用于库学习的数值超优化

JONAS REGEHR、MITCH BRILES、ZACHARY TATLOCK 和 PAVEL PANCHEKHA

数值软件依赖于快速、精确的数学原语实现，例如正弦函数、指数函数和对数函数。现代超优化器能够针对一组给定的原语优化浮点内核，但一个更根本的问题仍未解决：哪些新原语值得优先实现?

我们将此问题表述为数值库学习：给定一组浮点内核工作负载，找出那些经专家实现后最能提升速度和精度的数学原语。我们的关键洞见是，数值超优化器已经具备非常适合解决这一问题的机制。其搜索过程恰好会枚举候选原语，等价性判定过程可以对候选项进行泛化和去重，而成本模型则可以估算反事实效用：如果某个原语可用，工作负载的表现会提升多少。

我们介绍了 GrowLibm，它将 Herbie 超优化器改造为数值库学习器。GrowLibm 从超优化器的中间搜索结果中挖掘候选原语，按照反事实效用对它们排序，并剔除冗余候选项。在三个科学应用（PROJ、CoolProp 和 Basilisk）中，GrowLibm 找到了紧凑、可复用的原语，能够利用标准数值技术有效实现。将这些专家实现集成到 Herbie 后，在精度不变的情况下，内核速度最高提升了 2.2×；最大可达精度也有所提高，其中一个案例从 56.0% 提升到 93.5%。我们还制作了一个 LLVM 匹配器原型，可在优化后的中间表示中识别学习得到的原语，并在五种 PROJ 投影中找回 26 个替换位置，证明了这种方法对端到端速度和精度均有可测量的影响。

ACM 引用格式：
Jonas Regehr、Mitch Briles、Zachary Tatlock 和 Pavel Panchekha。2026 年。用于库学习的数值超优化。1，1（2026 年 3 月），25 页。https://doi.org/10.1145/nnnnnnn.nnnnnnn

arXiv:2603.24812v1 [cs.PL] 25 Mar 2026

1 引言

地图与导航、物理仿真、工程计算机辅助设计——这些日益重要的领域中的计算依赖于简短的浮点内核，其速度和精度会影响应用程序的行为。而这些内核又由目标平台上可用的数学原语构成。这些原语包括基本算术运算符，以及系统数学库提供的初等函数，例如正弦函数、指数函数和对数函数。它们还可能包括 GSL 等第三方库，甚至由专家编写的应用专用库函数。浮点内核随后将这些原语组合成更高层次的数学运算，同时注意确保速度和精度。

事实上，编程语言社区花了十多年时间开发支持这项任务的工具。现代数值超优化器（如 Herbie）[40], 例如，越来越能够针对一组给定的可用原语优化内核 [45],甚至将成果扩展到 LULESH 流体动力学应用等真实应用程序 [44]。Satire 等现代验证工具 [13] 能够在给定这些原语的误差模型后，对这些实现进行静态验证。而 FPTuner [8]和 POP [3] 等现代精度调节器甚至能够优化内核

作者联系信息：Jonas Regehr；Mitch Briles；Zachary Tatlock；Pavel Panchekha。

特此授予许可，允许为个人或课堂使用而免费制作本作品的全部或部分数字版或纸质副本，但前提是不得出于营利或商业利益制作或分发副本，且副本必须包含本声明以及首页上的完整引用信息。本作品中由作者以外的其他人拥有版权的组成部分，必须遵守相应版权要求。经注明出处后，允许进行摘要。除上述情况外，如需复制、再版、发布到服务器或重新分发至列表，必须事先获得特别许可和/或支付费用。请通过 permissions@acm.org 联系以申请许可。© 2026 版权归所有者/作者所有。出版权授权给 ACM。ACM
XXXX-XXXX/2026/3-ARThttps://doi.org/10.1145/nnnnnnn.nnnnnnn

以利用混合精度。针对一组特定原语设计特定的浮点内核，正越来越得到良好支持。

这就引出了一个问题：哪些新原语值得构建？有些原语，例如正弦函数或矩阵乘法，应用十分广泛，实现它们本身就有充分理由。但面对地理空间投影、亥姆霍兹能量公式、天体动力学等诸多领域中的各种数学函数，库作者应如何确定优先级？库作者又该如何判断，哪些原语在各自的定义域中最常用、最重要？

我们将这个问题称为数值库学习：给定一个应用程序和一个代表性工作负载，找出那些通过专门实现能够最大程度改善该应用程序速度与精度权衡的数学原语。数值计算尤其适合这一研究方向，因为原语是小型的纯表达式，而其实现方式则多种多样，包括代数恒等式、多字算术、各种类型的近似、数字递推，甚至硬件感知调优。换言之，原语易于描述，但要想实现得好则需要专业知识。因此，数值库学习旨在优先运用这些专业知识。

我们的核心洞见是，数值超优化器已经具备了库学习器所需的机制。它们的搜索过程会枚举合理的原语，等价性判定过程会识别同一潜在函数的不同表示，而它们的误差和成本模型则回答了库设计者真正关心的问题：如果存在这个原语，它会在哪些地方用到，又能带来多大帮助？

我们提出了 GrowLibm，这是一种数值库学习器，它以“逆向”方式使用数值超优化器：它不再针对一组固定原语优化内核，而是询问哪些新原语能够同时改进多个内核。GrowLibm 从超优化器自身的搜索空间中挖掘候选原语，对它们进行语义去重和泛化，并根据反事实效用对其排序：也就是如果这些原语可用，工作负载能够得到多大改进。直观而言，GrowLibm 倾向于选择出现频繁、规模小，却仍难以由超优化器自行优化的候选项。最终得到的是一份精简清单，范围恰到好处，便于专家关注：这些原语简洁、可复用，在多个内核中反复出现，并能解锁更好的实现。

我们在三个重要的科学应用中评估了 GrowLibm：PROJ 中的地图投影 [43], CoolProp 中的热物性计算 [2], 以及 Basilisk 中的天体动力学内核 [26]。在这些应用中，GrowLibm 提出了紧凑的领域专用原语，数值计算专家可以有效地实现这些原语。实现得当时，许多这类原语会在优化后的内核中重复使用，并拓展速度与精度前沿；与无法使用 2.2× GrowLibm 提议的原语、经过类似优化的内核相比，加速最高可达。新的原语还使得精度能够超越以往水平：在 PROJ 中，有一个内核借助 2 个 GrowLibm 提议的原语，精度从 56.0% 提升至 93.5% 。我们还对 LLVM 匹配器进行了原型设计，使其能够在优化后的 LLVM 中间表示中识别学习得到的原语，并在五种 PROJ 投影中找回了 26 个替换位置。有些投影速度更快（最高提升5%），精度也更高；另一些则以略微减速为代价，大幅提高精度。这表明，学习得到的原语能够拓展部署级速度与精度前沿。

贡献。

- 我们提出数值库学习，这是一个由工作负载驱动的问题：对于给定的数值应用，确定哪些新的数学原语值得由专家实现。
- 我们介绍 GrowLibm：它将 Herbie 超优化器改造为库学习器，从其搜索空间中挖掘候选项，对它们进行语义去重和泛化，并依据反事实效用进行排序。

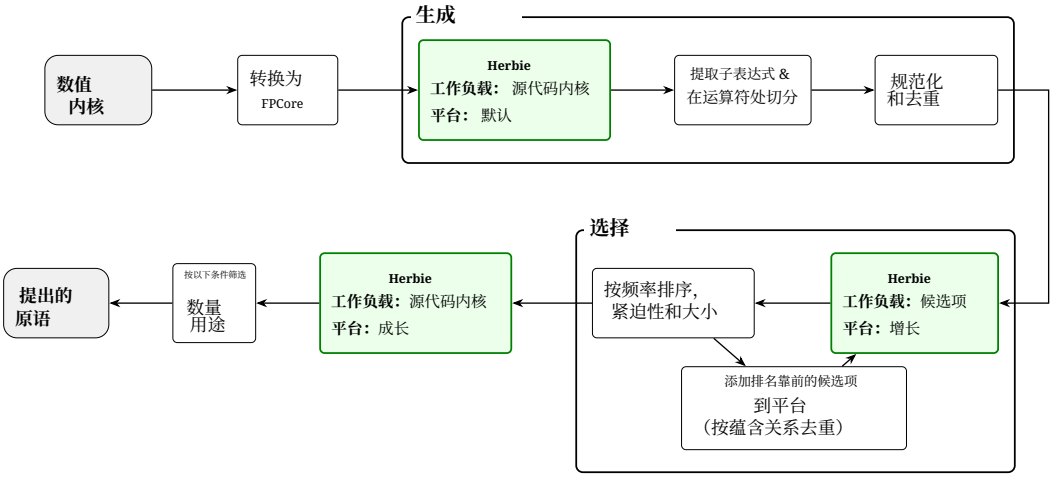


图 1。 GrowLibm 流水线。给定转换为 FPCore 的数值内核 [11, 18], 这一标准浮点交换格式，**生成**阶段运行 Herbie，探索等价程序，从 Herbie 的中间结果中提取子表达式，并对它们进行规范化和去重。 **筛选**阶段通过频率、紧迫性和大小对候选项进行迭代排序；解决顶级候选项之间的蕴含关系，以免同时选中冗余变体；并将胜出项添加到 Herbie 的平台中以重新评估。最后，Herbie 再运行一遍，以确认哪些候选项实际用于优化后的内核，从而生成提议的原语。

- 我们证明，在 PROJ、CoolProp 和 Basilisk 中，GrowLibm 能找到紧凑原语；数值计算专家可以高效实现这些原语，Herbie 可以在优化后的内核中复用它们，而且它们还能拓展可达的速度与精度前沿。

- 我们通过一个原型 LLVM 匹配器展示了部署路径：该匹配器能够在优化后的中间表示中识别学习得到的原语，并将其重写为已部署的实现。

2 概述

我们通过 PROJ 制图库中的一个实例来说明数值库学习 [37],。该库由开放地理空间基金会开发， [38], 用于在不同坐标系之间转换地理空间数据。PROJ 包含大量地图投影，每种投影都定义了正向和逆向投影函数。图 2展示了瑞士斜轴墨卡托投影中的一个投影函数示例。该投影是标准墨卡托投影的一种变体，它倾斜投影圆柱，使真比例尺线由西向东横贯瑞士。这种（较为冷门的）投影是瑞士政府的标准地图投影，因此适用于处理一些瑞士政府数据。

这段简短的代码片段使用了若干复杂的数学表达式，例如 $2\,\mathrm{atan}(\exp(x)) - \pi/2$、$\log(\tan(\pi/4 + 0.5x))$ 和 $\log((1+x)/(1-x))$，它们属于坐标系变换的定义域。在典型系统上，这些表达式会被编译为对底层系统数学库（如 GLibC）的调用，而这些库通常由数学函数实现方面的专家编写。这些系统数学库既准确又快速，这在很大程度上要归功于这些专家以及更广泛的数值分析领域所付出的努力。

问题在于，尽管系统数学库可能快速且精确，但组合精确函数并不总能得到精确的表达式。事实上，上面提到的三个表达式——$2\mathrm{atan}(\exp(x)) - \pi/2$、$\log(\tan(\pi/4 + 0.5x))$ 和 $\log((1+x)/(1-x))$——在某些情况下并不准确

```
static PJ_XY somerc_e_forward(PJ_LP lp, PJ *P) { /* 椭球体，正向 */ PJ_XY xy  =
{0.0, 0.0}; double phip, lamp, phipp, lampp, sp, cp; struct pj_somerc *Q  =static_
cast<struct pj_somerc *>(P->opaque); sp  =  P->e * 正弦函数(lp.phi); phip  = 2. *
atan(exp(Q->c * (对数函数(tan(M_FORTPI  + 0.5 * lp.phi)) - Q->hlf_e * 对数函数
((1.  +sp)/ (1. - sp)))  +Q->K)) - M_HALFPI; }
```

