 24 × 24)，此时单元计算耗时占比可能重新上升。尽管如此，本文的优化思路仍适用于上述复杂场景，“性能剖析→热点识别→Cython 靶向重构”的方法论框架具有较强的可迁移性。

后续工作将围绕求解器的并行优化展开，包括探索基于 GPU 的稀疏直接求解器(如 cuSOLVER)或迭代法预处理技术(如 ILU 预条件子)，以突破新的性能瓶颈，实现更大规模三维正演的高效计算。

基金项目

省部级项目“频率域可控源电磁法与地面核磁共振法三维交叉梯度联合反演”，广西自然科学青年基金(2024GXNSFBA010257)；省部级项目“基于混合并行的可控源电磁监测数据四维自适应快速反演”，广西科技基地和人才专项基金(桂科 AD23026231)资助。

参考文献

[1] 黄俊革. 三维电阻率/极化率有限元正演模拟与反演成像[D]: [博士学位论文]. 长沙: 中南大学, 2003.

[2] Qin, C., Fu, W., Zhao, N. and Zhou, J. (2025) 3D Adaptive Finite-Element Forward Modeling for Direct Current Resistivity Method Using Geometric Multigrid Solver. *Computers & Geosciences*, **196**, Article ID: 105840. https://doi.org/10.1016/j.cageo.2024.105840

[3] Jarecka, D., Arabas, S., Fijalkowski, M., *et al*. (2012) On the Tradeoffs of Programming Language Choice for Numerical Modelling in Geoscience. A Case Study Comparing Modern Fortran, C++/Blitz++ and Python/NumPy. *Proceedings of the EGU General Assembly Conference Abstracts*, Vienna, 22-27 April 2012, EGU2012-680.

[4] Arabas, S., Jarecka, D. and Jaruga, A. (2013) Object-Oriented Implementations of the MPDATA Advection Equation Solver in C++, Python and Fortran. arXiv: 13011334.

[5] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S. and Smith, K. (2011) Cython: The Best of Both Worlds. *Computing in Science & Engineering*, **13**, 31-39. https://doi.org/10.1109/mcse.2010.118

[6] 石靖. 基于 Python 一站式编程的谱激电法三维有限元数值模拟[D]: [硕士学位论文]. 桂林: 桂林理工大学, 2025.

[7] 徐世浙. 地球物理中的有限单元法[M]. 北京: 科学出版社, 1994.

[8] Zhao, S. and Yedlin, M.J. (1996) Some Refinements on the Finite-Difference Method for 3-D Dc Resistivity Modeling. *Geophysics*, **61**, 1301-1307. https://doi.org/10.1190/1.1444053

[9] 阮百尧, 熊彬, 徐世浙. 三维地电断面电阻率测深有限元数值模拟[J]. 地球科学: 中国地质大学学报, 2001, 26(1): 73-77.

[10] 熊彬, 熊心如, 徐志锋, 等. 引入 Green 函数实现电阻率法总电位高精度三维正演[J]. 地球物理学报, 2025, 68(7): 2786-2791.

[11] Lam, S.K., Pitrou, A. and Seibert, S. (2015) Numba: A LLVM-Based Python JIT Compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, Austin, 15 November 2015, 1-6. https://doi.org/10.1145/2833157.2833162

[12] Foreman, D.J., Chalasani, S. and Walker, M.R. (2025) Quantifying Code Binding Performance for Navigation Algorithms. 2025 *IEEE/ION Position, Location and Navigation Symposium* (*PLANS*), Salt Lake City, 28 April-1 May 2025, 166-172. https://doi.org/10.1109/plans61210.2025.11028355

[13] Schenk, O. and Gärtner, K. (2004) Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO. *Future Generation Computer Systems*, **20**, 475-487. https://doi.org/10.1016/j.future.2003.07.011