图 2. PROJ 库中瑞士斜轴墨卡托投影的正向椭球投影函数 [42]。请注意其中数学函数的复杂复合，包括2 倍反正切(指数函数(x)) $-\pi/2$、对数函数(正切($\pi/4 + 0.5x$))以及对数函数($(1+x)/(1-x)$)。在浮点数中准确计算这些复合函数颇具挑战，通常超出现有数值超优化器的能力。

```
双精度log1pmd(双精度x) {静态常量双精度Lp1 = 6.
666666666666735130e-01, Lp2 = 3.999999999940941908e-01,Lp3 = 2.
857142874366239149e-01, Lp4 = 2.222219843214978396e-01,Lp5 = 1.
818357216161805012e-01, Lp6 = 1.531383769920937332e-01,Lp7 = 1.
479819860511658591e-01;双精度z  = x *x;双精度R  =
fma(z,fma(z,fma(z, fma(z, fma(z,
fma(z,Lp7,Lp6),Lp5),Lp4),Lp3),Lp2), Lp1);返回fma(x * z,
R,2.0*x);}
```

图 3. 数值计算专家对 对数函数($(1+x)/(1-x)$) 的实现，使用了一个对输入 $|x| < .1716$ 精确的多项式。该实现借鉴了著名的 `fdlibm` 数学库。对于图 2 中的用例而言，输入范围已足够，因为其中的相关输入 x 是地球的偏心率乘以斜角的正弦函数，完全处于该输入范围内。

输入：对于 $x \approx 0$，2 atan(exp(x)) $-\pi/2$ 的精度不足；对于 $|x| > 1$或 $|x| \approx 0$，log(tan($\pi/4 + 0.5x$)) 的精度不足；对于 $x \approx 0$，log(($1+x$)/($1-x$)) 的精度不足。要确定这些情况在物理上是否合理，并了解其对地理空间精度等应用级指标的影响，需要在该定义域具备丰富的专业知识。如果这些表达式确实存在严重的精度问题，则还需要数值方法专业知识来修复根本问题。

提高表达式精度的一种选择，是采用编程语言社区开发的众多数值超优化器和编译器，例如 Herbie [40, 45, 46],Salsa [9, 10, 21],和 Rosa [12, 22]。虽然这些工具使用方便，但它们所采用的技术有限，因此能力不如数值分析师。例如，Herbie 可以为 log1pmd[1] 提出一个精度较高（但速度较慢）的实现，却无法找到一个精度较高的

[1]Herbie 提出 log1p(x) $-$log1p($-x$)；另一种有效实现可能是 2 反双曲正切(x)。这两种实现都不如图 3中的专家实现快，后者利用了额外的领域知识。

```
tmerc.cpp:130: xy.x =Q->ml0 * 对数函数((1. + b) / (1. - b)); omerc.cpp:62:
v = 0.5 * Q->ArB * 对数函数((1. - U) / (1. + U)); isea.cpp:1343: double
log1pe_1me = 对数函数((1 + P->e) / (1 - P->e)); labrd.cpp:26: V2 = .5 *P->e
* Q->A * 对数函数((1. + t) / (1. - t)); labrd.cpp:76: V2 = .5 *P->e * Q->A * 对
数函数((1. + tpe) / (1. - tpe)); labrd.cpp:126: Q->C = .5 *P->e * Q->A * 对数
函数((1. + t) / (1. - t)) +somerc.cpp:27: Q->hlf_e * 对数函数((1. + sp) / (1. -
sp))) +somerc.cpp:54: Q->hlf_e * 对数函数((1. + esp) / (1. - esp))) *
somerc.cpp:89: Q->hlf_e * 对数函数((1. + sp) / (1. - sp)));
```

图 4. PROJ 中计算 log1pmd 表达式的其他代码行。请注意，其中一些用法将 $1+x$ 项放在除法的分子中，而另一些则将其放在分母中。这在数值代码中很常见，这意味着要找到共同函数，就需要对代数等价和复合进行推理。

图 2 中其他两个数学表达式之一的实现，展示了专家实现的原语能够弥补的差距。

如果自动化工具无法胜任，数值方法专家或许能解决问题。通常，数值方法专家会将有问题的计算——即上述三个表达式中的任意一个——提取为库函数，然后运用其数值方法专业知识，为该函数编写一个定制且准确的实现。例如，专家可以选择将 对数函数(($1+x$)/($1-x$)) 函数提取出来，封装成一个定制辅助函数，或许称为 `log1pmd` （意为“对数，1 加负数除法”），然后使用例如多项式近似的方法实现该函数，如图 3所示。或者，专家可能知道已有相同函数的实现；例如，影响深远的 `fdlibm` 数学库通过调用 log 的实现作为子程序来实现 `log1pmd` ，专家也可以提取并复用该子程序。即使自动化工具已经提供了解决方案，专家的帮助仍可能有所裨益。例如，图 3 中的专家实现利用了关于 x可能取值范围的额外领域信息（它等于 e正弦函数 ϕ，其中 e是地球的偏心率，因此 $|x| < 0.08182$），所以它的速度比 Herbie 的实现快得多。

当然，问题在于，即便是 PROJ 这样影响深远的项目，也不能指望数值方法专家投入无限的精力。因此，他们必须有所取舍。例如，瑞士斜轴墨卡托投影对瑞士而言固然很重要，但若将宝贵的数值方法专业知识用于本质上只服务于一个国家的库例程，那就太可惜了。能够处理许多地图投影都会用到的计算的通用、可复用的例程，会更有价值。幸运的是，log1pmd是一种通用且可复用的例程：它所计算的表达式在许多地图投影中都会出现。但是，要在大型现有代码库中找出这些通用例程，以最大限度地利用现有的数值专业知识，并非任何现有工具都能胜任。

GrowLibm 是首个此类工具。GrowLibm 会分析大量数值基本块，并找出数值计算专家可以实现的常见可复用子程序，以提升应用程序的精度和性能。这些子程序是简短的数学表达式，当前的数值超优化器无法对其进行改进，但它们会出现在许多基本块中，并显著影响这些基本块的精度和性能。正如 log1pmd 示例所示，从工作负载中恢复可复用原语面临三项挑战：识别不同上下文中语义等价的变体，估计哪些候选抽象最能改善工作负载，以及推广到源代码中发现的特定表达式之外。GrowLibm 通过重新利用现有的超级优化器机制来应对每项挑战。

首先，GrowLibm 能够推理语义等价性，包括代数恒等式下的等价性。例如，图 4 展示了 log1pmd 在 PROJ 中的其他一些用途。图 4 中的许多匹配项使用了不同的变量名，甚至不同的表达式来表示 x。此外，有些用对数函数($(1+x)/(1-x)$)，这正是 log1pmd 的实现方式；另一些则用对数函数($(1-x)/(1+x)$)，这就需要调用 log1pmd($-x$)。GrowLibm仍然能够看出它们都共有 log1pmd ，因为它利用了 Herbie 基于 e-graph 的等价性推理，能够高效判断哪些候选库函数彼此等价，或者较宽泛地说，哪些函数可以用来实现其他函数。

其次，GrowLibm 能够推理每个候选库函数对整个表达式精度和性能的影响。例如，图 2还包含表达式 $e \cdot$ 正弦函数(ϕ)。不过，这个表达式的精度并没有特别差，而且它不在最长延迟路径上，因此其性能也没那么重要。GrowLibm 因此不会建议实现该函数。更一般地说，GrowLibm 利用 Herbie 的精度与性能成本模型来确定哪些候选项值得专家关注。该模型能够估算反事实影响（如果某个原语可用，工作负载的性能会提升多少）。

最后，GrowLibm 能够对源代码中实际出现的模式进行泛化和抽象。例如，在图 2中，内联 sp 变量（编译器会这样做）后可以看出，实际计算的函数并非 对数函数($(1+x)/(1-x)$)，而是更具体的 对数函数($(1+e\cdot$正弦函数$\phi)/(1-e\cdot$正弦函数$\phi)$)。然而，这个更具体的函数并不适用于图 4中其他所有用法。为了以这种方式进行泛化和抽象，GrowLibm 利用 Herbie 的搜索过程，探索代数重写、近似和对称性破缺，以找到与源代码中原样出现的表达式相关、且适合作为库函数的表达式。

值得注意的是，这些能力都不是 GrowLibm 或数值计算所特有的。等价性推理、影响估计和泛化是任何超优化器的核心组成部分；GrowLibm只是将它们重新用于库学习，而非优化。任何拥有超优化器的定义域（无论针对数值内核、二进制代码、SQL 查询还是硬件电路）原则上都可以采用相同的方法，发现哪些新原语最能使其工作负载受益。第 3 节和第 4 节将详细介绍 GrowLibm 如何实现这种联系。

3 生成候选原语

本节介绍 GrowLibm 如何生成候选原语：这些子表达式如果由专家实现，就可以作为目标应用程序内核的新原语。输入是以 FPCore 中的基本块表示的应用程序内核；输出是一个大型去重候选项集合，供选择阶段（第 4节）进行排序。核心见解是，超优化器的搜索过程已经探索了相关空间：为了优化一个内核，它会枚举许多等价程序，而这些程序的子表达式经过规范化和泛化后，自然就成为新原语的候选项。

超优化器必须在庞大的程序集合中进行搜索——理想情况下，搜索所有与给定输入程序等价的程序。例如，数值编译器/超优化器 Herbie 使用等式图高效地搜索代数等价项，并且还包含用于函数近似和对称性破缺的子组件。它利用这一搜索过程，选出与输入程序等价、但精度更高或求值更快（或在帕累托前沿上兼具两者）的程序。

其中一些等价程序会包含具有明确语义的新颖指令子序列，可将其提取为候选原语。由于 GrowLibm 重用了 Herbie 的

搜索过程，该过程经过十余年真实部署的不断完善，[2] 因此它无需从头构建，就能继承高质量、高吞吐量的候选生成过程。

更具体地说，GrowLibm 的候选生成遍会对所有输入内核运行 Herbie。对于每个输入内核，Herbie 会搜索许多——通常是数千或数万个——等价程序，剪去绝大多数，并输出它找到的最佳程序。关键在于，GrowLibm 会忽略 Herbie 的最终输出，因为这些输出尚无法从 GrowLibm 正在尝试发现的原语中获益；它会转而收集剪枝之前探索过的所有中间表达式。这颠倒了超优化器通常的作用：GrowLibm 并非利用搜索来寻找最佳程序，而是挖掘搜索本身，寻找在多个内核中反复出现的子表达式。这些中间表达式，而非最终优化后的程序，才是库学习的关键，因为它们代表了表达式空间，而 GrowLibm 随后将估算这些表达式的反事实效用。

超优化搜索过程中找到的程序与某个原始内核等价。由于 GrowLibm 要寻找可用于这些内核之一的候选原语，因此它会考察某个搜索程序的每个子表达式，并提取任意切分。例如，如果搜索步骤考察了程序 $(\log(1 + x \cdot y))^2$，那么 GrowLibm 会考察 $\log(1 + x \cdot y)$ 这样的子表达式，以及 $\log(1 + z)$ 这样的子表达式切分。为了生成切分，GrowLibm 不采用固定深度，而是在每个二元或更高元运算符处考察切分。因此，对于 $\log(1 + x \cdot y)$，完整的切分集合为 $\log(z)$、$\log(1 + z)$ 和 $\log(1 + x \cdot y)$。然而，不会考察 $\log(z + x \cdot y)$，因为常数 1 不是二元运算符。其直觉是，二元运算符会增加扇出，进而增加候选原语的参数数量（参数很多的原语更难实现）；而一元运算符不会增加扇出，因此没有理由在这些运算符处引入切分。一般而言，对于平衡二叉树，此类切分的数量会呈指数增长，但在我们的实验中，这种情况很少见，切分数量通常也较少。我们测试了其他切分策略，包括较简单的固定深度策略，以及根据切分运算符的类型进行调整的较复杂策略，并发现这种简单的二元启发式效果最好。

提取所有切分后，GrowLibm 拥有大量要传递给选择步骤的候选原语集合。然而，该集合中包含许多重复项。这些重复项的出现有两个原因。首先，两个表达式可能仅在 α重命名上有所不同，例如 $\log(1 + x)$ 和 $\log(1 + y)$。其次，两个表达式可能彼此等价，例如 $\log(1 + x)$ 和 $\log(x + 1)$。重复项的数量可能非常多，而去重可以让 GrowLibm 的选择阶段快得多。因此，GrowLibm 通过以下方式对表达式进行去重：首先规范化所有变量名；然后考虑变量的所有排列；最后使用等式图，再次复用超优化器的等价性机制，尽可能证明更多表达式之间的等价性。例如，给定表达式 $\frac{x}{1+y}$，GrowLibm 首先规范化变量名，生成 $\frac{t_1}{1+t_2}$。随后，它考虑规范化变量的所有排列，得到 $\frac{t_1}{1+t_2}$ 和 $\frac{t_2}{1+t_1}$。这样，GrowLibm 就能对 $\frac{x}{1+y}$ 和 $\frac{y}{1+x}$ 这样的表达式进行去重，因为它们仅因变量重命名而不同。凡是其某个排列被证明相等的两个表达式，都会被去重（并记录节省的重复项总数，以供选择时使用）。这种方法可以对 $(1 + x) \cdot y$ 和 $x \cdot (1 + y)$ 这样的表达式进行去重；只有同时考虑 α重命名（交换 x 和 y）与代数重排（加法交换律），才能证明二者等价。不过，这一等价性过程开销很大，因此需要将表达式限制

[2]国家实验室（桑迪亚国家实验室、美国国家航空航天局）和工业界（英伟达、英特尔）都有 Herbie 的用户，且已有许多出版物以优化其搜索过程为主题 [15–17, 27, 33, 39, 40, 45, 46, 60]。

为最多包含三个变量。由于包含许多变量的数学库原语更难实现，这一限制相对温和。

4 选择库原语

生成阶段会产生数十万个候选原语。选择阶段的任务是从中挑选少量原语，以最大程度地改进目标应用。核心见解是，超优化器可以直接回答库设计者的问题：如果存在某个给定的原语，工作负载能提升多少？这正是超优化器的成本模型旨在估算的反事实效用。

4.1 通过平台扩展评估候选项

超优化器通常由针对目标编程语言或平台中可用指令的机械化语义驱动，这些语义用于证明程序与原程序等价；此外，超优化器还会使用某种成本模型来评估哪些指令序列最佳。例如，Herbie 以平台描述 [45] 为参数，该描述定义了所有可用的浮点运算、它们的输入和输出类型、实数语义，以及用于估算其成本和精度的函数。随后，Herbie 使用基于指令语义的实数推理，以及针对其成本函数的组合优化，来搜索可能的指令序列。候选原语可以表示对该平台定义的扩展：从原语定义中获取类型和语义，并添加假设的成本和精度函数。评估每个候选原语时，可以：1）将该候选原语加入超优化器平台进行扩展；2）使用扩展后的平台重新优化工作负载的内核；3）衡量精度和性能的提升。这就是反事实效用计算：超优化器会直接回答这个问题：“如果能使用该原语，这个工作负载是否会有所改善？”我们用$\mathrm{acc}(A, P)$ 表示在平台扩展为包含一组原语后，对 A 中的内核进行超优化所达到的归一化精度；该值在所有内核上取平均，并且以 P为界。 $[0, 1]$。

假设目标应用程序包含一组 K 内核。候选项生成步骤会对所有 K进行超优化，从而得到一组 C 候选原语。GrowLibm 的最终目标是选取一个子集 $N \subseteq C$，使 $\mathrm{acc}(K,\ N)$ 最大化。最朴素的方法是枚举C的所有子集，并针对每个子集重新优化 K，这需要进行 $|K| \cdot |C|^{|N|}$ 次超优化。然而，这种做法极不切实际：在我们的评估中，大约有 $|K| \approx 10^2$、$|C| \approx 10^6$ 和 $|N| \approx 10^1$，这意味着需要进行约 10^{62} 次超优化，而每次都要花费几秒钟。因此，GrowLibm 会采用一系列启发式方法和优化，以将所需的超优化次数减少到合理的范围。

我们将在下文讨论排序启发式方法，但首先关注另一种启发式方法：在候选原语扩展超优化器平台时所使用的成本和精度函数。成本和精度的定义必须 乐观：我们的目的是让人类专家协助实现选定的候选原语，因此其成本和精度将优于 超优化器找到的实现。为此，我们让超优化器假定候选项可以做到零误差，且成本仅为其朴素实现成本的 $1/5$ 。这些乐观的成本和精度定义会激励超优化器寻找候选原语的用途，从而帮助确定哪些候选原语最有用。当然，存在过于乐观的风险；在这种情况下，选定的候选项可能根本无法高效实现。我们将在第 5 节进一步讨论这个问题。这些乐观假设仅用于排序；第 6 节将使用真实的专家实现验证最终结果。不过，为降低这一风险，排序启发式方法会优先选择更有可能高效实现的项。

4.2 按大小、频率和紧迫性对候选项进行排序

GrowLibm 做出的第一项简化，是采用贪心策略，每次添加一个原语来扩充选定原语集合。也就是说，GrowLibm 维护一个已选候选原语集合 S。然后，它会考察每个尚未选定的候选原语 f，并搜索使 $\mathrm{acc}(K,\ S\cup\{f\})$ 最大的候选项，将其加入 S，然后重复此过程，直到 S 达到所需大小。然而，这种贪心算法的朴素实现仍需要进行 $|K|\cdot|C|\cdot|N|$ 次超优化，数量依然过大，约为 10^9。

为了进一步缩短运行时间，GrowLibm 发现，候选原语要值得选择，必须具备三个属性，而这些属性都可以利用超优化器现有的机制进行估算。首先，大小启发式基于候选项在超优化器成本模型中的成本。该启发式倾向于选择较小的候选原语，也就是说，其描述实数行为的抽象语法树较小。较小的候选原语通常更容易让人理解和实现，也不太可能过度专用。其次，频率启发式倾向于选择能在目标应用程序中更频繁使用的候选原语。频率统计超优化器考虑的、包含该候选原语的表达式出现了多少次。这利用超优化器的搜索过程作为一种代理指标，用以估计如果该候选原语可用，超优化器使用它的可能性。第三，紧迫性启发式倾向于选择超优化器无法自行改进的候选原语。如果超优化器能够自行找到候选原语的高质量实现，专家就没有必要亲自编写实现。紧迫性通过 $1-\mathrm{acc}(f,S)$ 衡量，也就是说，对每个候选项进行超优化时，f 给定已选定的候选项 S；精度则由实际达到的精度与最大精度之间的差距决定。三种启发式以一种直接的方式组合：将频率与紧迫性相乘，再除以大小。我们观察到，在这三种启发式下得分较高的候选原语，也最有可能提高 $\mathrm{acc}(K,S)$。此外，这些启发式的测量成本更低：大小和频率可以在原始生成阶段测量，而紧迫性启发式只需要 $|N|\cdot|C|$ 次超优化。加上生成阶段的 $|K|$ 次超优化，总运行时间降至 $|K|+|N|\cdot|C|\approx 10^7$ 次超优化。

在实践中，这三种启发式都能帮助区分有希望的候选项和没有希望的候选项。尽管如此，按某一指标排名靠后的候选项，往往在其他指标上的表现也很差。例如，一个冗长且过于复杂的候选原语，很可能不仅大小过大（未通过大小启发式），而且过于专用（未通过频率启发式）；而一个包含许多不必要或重复操作的候选原语，很可能既过大（未通过大小启发式），又能通过简单方法进行优化（未通过紧迫性启发式）。这表明，应先依据大小和频率启发式进行筛选，因为评估这两项不需要进行超优化；紧迫性则只需在第二遍中使用。因此，我们将启发式排序分为两个阶段：首先依据大小和频率启发式选出排名前 T_1 的候选原语，然后仅对这些候选项按紧迫性排序。这会将运行时间从 $|K|+|N|\cdot|C|$ 次超优化降至 $|K|+|N|\cdot T_1$；GrowLibm 将 $T_1=625$ 设为，因此超优化次数约为 6×10^3。将 T_1 设得较高，可确保总体排序不会有太大变化：按大小、频率和紧迫性排名靠前的候选项，必定也在仅按大小和频率排序时的前 T_1 名之内，因为有相当多的候选项（在我们的评估中约为 4%）具有较高的紧迫性。大小和频率指标较低的候选项不可能具有足够高的紧迫性（因为紧迫性在 $[0,1]$ 范围内有界），因此无法排在它们前面。

图 5 从经验上证实了这一结构：候选项的大小和频率分布较为分散，而紧迫性则呈现明显的双峰分布，约 96% 的候选项紧迫性接近于零，只有一小部分紧迫性较高。这恰好说明，先按大小和频率进行筛选，再按紧迫性排序，是一种有效的方法。

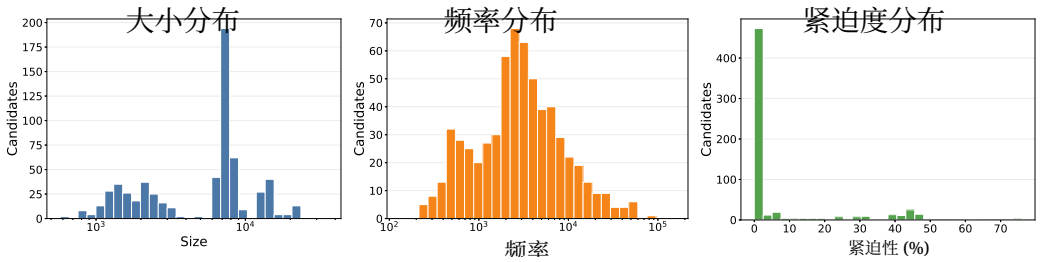


图 5. PROJ 一次运行中大小、频率和紧迫性启发式的分布。大小和频率分布均匀，提供了良好的区分信号。紧迫性集中在零附近，但有一条延伸至高紧迫性候选项的长尾，这证实了双峰结构，也证明了采用两阶段筛选的合理性。

4.3 通过蕴含剪除冗余候选项

为进一步缩短运行时间，GrowLibm 按批次而非一次一个地选择候选项。然而，若仅根据大小、频率和紧迫性选出排名靠前的几个候选项，往往会选中同一候选项的不同变体。例如，$\log(1+x)$ 和 $\log(1+x\cdot y)$ 可能在大小、频率和紧迫性方面都排名靠前。这些候选原语并不等价，因此在生成阶段不会被去重。然而，任何对 $\log(1+x)$ 的调用都可以替换为调用 $\log(1+x\cdot y)$，其中 $y=1$；反过来，任何对 $\log(1+x\cdot y)$ 的调用也可以替换为调用 $\log(1+x)$，并将其参数设为 $x\cdot y$。要求人类专家同时实现这两者，会浪费他们的时间。事实上，出现这类变体是意料之中的：这些候选项的频率往往相近，因为它们都是同一个初始表达式的切分；紧迫性也相近，因为它们对应相似的数值问题（这里是 log 在 1 处的零点）。再举一例，考虑候选原语 $\mathrm{pow}(\cos(x), 6)$；任何对该候选原语的调用都可以替换为对更通用的 $\mathrm{pow}(x, 6)$ 函数的调用，但反过来则不成立，因为 $\cos(x)$ 的取值范围并不覆盖 x 的整个定义域。在这种情况下，最好先选择更通用的候选原语，并推迟选择更具体的候选项，直到重新计算其紧迫性之后。

要一次添加多个排名靠前的候选项，解析此类蕴含关系至关重要。考虑一个由 T_2 个按大小、频率和紧迫性排名靠前的候选项组成的“批次”。GrowLibm 会针对批次中的每一对候选项 f 和 g，测量 $\mathrm{acc}(g, S\cup\{f\})$；如果结果超过 95%，它便记录下 $f \prec g$，这意味着选择 f 会使 g 变得冗余。由于批次中已经包含紧迫性排名靠前的候选项，$\mathrm{acc}(g, S)$ 通常较低；如果 $\mathrm{acc}(g, S\cup\{f\})$ 较高，那么选择 f 就不必再选择 g。这种方法复用了反事实效用计算：超优化器通过检查不包含某个候选项时工作负载是否同样有所改善，来判断一个候选项是否会使另一个候选项变得冗余。$\prec$ 关系构成了批次中 T_2 个候选原语的有向图；GrowLibm 将此图划分为由强连通分量构成的有向无环图 [54]，并从该图无环部分中没有入边的每个强连通分量里选出一个候选原语。这样可以确保选定的原语彼此不会相互取代，避免浪费专家的工作。假设 T_2 个候选原语中有一个固定比例 α 被选中，则所需的超优化次数会减少到 $|K|+(|N|/\alpha T_2)(T_1+T_2^2)$。在 GrowLibm 中，$T_2$ 设为 25，α 约为 1 4，因此超优化次数减少到大约 $10^2 + \frac{10}{0.25}\cdot 25\ (625 + 25^2) \approx 2{,}100$。由于每次超优化需要几秒钟，最终流水线只需几小时即可运行完成（更多信息请参见第6节）。

调整 T_1 和 T_2 以实现可处理的选择。T_1 和 T_2 都是参数，用于权衡选定原语的质量与选择阶段所需的时间。为了选择 T_1，我们观察到候选原语的平均紧迫度约为 4%。在实践中，紧迫度

呈现明显的双峰分布——大约 96% 的候选项几乎没有紧迫性，其余约 4% 则具有较高的紧迫性。因此，按大小、频率和紧迫性启发式排序的前 $.04T_1$ 个候选项，很可能也属于仅按大小和频率排序的前 T_1 个候选项。由于蕴含步骤只使用前 T_2 个候选项组成的一批数据，这表明 $T_2 = .04T_1$。同时，为了尽量减少超优化的次数，应选择 T_1 和 T_2，以使 $(1/T_2)(T_1 + T_2^2)$ 最小。该函数在 $T_2 = \sqrt{T_1}$处达到最小值。结合这两个公式可得 $T_1 = 625$ 和 $T_2 = 25$。

5 实现库原语

GrowLibm 会建议候选原语，但不会自动实现它们；这项工作需要数值计算专业知识。因此，一个关键问题是：GrowLibm 的建议是否处于专家有限精力所能处理的合适范围内？为了检验这一点，我们在评估中实现了 GrowLibm 选出的每个原语。本节将详细介绍这些实现，并展示多种标准技术（现有库函数、无误差变换、区间缩减和多项式近似）足以实现快速而精确的实现。对于每个原语，我们力求达到 1个ULP 或几个 ULP 的精度，同时将成本控制在足够低的水平，使 Herbie 倾向于采用它们。为了方便读者（也方便我们自己），我们在下文为每个原语命名，尽管 GrowLibm 本身只输出数学规范。除非另有说明，下文所有原语均来自 PROJ——这是我们三个工作负载中规模最大、种类最丰富的一个（涵盖五种地图投影，共 92 个内核）；CoolProp 和 Basilisk 各自产生的原语集合规模较小，也更为集中。

我们的实现由一位作者指导，他是初等函数实现方面的专家。不过，为了限制投入实现的工作量，我们请这位专家只向另一位作者发送简短指示；这位作者是一名从未实现过数学函数的学生。专家作者本人没有参与实现。除了发挥自身能力并借助 GPT 5.3 Codex 之外，第二位作者还使用了知名的`fdlibm`库 [35] 以及 Sollya 工具 [7] 来生成多项式近似，这两者均由我们的人类专家推荐。每种实现均使用 C语言编写，由 Apple clang 17.0.0 版编译，并在计时和测试框架中运行。该框架将计算结果与 Rival 库生成的高精度结果进行比较 [16]以评估精度，同时为执行计时以估算成本。随后，我们利用所得的精度和成本数据，创建了一个包含所有候选原语实现的新 Herbie平台，并再次用它对目标应用程序进行超优化。

5.1 现有库函数

GrowLibm 推荐的某些原语已在标准数学库中实现，或者可以通过提高 Herbie 的调优参数来处理，这验证了 GrowLibm 的选择与已知的数值计算实践相符。

GrowLibm 选择了标准原语 $\mathrm{hypot}(x, y) = \sqrt{x^2 + y^2}$。标准数学库中已提供此函数，因此除了链接到正确的函数名称外，无需额外工作来实现它。

GrowLibm 选择了原语 $\mathrm{verdcos}(x) = \cos(2x) - 1$，这是众所周知的“余弦函数”的一种变体。虽然在 GrowLibm 使用的标准设置下，Herbie 无法找到更好的实现，但只需提高调优参数（例如允许的迭代次数），Herbie 就能提出实现 $\mathrm{verdcos}(x) = -2\sin(x)^2$,，该实现既快速又准确。据推测，如果 GrowLibm 在生成和选择阶段使用这些更高的参数，就不会选择 verdcos，因为它的紧迫性会很低。

GrowLibm 选择了原语 $\mathrm{log1pmd}(x) = \log((1 + x)/(1 - x))$。使用更高的调优参数运行 Herbie 后，Herbie 提出了实现 $\mathrm{log1pmd}(x) = 2$反双曲正切(x),

但我们还注意到，fdlibm 包含一种使用七次多项式的优化实现，其速度更快且精度相同。

5.2 双双精度技巧

其他原语　需要采用扩展精度技术来处理其输入中的舍入误差。

GrowLibm 选用了原语 sinprod(x, y) = sin$(x \cdot y)$，以及类似的余弦乘积原语。当 x 和 y 相对于 sin 的周期 2π 较大时，直接实现的精度不足，因为舍入误差可能大于周期。我们的人类专家建议将 $x \cdot y$ 计算为一种“无误差变换” [36],，它同时返回常规的浮点乘积 $z = x \hat{} \times y$ 及其误差 $e = \text{fma}(x, y, -z)$；随后利用正弦和角公式计算 sin$(z + e)$。为获得最佳性能，我们使用了标准库的 sincos 函数，它通过一次更快的函数调用同时计算 sin(z) 和 cos(z)，sin(e) 和 cos(e) 也同样如此。

GrowLibm 还选择了原语 sinquot(x, y) = sin(x/y)，以及类似的余弦商原语。基于无误差变换、类似于 sinprod 的初始实现并不十分准确，我们的人类专家认为，这是因为 $x \cdot y$ 的无误差变换是精确的，而 x/y 的无误差变换则不是。随后，我们的人类专家提出了一种实现方案：对于较大的 x 值，将 y 参数拆分为两半y_h和 y_l，然后使用 Payne-Hanek 归约，通过将 $\pi/2$ 与 y_h 和 y_l 进行高精度乘法，计算 $x \bmod y\pi/2$。该方案确实实现了很高的精度（得益于 GPT 5.3 Codex，且只需极少的额外提示），但生成的实现速度极慢，Herbie 因此没有在任何目标应用基本块中选择它。我们的专家认为，进一步提升性能会很困难。

5.3 区间缩减与多项式近似

数学库开发中一种常见的实现模式是将区间缩减与多项式近似结合起来。

GrowLibm 选择了原语 invgud(x) = log(tan$(\pi/4 + x/2)$)。这个函数显然称为“反古德曼函数”，与墨卡托投影有关。我们的人类专家建议使用 fdlibm 中的 rem_pio2 函数进行区间缩减，然后采用 Sollya 生成的多项式近似。由此得到的实现精度很高（在所有测试点上的误差均为 1 末位单位），而且效率也很高。

对于 Basilisk 工作负载，GrowLibm 选择了原语 powcos2(x) = pow(cos(x)，2)、powcos4(x) 和 powcos6(x)。对于这些函数，我们最初尝试采用 rem_pio2 区间缩减和 Sollya 生成的多项式近似，但结果速度太慢，Herbie 也没有选择使用它们。我们的专家建议采用 CORE- Math 库中的一种更快的区间缩减方法 [50],，虽然在少数点上的精度较低，但速度显著更快。Herbie 实际上采用了这个更快的实现。

对于 CoolProp，GrowLibm 选择了原语 pow1ms$(x,\ y)$ =pow$((x-1)^2,\ y)$。我们的专家建议将其改写为

$$\text{pow}((x-1)^2, y) = \exp(\log((x-1)^2) \cdot y) = \exp(\log(1 + x \cdot (x-2)) \cdot y)$$

并使用标准函数 log1p 计算 log$(1+z)$，其中 $z = x \cdot (x-2)$。这一实现速度足够快，因此被 Herbie 采用。

简而言之，除了 GrowLibm 建议的两个原语之外，标准技术足以应对其余所有原语。这两个例外是 sinquot 和 cosquot，它们需要多字算术，而我们的专家认为这种方法太慢，不具备实用性。其余原语的精度均达到接近 1个ULP，且无需采用新颖的数值方法，这表明 GrowLibm 往往会选择难度适中的原语。

6 评估

我们围绕六个研究问题评估 GrowLibm 建议的原语：

研究问题1 GrowLibm 提出了哪些新原语？研究问题2 专家能否很好地实现这些原语？研究问题3 Herbie 能否使用这些新原语来优化目标内核？研究问题4 这些新原语能否改善目标内核的性能并降低误差？研究问题5 学习得到的原语能否部署到编译后的应用程序中？研究问题6 GrowLibm 需要多长时间才能运行？

我们在来自不同科学领域的 3 个开源目标应用程序上评估 GrowLibm：

(1) PROJ [43] 是一个大地测量坐标变换和地图投影库，例如用于将地理空间数据集从一个坐标参考系转换到另一个坐标参考系。(2) CoolProp [2] 是一个用于计算流体热物理性质的库，例如用于为蒸汽轮机动力系统建模。(3) Basilisk 是一个天体动力学仿真框架，用于对航天器动力学建模，例如用于轨道规划或制导系统开发。

每个项目的规模都很大，因此无法使用 Herbie 对整个库进行超优化。我们改为从每个项目中选取一个子集：从 PROJ 中提取了 92 个内核（来自五个文件，这些文件实现了 krovak、斜轴墨卡托投影、横轴墨卡托投影、空间斜轴墨卡托投影和瑞士斜轴墨卡托投影），从 CoolProp 中提取了 24 个内核（来自一个实现亥姆霍兹能量公式的文件），并从 Basilisk 中提取了 28 个内核（来自一个实现轨道力学计算的文件）。我们选择了每个项目中彼此相关的内核，因为库学习旨在发现通用、可复用的原语，而同一应用领域中的内核更有可能产生此类原语。我们还将范围限制为至少调用一个初等函数的内核，因为 Herbie 的目标就是这类内核。我们将每个内核转换为 GrowLibm 的输入格式（FPCore [11, 18]），然后将每个项目分别用作GrowLibm的一个独立目标应用程序。除研究问题5（见第 6.5节）外，我们所有实验均在一台配备英特尔 i7-8700K CPU 和 32 GiB 内存的机器上进行，运行 Ubuntu 24.04.4、Rival 2.3、Egg 0.11.0 和 Racket 9.1。

以下各小节追踪GrowLibm从提出建议到部署的整个过程。研究问题1–研究问题3通过图6展示GrowLibm 提出的原语简洁且适合相应定义域；这些原语能够被准确地实现（我们报告 Herbie 的平均误差位数，越低越好），并且效率高（我们报告相对于基线的吞吐率倒数，越低越好）；Herbie 还会大量复用这些原语。研究问题4接着考察这些实现能否真正提升内核级性能，使用速度／精度前沿曲线（图7）和逐内核误差对比（图8）。研究问题5通过一个原型 LLVM 匹配器，测试在编译后的 PROJ 代码中进行部署的效果；研究问题6则报告GrowLibm自身的运行时开销。

6.1 研究问题1：GrowLibm 提出了哪些原语?

GrowLibm从超优化器的搜索空间中挖掘原语。得到的结果是否简洁、易于识别且适合相应定义域? 图6 的前两列列出了GrowLibm 为 PROJ、CoolProp 和 Basilisk（依次排列）选出的所有原语。

所有这些函数的规格都简短而通用。有些函数（如 hypot）可在标准数学库中找到。另一些函数（如 invgud 和 verdcos）则是其定义域中广为人知的函数（或其变体）。例如，invgud 是反古德曼函数，它与墨卡托投影有关 [59]。有些函数令人意外；例如，当 Herbie 将朴素实现的 cosprod 和 sinprod 原语评为精度不足时，我们感到很惊讶。然而，咨询我们的专家后我们得知，对于非常大的输入 x 和 y（满足 $xy > 2^{52}$），xy 的舍入误差可能与正弦函数或余弦函数的周期相当，甚至更大；在这种情况下

Name	公式	朴素 Cost	朴素 误差	专家 Cost	专家 误差	Uses
hypot	$\sqrt{x^2 + y^2}$	1.201	29.571	**0.864**	**0.000**	39
invgud	对数函数 (tan ((2x + π)/4))	4.186	59.875	9.955	**0.082**	22
log1pmd	对数函数 ((1 + x)/(1 − x))	2.437	58.944	**1.093**	**0.000**	30
cosprod	$\cos(xy)$	2.106	27.309	4.086	**7.684**	4
sinprod	正弦函数(xy)	2.145	27.940	4.086	**7.728**	9
cosquot	$\cos(x/y)$	2.192	26.896	$1。9 \cdot 10^6$	**0.100**	0
sinquot	正弦函数(x/y)	2.231	28.250	$1。9 \cdot 10^6$	**0.239**	0
verdcos	$\cos(2x) - 1$	2.441	15.306	2.562	**0.272**	10
pow1ms	$\mathrm{pow}((x-1)^2, \quad y)$	3.311	15.099	**2.855**	**0.041**	87
powcos2	$\cos(x)^2$	4.104	0.285	**4.056**	0.355	2
powcos4	$\cos(x)^4$	4.104	0.414	**3.461**	0.449	41
powcos6	$\cos(x)^6$	4.104	0.628	**3.884**	0.726	6

图 6. GrowLibm 针对三个目标应用程序——PROJ、CoolProp 和 Basilisk（依次排列，以水平线分隔）建议的库原语。对于每个库原语，“朴素实现”显示直接实现该公式的成本（相对于基准的吞吐率倒数）和误差（Herbie 的平均误差位数指标）；“专家实现”则显示我们的专家建议的实现所需的成本和误差。GrowLibm 建议的原语可以看出是简单、通用的原语，其朴素实现通常不准确；而专家实现总体上极其准确，且成本可以接受。

对这些原语采用更复杂的实现可以提高精度。同样，我们起初不明白为什么 pow1ms 函数要将 $x - 1$ 项平方。我们的专家向我们解释说，由于 $(x - 1)^2$ 保证为正数，因此这种形式的 pow1ms 更容易实现高精度；显然，pow 很难处理负的第一个参数。总之，对 GrowLibm 提出的原语进行考察，支持了这样一种观点：这些原语经过了精心选择，适用于各目标应用程序各自的定义域。

此外，值得注意的是，这三个目标应用程序的 GrowLibm 建议原语集合彼此完全不相交。这表明 GrowLibm 建议的是针对各个目标的特定原语。另一个佐证是，例如，反古德曼函数已知与墨卡托投影有关，而其变体正是我们许多 PROJ 内核的来源。

6.2 研究问题2：专家能否实现 GrowLibm 选定的原语?

要让数值库学习切实可行，所建议的原语必须具有恰当的范围，确保专家能够在有限的投入下完成实现。

图 6接下来的四列列出了每个 GrowLibm建议原语的朴素实现和专家实现的成本与误差。成本和误差越低越好；朴素实现始终是将公式直接转换为浮点运算，而专家实现则如第 5 节所述。误差由 Herbie 测量，成本则由一个小型基准测试框架测量；两者都通过对数千个输入进行采样来评估。专家编写的实现通常精度很高；除了 `sinprod` 和 `cosprod`之外，所有实现的平均误差都不到 1 位。[3] 朴素实现通常速度更快，但其

[3]我们的专家实现的 sinprod 和 cosprod 在 xy 溢出的点上不准确，此时我们的专家实现和朴素实现都会返回非数。在其他所有点上，专家实现仍比朴素实现准确得多。我们的专家确实提出了一种方法，可以让 sinprod 和 cosprod 即使在这些最具挑战性的输入上也能准确实现，但我们没有时间付诸实施。

精度不足，几乎无法使用，而专家编写的高精度版本通常只比朴素版本慢几倍。例如，尽管专家编写的 invgud 比朴素实现慢四倍以上，但 Herbie 测得朴素实现的误差接近 60 位，[4] 而专家实现的误差不到 0.1 位。三种专家实现——hypot、log1pmd和 pow1ms——速度更快，而且比朴素实现更精确。另一方面，三种 powcos 实现的朴素版本本身已经很精确，但专家版本的速度显著更快。总之，专家对大多数 GrowLibm建议的原语都实现得既准确又高效。

这些良好结果中唯一的重大例外是 cosquot 和 sinquot 原语。准确实现这些原语颇具挑战，因为与 cosprod 和 sinprod 不同，除法运算的舍入误差无法用浮点数精确表示。因此，专家推荐的实现需要极其昂贵的多字算术运算；尽管结果极其精确，但速度太慢，不适合实际使用。进一步咨询我们的专家后得知，虽然实现可能还可以有所提速，但我们的专家认为，没有办法让这些实现同时达到足够的精度和速度，以用于现实世界中的实际场景。[5]

6.3 RQ3：Herbie 会使用 GrowLibm 推荐的原语吗?

这个研究问题直接检验 GrowLibm 的反事实效用预测：如果 GrowLibm 预测某个原语会有用，那么当该原语可用时，Herbie 实际上会使用它吗?

图6 的最后一列列出了 Herbie 在优化每个目标应用的内核时使用各个GrowLibm推荐原语的频次。更具体地说，我们为 Herbie 编写了三个新的平台定义文件，每个目标应用各一个，列出该应用的所有 GrowLibm推荐原语及其成本和精度函数。GrowLibm推荐的所有原语都会被使用，唯独cosquot 和sinquot 除外（如上文所述，它们速度太慢，不适合实际使用）。总之，所有目标应用都数十次使用了 GrowLibm推荐的原语，这表明这些由专家编写的实现确实有助于准确、高效地实现内核。CoolProp 的表现尤其突出。尽管 GrowLibm 仅为 CoolProp 推荐了一个原语pow1ms，但这个原语在 CoolProp 的内核中被使用了 87 次，这表明它非常实用，以至于许多内核会多次调用它。这很可能是因为该原语的专家实现比朴素实现更快、精度更高。简而言之，GrowLibm推荐的库原语的专家实现有助于对我们的所有目标应用进行超优化。

6.4 RQ4：GrowLibm 建议的原语能提升内核性能吗?

专家实现GrowLibm建议的原语应能通过两种方式拓展可达的速度/精度前沿：在任意给定精度下，内核都应运行得更快；同时，可达到的最佳精度也应有所提升。图7 检验第一项主张；图8 检验第二项。

图 7 比较了各项目内核在使用和不使用 GrowLibm 推荐原语的专家实现时的速度。每条曲线汇总了至少使用一种 GrowLibm 推荐原语的所有内核，绘制加速比随精度水平变化的曲线。使用GrowLibm 推荐的原语，在任何给定精度水平下都能显著降低成本。

[4]按照 Herbie 的度量，这意味着朴素实现的指数经常不正确。[5]我们的专家确实将准确实现这两个原语的任务称为一个“有趣的谜题”。

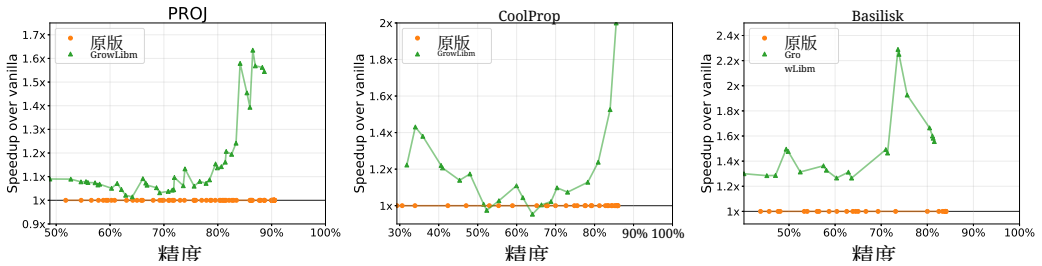


图 7。PROJ、CoolProp 和 Basilisk 的加速曲线；数值越高越好。每条曲线针对给定的精度水平，对比使用和不使用 GrowLibm 建议的库原语时 Herbie 的成本估计。具体而言，每条曲线汇总目标应用程序中使用了 GrowLibm 建议的库原语且达到给定平均精度水平的所有内核。GrowLibm 建议的库原语显著提升了整个精度范围内实现的速度；在高精度下，加速比最大，此时 GrowLibm 建议的原语的专家实现最为重要。

尽管许多专家编写的实现比朴素实现慢，但情况依然如此。由于这些实现的精度高得远超朴素实现，因此 Herbie 能够降低内核其他部分的精度，弥补损失的性能，甚至进一步提升。更确切地说，在 PROJ 基准测试中，经过GrowLibm增强的平台定义使 Herbie 在精度水平高于 80% 时，最高可实现 1.6× 的加速比。CoolProp 也有相同情况，最高加速比可达 2×。对于 Basilisk，其GrowLibm 推荐的原语具有精度较高的朴素实现，但专家实现速度更快，因此加速主要集中在 55%–80% 的中等精度水平，可实现最高 2.2×的加速比。[6] 简言之，GrowLibm 推荐原语的专家实现能够在广泛的精度水平上带来显著加速。

这些GrowLibm 建议的原语还使误差低于 Herbie 单独能够达到的水平。图 8 通过绘制 Herbie 在三个应用程序中每个内核上达到的最小误差，展示了这一点；其中 Herbie 至少使用了一个GrowLibm 建议的原语。在几乎所有此类内核中，误差都显著降低，这表明 GrowLibm 建议的原语和专家实现的原语不仅能在任意给定误差水平下提高加速比，还能使误差低于 Herbie 单独能够达到的水平。对于少数内核，误差确实会增加，这可能是因为 GrowLibm 建议的库原语会影响 Herbie 的内部搜索过程，但这些增幅很小，而降幅通常非常显著。例如，在 PROJ 的`init-K`块中，GrowLibm 建议的库原语（如专家实现的`invgud`和`log1pmd`）可将误差从大约 28 位（约 44% 的误差，即 56% 的精度；除非 $|\phi|$和 $|\phi_p|$ 接近 1，否则几乎无法使用）降低到大约 4 位（约 6% 的误差，即 94% 的精度）。Herbie 以位为单位测量误差，其中每一位误差对应结果中一位不可靠；图 8 将误差绘制为 64 位结果的百分比，即误差位数除以 64。

6.5 研究问题5：学习得到的原语能否部署到编译型应用程序中?

研究问题1至4表明，GrowLibm 的原语在内核级很有用；本问题探讨它们是否也能在编译后的应用程序代码中被识别和部署。

[6]在某些精度要求下，CoolProp 的运行速度略有下降；我们认为这主要是噪声所致，因为 GrowLibm 建议的原语会让 Herbie 以不同方式探索其搜索空间。当然，用户可以使用不同的随机种子多次运行 Herbie，也可以分别在启用和不启用 GrowLibm 建议的内核时运行，然后选取最佳实现。

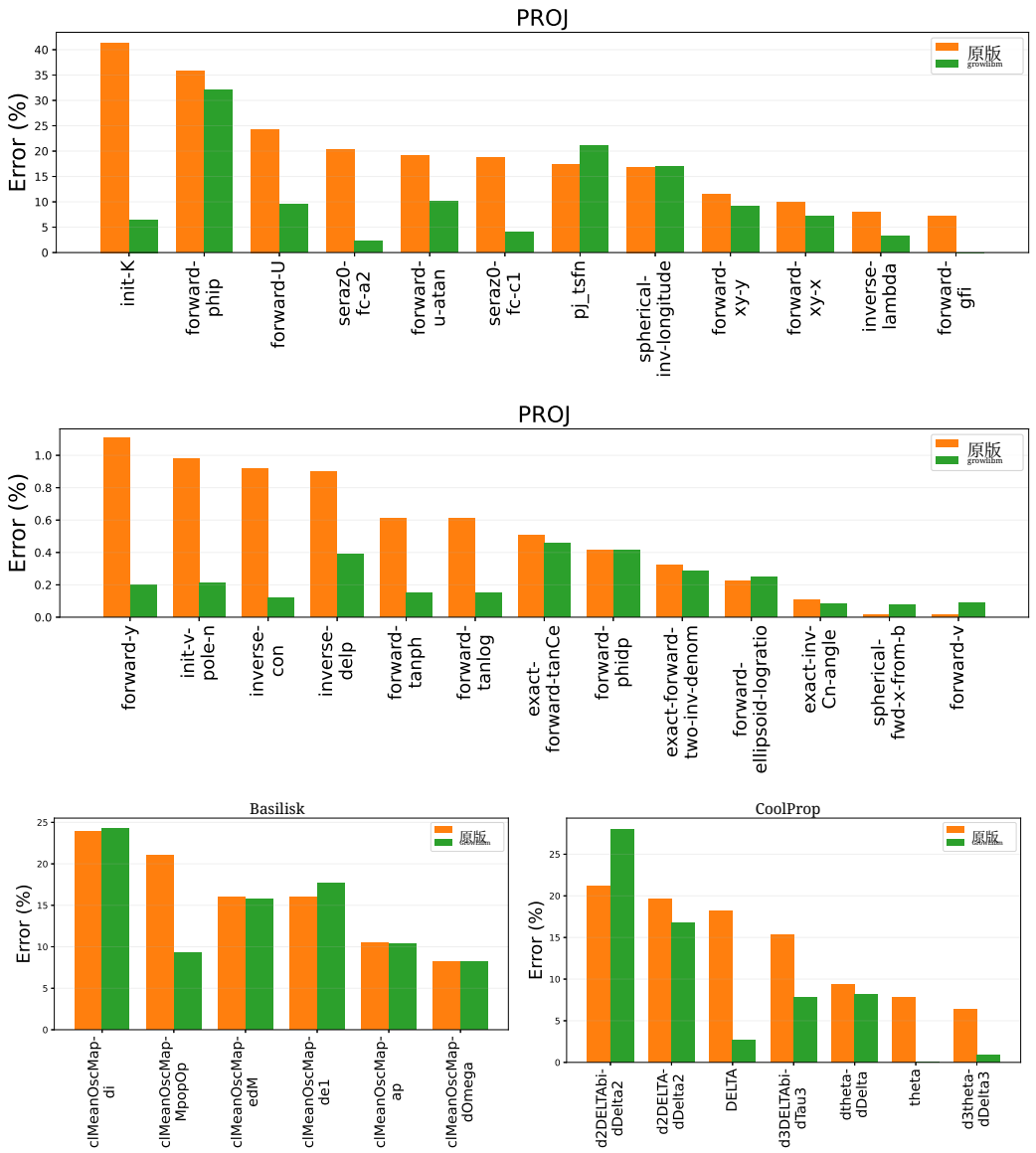


图 8. 对于三个目标应用程序中采用 GrowLibm 所建议库原语的内核，Herbie 所实现的最低误差。误差以 64 位结果的百分比表示（误差位数除以 64）；数值越低越好。为便于阅读，PROJ 内核分为两张图表（高误差和低误差）。GrowLibm 所建议的库原语能够显著降低大多数内核的误差。

测试整个应用程序的性能需要与目标应用程序的构建系统集成。由于使用 Herbie 完成这项工作极具挑战性，我们转而开发了一个 LLVM 过程，用于重写 LLVM 中间表示，以使用 GrowLibm 选定的原语。与 Herbie 一样，该过程可以应用代数恒等式并进行重写，但由于它在 LLVM 编译流水线内部运行，因此能力必然更受限。

投影	log1pmd	invgud	hypot	总计
krovak	0	0	0	0
omerc	1	1	0	2
som	0	1	4	5
somerc	8	10	0	18
tmerc	1	0	0	1
总计	10	12	4	26

图 9. 原型 LLVM 过程在五种 PROJ 投影中针对 GrowLibm 选定的各种原语执行的替换次数：Krovak（krovak）、斜轴墨卡托投影（omerc）、空间斜轴墨卡托投影（som）、瑞士斜轴墨卡托投影（somerc）和横轴墨卡托投影（tmerc）。

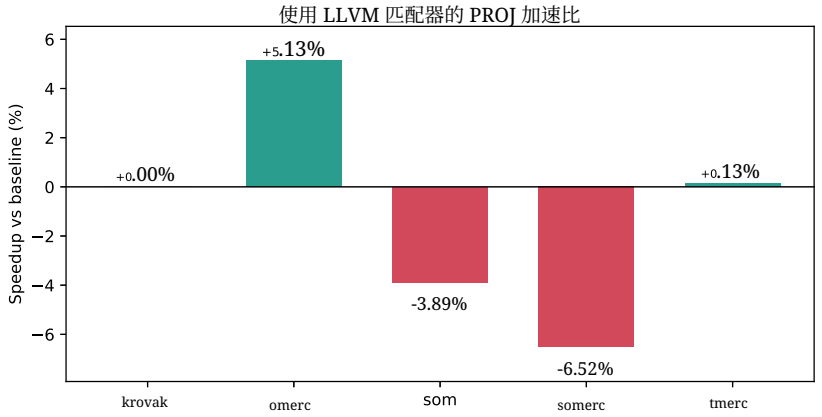


图 10. 五种 PROJ 投影的加速比。使用我们的原型 LLVM 过程进行编译，并在 10,000,000 个点上进行测试。数值越高越好。

该过程会在每个 LLVM 基本块上运行，遍历每条 IR 指令，并使用 LLVM 的模式匹配器来选择特定的浮点指令。对于每条指令，我们会将以该指令为根的指令有向无环图与一系列模式进行比较；如果匹配，就将其重写为对某个 GrowLibm 选定原语的调用。这些模式允许浮点重结合和使用分配律；例如，表达式 $(2x + \pi)/4$（invgud 的一部分）通常会被优化为 $0.5x + \pi/4$ 或 fma(x，0。5，π/4)，其中 $\pi/4$ 是一个常数。我们的 LLVM 过程会与所有这些模式进行匹配。找到匹配项后，该指令会被替换为一条调用指令，其名称指向相应的 GrowLibm 建议库原语。需要强调的是，这个 LLVM 过程只是一个原型，功能远不如 Herbie 完备，并且仅纳入了与 PROJ 相关的三个 GrowLibm 选定原语（log1pmd、invgud 和 hypot）。与 Herbie 不同，它也没有尝试纳入成本模型。因此，使用这个 LLVM 过程得到的结果旨在展示：GrowLibm 选定的原语能够在编译后的应用程序中被识别和部署，并对速度和精度产生可测量的影响。

图 9 统计了该遍在五种地图投影中执行的替换次数。匹配器找到了 26 个使用位置，比图 6 中对应的 Herbie 用法更少，因为 LLVM 中间表示中的模式匹配从根本上说比超优化的能力更有限。我们使用新的 LLVM 过程构建了 PROJ 应用程序，将其与专家实现库链接，并将结果连接到一个驱动程序，该程序对五种地图投影各执行 10

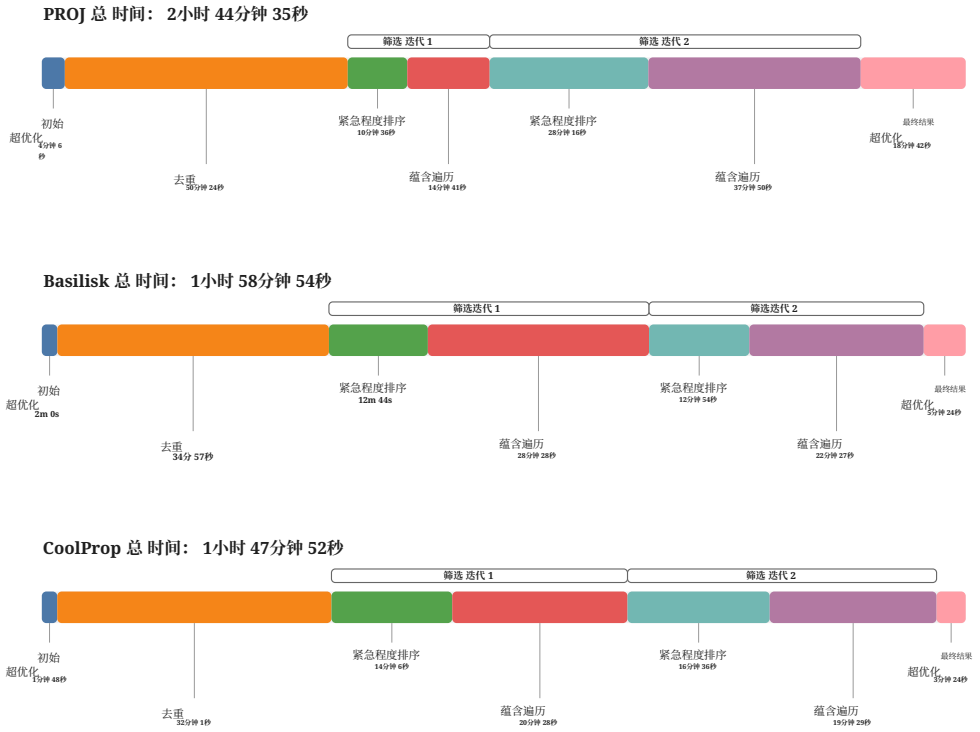


图11. GrowLibm 在 PROJ、Basilisk 和 CoolProp 应用程序中的各阶段时间线。

百万个点。随后，我们在一台配备 18 GB 内存、运行 macOS 15.3.1 并使用 Apple Clang 17.0.0 编译的 Apple M3 Pro Macbook Pro 上测量了生成程序的运行时间。

图 10 展示了结果。斜轴墨卡托投影（omerc）和横轴墨卡托投影（tmerc）的加速比分别为 5.1% 和 0.1%，同时也受益于由GrowLibm选出的原语实现所带来的更高精度。这是双赢：速度更快，而且精度更高。空间斜轴墨卡托投影（som）和瑞士斜轴墨卡托投影（somerc）的替换位置较多（分别为 5 个和 18 个，主要是invgud），其性能分别下降了 3.9% 和6.5%。这是因为invgud的专家实现精度显著更高，但速度比其所替代的朴素代码更慢。这种以精度换速度的权衡是否值得，取决于应用程序层面的决策；对于重视地理空间精度的制图库而言，这种权衡或许是值得的。krovak 投影没有替换位置，因此性能没有变化。

简而言之，LLVM 匹配器表明，GrowLibm 选定的原语可以被识别并部署到编译后的应用程序中，并对速度和精度产生可测量的影响。将 LLVM 过程与成本模型集成后，便可仅在权衡有利时选择性地应用替换。

6.6 研究问题6：GrowLibm 需要运行多长时间?

要让 GrowLibm 实用，完整流水线必须在合理的时间内运行完毕。

图 11 展示了 GrowLibm在三个案例研究中的运行时间。最长的一次运行发生在 PROJ 上，耗时略少于三小时。我们认为，对于当前任务而言，这样的运行时间可以接受：实现选定的原语肯定会花更久——事实上，仅在数十亿个输入上测试它们就花了我们更长时间——而且也不需要经常重复。图11 进一步将运行时间细分为七个步骤：使用 Herbie 进行初始超优化并导出所有生成的候选项；候选项去重，这是单个步骤中最耗时的一步；两个选择阶段，每个阶段都需要进行紧急程度排序和蕴含遍历；最后，使用选定的原语进行最终超优化。这样的细分凸显了 GrowLibm的大小、紧迫性和频率启发式的重要性；这些启发式大幅减少了两个选择阶段所需的超优化次数，从而缩短了 GrowLibm的运行时间。两个选择阶段已经占据了 GrowLibm大部分的运行时间。我们推测，在相同条件下，按照第 4节所述的简单贪心选择算法将耗时数周。请注意，在 PROJ 中，第二个选择阶段和最终超优化都明显比第一个选择阶段和初始超优化耗时更长。这是由于对选定候选项使用了偏乐观的精度模型： GrowLibm 在这些成本模型中使用 Rival 区间算术库，因此显著增加了运行时间。

7 相关工作

本节回顾了库学习、数学库合成、基于等式图的系统以及超优化器方面的既有工作，并阐明了我们的工作与这些研究方向的联系和区别。

7.1 库学习

库学习是一种从输入程序语料库中提取共同结构的技术。DreamCoder [14] 使用库学习来生成抽象，这些抽象随后可用于解决其他任务。它使用版本空间 [28] 来高效存储程序语料库。后续的库学习工作 [6] 使用基于等式图的反统一来生成候选项，并引入等式理论，使库学习能够更好地适应语法变化。它还使用一种依赖公共子表达式消除的等式图提取算法。Stitch [4] 是另一种库学习工具，能够为所发现的抽象提供最优性保证，但不支持等式理论。GrowLibm 使用生成与选择循环来学习抽象，与先前的工具 [6]类似。

虽然库学习从一组程序中发现抽象，但以往的研究也关注其他形式的抽象发现。例如，Shapemod [23] 学习用于表达三维形状的宏，而 Szalinski [32] 则利用基于等式图的重写，从扁平化三维模型中发现映射与折叠。Szalinski 等工具背后的原理与 GrowLibm 类似：它们借助重写进行程序优化，从而发现新的抽象。

7.2 自动数学库生成

虽然 GrowLibm 不会生成其所提出原语的实现，但它补充了以往自动构建此类实现的研究。MegaLibm [5] 提出了一种用于实现此类原语的领域特定语言。MegaLibm 的核心理念是将高层算法结构与底层浮点调优决策分离。其他工具，如 RLIBM [29, 41] ，则利用线性规划自动生成正确舍入的数学库函数的实现。

这些系统关注的是如何实现选定的原语。GrowLibm 解决的是一个正交且更早期的问题：一开始应该实现哪些原语。这凸显了 GrowLibm 与自动实现工具结合使用的价值。

7.3 使用等式图进行综合与优化

最初为在定理证明器中高效表示同余关系而开发的等式图数据结构 [34], 已证明可通过等式饱和有效地用于程序优化 [55]。 egg 库 [60] 提供了一个快速且可复用的等式图库，已应用于众多系统 [6, 14, 20, 24, 32, 33, 39, 40, 45, 46, 51–53, 56–58]。尽管这些系统来自不同领域，但它们都利用等式饱和来执行程序综合、优化和等价性检查等任务。

由等式图驱动的重写规则综合工具（如 Ruler [33] 和 Enumo [39] ）大致遵循与 GrowLibm 类似的生成—筛选循环。Ruler 和 Enumo 的目标都是综合重写规则，而 GrowLibm 则识别潜在数学原语，以拓展内核实现中最优速度—精度权衡的前沿。GrowLibm 高度依赖 Herbie 基于等式图的功能 [40]。具体来说，GrowLibm 的候选项生成会利用 Herbie 基于等式图的项重写，在等价项中进行搜索；GrowLibm 的去重则会利用 Herbie 基于等式图的等价性检查。

7.4 超优化编译器

超优化编译器会在程序构成的搜索空间中进行搜索，以生成更好的代码 [19, 31]。这与普通优化编译器不同，后者会应用一组固定的变换。数十年来，研究人员一直在开发超优化器。Bansal 和 Aiken [1] 开发了一个针对 x86 汇编代码的超优化器，并依靠穷举枚举来探索程序空间，通过规范化来剪枝搜索空间。STOKE [48, 49] 是一个超优化器，它依靠随机搜索有效地探索程序空间。Denali 超优化器 [25] 首次展示了如何利用基于等式图的重写来实现超优化器。Souper [47] 是一个面向 LLVM 的中端超优化器。它利用程序综合来发现具体优化，并通过巧妙地缓存综合结果来实现可扩展性。Minotaur [30] 在 Souper 的基础上构建，利用程序综合来搜索低成本组件。GrowLibm 的核心思想是重新利用超优化器中常见的机制，发现可复用原语，从而改进多个内核的实现。据我们所知，此前的工作尚未探索过这一思路。

8 结论

本文提出了数值库学习这一问题：给定一个由浮点内核组成的工作负载，哪些新原语值得由专家实现？事实证明，数值超优化器已经具备解决这一问题所需的机制。它们的搜索会枚举候选项，等价性推理会去重并进行泛化，而成本模型则会估算反事实效用。GrowLibm 将这一发现付诸实践，把 Herbie 改造为库学习器。

在 PROJ、CoolProp 和 Basilisk 中，GrowLibm 找到了 log1pmd、invgud 和 pow1ms 等紧凑原语，专家可以使用标准数值技术实现它们。Herbie 大量复用了这些实现，在精度固定时将速度与精度前沿最多提升 2.2×，并在最佳情况下将精度从 56.0% 提高到 93.5%。一个原型 LLVM 匹配器表明，学习得到的原语也可以在编译代码中被识别和部署。

我们的方法存在明显的局限性。GrowLibm 的排序启发式方法未能充分考虑实现难度：两个建议的原语（sinquot 和 cosquot）尽管在频率、紧迫性和大小方面排名靠前，最终却需要成本高得令人望而却步的多字算术。将实现难度估计纳入排序或许可以解决这一问题。LLVM 部署原型缺少成本模型，目前还无法针对每个站点进行速度与精度权衡；若能与 Herbie 更紧密地集成，就可以有选择地在权衡有利的地方进行部署，而

我们认为，尽管库学习框架可以推广到数值计算以外的领域，但我们的评估仅使用了 Herbie，且只涉及数值工作负载。

这些局限也为未来研究指明了方向。当前流水线需要专家指导才能实现建议的原语；代码生成模型可以部分自动化这一步，尤其是那些实现遵循区间缩减和多项式近似等标准模式的原语。接下来，自然也可以尝试使用其他超优化器进行库学习。像 STOKE 这样的工具 [48]（用于 x86）、Souper [47] 和 Minotaur [30]（用于 LLVM 中间表示），以及 Denali [25]（用于基于等式图的重写），都共享核心机制（搜索、等价性推理、成本模型），而 GrowLibm 将这些机制从 Herbie 中重新加以利用。将工作负载驱动的原语发现与通过代码模型实现的自动实现相结合，最终或许能够完全闭合这一循环：超优化器不仅能识别哪些原语有帮助，还能生成并验证这些原语的实现。

这篇论文的核心观点是，决定构建哪些原语，可以通过与利用现有原语优化代码相同的机制来回答。在我们研究的三种工作负载中，该机制找到了紧凑、可复用、可实现且确实有用的原语。

数据可用性声明

我们将提交 GrowLibm 的源代码以及用于生成 第6节 中结果的脚本，作为 OOPSLA 制品评估流程的一部分，并附上说明如何运行这些脚本的文档。我们评估中使用的所有基准测试均为开源。代码、脚本和基准测试集将公开发布。

参考文献

[1] Sorav Bansal 和 Alex Aiken。2006年。自动生成窥孔超优化器。载于《第12届编程语言与操作系统体系结构支持国际会议论文集》（美国加利福尼亚州圣何塞）（ASPLOS XII）。计算机协会，纽约，美国，394–403。https://doi.org/10.1145/ 1168857.1168906[2] Ian H. Bell、Jorrit Wronski、Sylvain Quoilin 和 Vincent Lemort。2014年。纯流体与拟纯流体热物性评估及开源热物性库 CoolProp。《工业与工程化学研究》 53, 6 2014 2498 2508 10 1021 4033999 ( )，– 。https://doi.org/ . /ie Dorra Ben [3] Khalifa、Matthieu Martel 和 Assale Adjé。 。POP：2020混合精度浮点计算调优助手。载于《安全关键系统形式化技术》（FTSCS 2019）（《计算机与信息科学通讯》，卷 *1165* 77 94 10 1007 978 3 030 46902 3_5 ）。施普林格，– 。 [4] https://doi.org/ . / - - - - Matthew Bowers、Theo X. Olausson、Lionel Wong、Gabriel Grand、Joshua B. Tenenbaum、Kevin Ellis 和 Armando Solar-Lezama。2023年。面向库学习的自顶向下综合。《ACM 程序设计语言会议录》7，POPL，文章41（1月 2023 32 10 1145 3571234 ），页数。https://doi.org/ . / Ian Briggs、Yash Lad 和 Pavel Panchekha[5]。 。数学库函数的实现与综合。《ACM 2024程序设计语言会议录》 8, 32 2024 28 10 1145 3632874 POPL，文章（1月），页数。 [6] https://doi.org/ . / David Cao、Rose Kunkel、Chandrakana Nandi、Max Willsey、Zachary Tatlock 和 Nadia Polikarpova。2023年。babble：利用等式图与反统一学习更佳抽象。《ACM 程序设计语言会议录》7，POPL，文章14（1月 2023 29 10 1145 3571207 ），页数。https://doi.org/ . / Sylvain Chevillard、Mioara Joldeş 和 [7] Christoph Lauter。2010年。Sollya：数值代码开发环境。载于《第三届数学软件国际大会论文集》（ICMS'10）。施普林格出版社，28–31。[8] Wei-Fan Chiang、Mark Baranowski、Ian Briggs、Alexey Solovyev、Ganesh Gopalakrishnan 和 Zvonimir Rakamarić。 2017 300 315 10 1145 3009837 3009846 。严格浮点混合 [9] 精度调优（POPL）。– 。https://doi.org/ . / 2017. Nasrine Damouche 和 Matthieu Martel。 。Salsa：提升程序数值精度的自动化工具（AFM）。[10] Nasrine Damouche、Matthieu Martel 和 Alexandre Chapoutot。2017年。通过自动变换提升程序的数值精度。《软件工具与技术转移国际期刊》19，4（2017年），427–448。https://doi.org/10.1007/s10009-016-0435-0

[11] Nasrine Damouche、Matthieu Martel、Pavel Panchekha、Jason Qiu、Alex Sanchez-Stern 和 Zachary Tatlock。2016年。迈向浮点分析的标准基准格式与套件。（2016年7月）。[12] Eva Darulova 和 Viktor Kuncak。2014年。实数的可靠编译（POPL）。14页。http://doi.acm.org/10.1145/ 2535838.2535874[13] Arnab Das、Ian Briggs、Ganesh Gopalakrishnan、Sriram Krishnamoorthy 和 Pavel Panchekha。2020年。可扩展且严格的浮点误差分析。载于《2020年SC20：国际高性能计算、网络、存储与分析会议》（SC）。IEEE计算机学会，洛斯阿拉米托斯，加利福尼亚州，美国，1–14。https://doi.org/10.1109/ SC41405.2020.00055[14] Kevin Ellis、Catherine Wong、Maxwell Nye、Mathias Sablé-Meyer、Lucas Morales、Luke Hewitt、Luc Cary、Armando Solar-Lezama 和 Joshua B. Tenenbaum。2021年。DreamCoder：通过唤醒-睡眠库学习引导归纳式程序合成。载于《第42届ACM SIGPLAN程序设计语言设计与实现国际会议论文集》（线上会议，加拿大）（PLDI 2021）。计算机协会，纽约，美国，

835 850 10 1145 3453483 3454080 – 。https://doi.org/ . / . Oliver Flatt、Samuel Coward、Max Willsey、[15] Zachary Tatlock 和 Pavel Panchekha。2022年。基于同余闭包的小型证明。载于《第22届计算机辅助设计中的形式化方法会议论文集》（FMCAD）。维也纳工业大学学术出版社，

75 83 10 34727 2022 978 3 85448 053 2_13 – 。https://doi.org/ . / /isbn. - - - - Oliver [16] Flatt 和 Pavel Panchekha。 。让区间算术更能抵御溢出。载于《2023年IEEE第30届计算机算术研讨会》（ARITH）。2023

44 47 10 1109 58626 2023 00022 – 。https://doi.org/ . /ARITH . . [17] Oliver Flatt 和 Yihong Zhang。 。egglog实践：自动改进浮点误差。载于EGRAPHS 2023：E-Graph研究、应用、实践与人因研讨会。https://effect.systems/doc/egraphs- 2023-egglog/paper.pdf[18] FPBench项目。2021年。FPCore 2.0：标准浮点基准格式。https://fpbench.org/spec/fpcore- 2.0.html 访问于2026年3月。[19] Christopher W. Fraser。1979年。一种紧凑的机器无关窥孔优化器。载于《第6届ACM SIGACT-SIGPLAN编程语言原理研讨会论文集》（得克萨斯州圣安东尼奥）（POPL ’79）。计算机协会，纽约，美国，

1 6 10 1145 567752 567753 – 。https://doi.org/ . / . Bo-Yuan [20] Huang、Steven Lyubomirsky、Yi Li、Mike He、Gus Henry Smith、Thierry Tambe、Akash Gaonkar、Vishal Canumalla、Andrew Cheung、Gu-Yeon Wei、Aarti Gupta、Zachary Tatlock 和 Sharad Malik。2024年。利用形式化软硬件接口进行加速器设计的应用级验证。《ACM设计自动化与电子系统汇刊》

29, 2, 35 2024 25 10 1145 3639051 文章（2月），页数。https://doi.org/ . / Arnault [21] Ioualalen 和 Matthieu Martel。 。Sardana：数值精度优化自动化工具。载于第15届GAMM-IMACS科学计算、计算机算术与验证数值国际研讨会（SCAN）。[22] Anastasiia Izycheva 和 Eva Darulova。2017年。浮点算术的可靠相对误差界（FMCAD）。2012

15 22 10 23919 2017 8102236 – 。https://doi.org/ . /FMCAD. . R. Kenny Jones、[23] David Charatan、Paul Guerrero、Niloy J. Mitra 和 Daniel Ritchie。2021年。ShapeMOD：三维形状程序的宏操作发现。《ACM图形学汇刊》40，4，文章153（2021年7月），16页。https://doi.org/10.1145/ 3450626.3459821[24] R. Kenny Jones、Paul Guerrero、Niloy J. Mitra 和 Daniel Ritchie。2023年。ShapeCoder：从非结构化基元中发现视觉程序的抽象。《ACM图形学汇刊》42，4，文章49（2023年7月），17页。https: //doi.org/ 10 1145 3592416 . / Rajeev Joshi、[25] Greg Nelson 和 Keith Randall。2002年。Denali：目标导向超优化器。载于《ACM SIGPLAN 2002年程序设计语言设计与实现会议论文集》（德国柏林）（PLDI ’02）。计算机协会，纽约，美国，

304 314 10 1145 512529 512566 – 。[26] https://doi.org/ . / . Patrick W. Kenneally、Scott Piggott 和 Hanspeter Schaub。 。Basilisk：灵活、可扩展且模块化的天体动力学仿真框架。《航空航天信息系统期刊》2020

17, 9 2020 10 2514 1 010762 ( )。[27] https://doi.org/ . / .I Bhargav Kulkarni 和 Pavel Panchekha。 。结合条件数与预言机进行精确浮点调试。载于《2025年IEEE第32届计算机算术研讨会》（ARITH）。[28] Tessa A. Lau、Pedro Domingos 和 Daniel S. Weld。2000年。版本空间代数及其在示例编程中的应用。载于《第十七届国际机器学习会议论文集》（ICML ’00）。Morgan Kaufmann出版社公司，旧金山，加利福尼亚州，美国，527–534。[29] Jay P. Lim 和 Santosh Nagarakatte。2021年。适用于32位浮点表示的高性能正确舍入数学库。载于《第42届ACM SIGPLAN程序设计语言设计与实现国际会议论文集》（线上会议，加拿大）（PLDI 2021）。计算机协会，纽约，美国，2025

359 374 10 1145 3453483 3454049 – 。https://doi.org/ . / . Zhengyang Liu、Stefan Mada 和 John Regehr。 。Minotaur：面向SIMD的合成超优化器。《ACM 程序设计语言会议录》8，OOPSLA2，文章326（2024年10月），25页。https://doi.org/10.1145/3689766 [30] 2024

[31] Henry Massalin。1987年。《超优化器：一窥最小程序》。载于第二届国际编程语言与操作系统架构支持会议论文集（美国加利福尼亚州帕洛阿尔托）（ASPLOS II）。计算机协会，纽约，美国，122 126 10 1145 36206 36194 –。[32] https://doi.org/ 。/ 。Chandrakana Nandi、Max Willsey、Adam Anderson、James R. Wilcox、Eva Darulova、Dan Grossman 和 Zachary Tatlock。2020年。《利用等式饱和与逆变换合成结构化 CAD 模型》。载于第41届 ACM SIGPLAN 编程语言设计与实现会议论文集（英国伦敦）（PLDI 2020）。计算机协会，纽约，美国，31–44。https://doi.org/10.1145/3385412.3386012[33] Chandrakana Nandi、Max Willsey、Amy Zhu、Yisu Remy Wang、Brett Saiki、Adam Anderson、Adriana Schulz、Dan Grossman 和 Zachary Tatlock。2021年。《使用等式饱和推断重写规则》。ACM 程序设计语言会议录 5，OOPSLA，文章 119 2021 28 10 1145 3485496（10月），页数。https://doi.org/ 。/ Charles Gregory Nelson。[34] 1979《程序验证技术》。博士论文。斯坦福大学。[35] K-C Ng 博士。1993年。FDLIBM。http://www.netlib.org/fdlibm/readme [36] Takeshi Ogita、Siegfried M. Rump 和 Shin'ichi Oishi。2005年。《精确求和与点积》。SIAM 科学计算杂志 26, 6 2005 1955 1988 10 1137 030601818 （），–。https://doi.org/ 。/ 开放地理空间基金会。n. d. 。[37] PROJ。https://proj.org/ 访问于2026年3月。[38] OSGeo。 [n. d.]。开放地理空间基金会。https://www.osgeo.org/ 访问于2026年3月。[39] Anjali Pal、Brett Saiki、Ryan Tjoa、Cynthia Richey、Amy Zhu、Oliver Flatt、Max Willsey、Zachary Tatlock 和 Chandrakana Nandi。2023年。《定制化等式饱和理论探索》。ACM 程序设计语言会议录 7，OOPSLA2，文章 258 2023 29 10 1145 3622834（10月），页数。https://doi.org/ 。/ Pavel Panchekha、Alex[40] Sanchez-Stern、James R. Wilcox 和 Zachary Tatlock。2015年。《自动提高浮点表达式的精度》（PLDI）。[41] Sehyeok Park、Justin Kim 和 Santosh Nagarakatte。2025年。《无需担心应用程序的舍入模式的正确舍入数学库》。ACM 程序设计语言会议录 9，PLDI，文章 229（2025年6月），24页。https://doi.org/ 10 1145 3729332。/ PROJ 贡献者。n. d.。《瑞士斜轴墨卡托投影实现》。https://github.com/OSGeo/PROJ/blob/ master/src/projections/somerc.cpp[42] [ ] 访问于2026年3月。[43] 2025 10 5281 5884394 PROJ 贡献者。《PROJ 坐标变换软件库》。https://doi.org/ 。/zenodo。[44] 2026Siyuan Brant Qian、Vimarsh Sathia、Ivan R. Ivanov、Jan Hückelheim、Paul Hovland 和 William S. Moses。《快速而正确地思考：通过编译器增强自动重写数值代码》。载于2026年 IEEE/ACM 国际代码生成与优化研讨会论文集（CGO）。[45] Brett Saiki、Jackson Brough、Jonas Regehr、Jesus Ponce、Varun Pradeep、Aditya Akhileshwaran、Zachary Tatlock 和 Pavel Panchekha。2025年。《面向目标的实数表达式实现》（ASPLOS 2025）。计算机协会，纽约，美国，1069 1083 10 1145 3669940 3707277 –。https://doi.org/ 。/ 。Brett Saiki、Oliver Flatt、[46] Chandrakana Nandi、Pavel Panchekha 和 Zachary Tatlock。2021年。《结合精度调节与重写》。载于2021年 IEEE 第28届计算机算术研讨会（ARITH）。[47] Raimondas Sasnauskas、Yang Chen、Peter Collingbourne、Jeroen Ketema、Gratian Lup、Jubi Taneja 和 John Regehr。2018 1711 04422 [ ] 1711 04422 。《Souper：一种合成式超优化器》。arXiv：。cs.PL https://arxiv.org/abs/ 。[48] 2013Eric Schkufza、Rahul Sharma 和 Alex Aiken。《随机超优化》。载于第十八届国际编程语言与操作系统架构支持会议论文集（美国得克萨斯州休斯敦）（ASPLOS '13）。计算机协会，纽约，美国，305–316。https://doi.org/10.1145/2451116.2451150[49] Eric Schkufza、Rahul Sharma 和 Alex Aiken。2014年。《利用可调精度对浮点程序进行随机优化》（PLDI '14）。[50] Alexei Sibidanov、Paul Zimmermann 和 Stéphane Glondu。2022年。《CORE-MATH 项目》。载于2022年 IEEE 第29届计算机算术研讨会（ARITH）。26 34 10 1109 54963 2022 00014 –。https://doi.org/ 。/ARITH 。 。[51] 2021Eytan Singher 和 Shachar Itzhaky。《由演绎合成驱动的理论探索》。载于《计算机辅助验证：第33届国际会议，CAV 2021，线上活动，2021年7月20日至23日，论文集，第二部分》。施普林格出版社，柏林，海德堡，125 148 10 1007 978 3 030 81688 9_6 –。https://doi.org/ 。/ - - - - Gus Henry Smith、Benjamin Kushigian、Vishal Canumalla、Andrew Cheung、Steven Lyubomirsky、Sorawee Porncharoenwase、René J ust、Gilbert Louis Bernstein 和 Zachary Tatlock。2024年。《使用草图引导的程序合成进行 FPGA 技术映射》。载于第29届 ACM 国际编程语言与操作系统架构支持会议论文集，第2卷（美国加利福尼亚州拉霍亚）（ASPLOS '24）。计算机协会，纽约，美国，416–432。https://doi.org/10.1145/3620665.3640387[53] Gus Henry Smith、Andrew Liu、Steven Lyubomirsky、Scott Davidson、Joseph McMahan、Michael Taylor、Luis Ceze 和 Zachary Tatlock。2021年。《通过访问模式进行纯张量程序重写（表示法精粹）》。载于第5届 ACM SIGPLAN 机器编程国际研讨会论文集（线上会议，加拿大）（MAPS 2021）。计算机协会，纽约，美国，21–31。https://doi.org/10.1145/3460945.3464953

[54] Robert Tarjan。1972年。《深度优先搜索与线性图算法》。SIAM J. Comput. 1，2（1972），146–160。https://doi.org/ 10 1137 0201010 。/ Ross Tate、Michael Stepp、Zachary Tatlock 和 Sorin Lerner。2009年。[55]《等式饱和：一种新的优化方法》。载于第36届 ACM SIGPLAN-SIGACT 编程语言原理研讨会年会论文集（美国佐治亚州萨凡纳）（POPL '09）。计算机协会，纽约，美国，264–276。https://doi.org/10.1145/1480881.1480915[56] Samuel Thomas 和 James Bornholt。2024年。《为可定制数字信号处理器自动生成向量化编译器》。载于第29届 ACM 国际编程语言与操作系统架构支持会议论文集，第1卷（美国加利福尼亚州拉霍亚）（ASPLOS '24）。计算机协会，纽约，美国，

19 34 10 1145 3617232 3624873 –。https://doi.org/ 。/ 。Alexa VanHattum、Rachit Nigam、Vincent T. Lee、James Bornholt 和 Adrian Sampson。2021年。[57]《通过等式饱和实现数字信号处理器向量化》。载于第26届 ACM 国际编程语言与操作系统架构支持会议论文集（美国线上）（ASPLOS '21）。计算机协会，纽约，美国，

874 886 10 1145 3445814 3446707 –。https://doi.org/ 。/ 。[58] Yisu Remy Wang、Shana Hutchison、Jonathan Leang、Bill Howe 和 Dan Suciu。2020。《SPORES：通过关系等式饱和实现大规模线性代数的和积优化》。Proc. VLDB Endow. 13，12（2020年7月），1919–1932。https://doi.org/ 10 14778 3407790 3407799 。/ 。维基百科贡献者。n. d.。《古德曼函数》。[59] https://en.wikipedia.org/wiki/Gudermannian_function [ ] 访问于2026年3月。[60] Max Willsey、Chandrakana Nandi、Yisu Remy Wang、Oliver Flatt、Zachary Tatlock 和 Pavel Panchekha。2021年。《egg：快速且可扩展的等式饱和》。ACM 程序设计语言会议录 5，POPL，文章23（2021年1月），29页。https://doi.org/10.1145/3434304