Towards Automated Reasoning Chains for Verification of LLM-Generated Scientific Code

Quentin Oschatz
Carnegie Mellon University
Pittsburgh, USA
qoschatz@cmu.edu

Naifeng Zhang
Carnegie Mellon University
Pittsburgh, USA
naifengz@cmu.edu

Mike Franusich
SpiralGen, Inc.
Pittsburgh, USA
mike.franusich@spiralgen.com

Franz Franchetti
Carnegie Mellon University
Pittsburgh, USA
franzf@cmu.edu

***Abstract*—With the rise of Large Language Model (LLM) generated code, including in domains like scientific computing, ensuring not only syntactical, but also mathematical correctness, has become a critical task. Traditional formal methods approaches often struggle with the ambiguity of floating-point code, and full symbolic execution is extremely costly and limited. We propose a chain-of-reasoning approach that iteratively lifts basic semantics from code into the SPIRAL system and then establishes numerical equivalency to the desired mathematical operation. Here, we leverage the ample mathematical knowledge already formalized in SPIRAL to enable the system to recognize not just different implementations of the same algorithm but fully separate approaches to solving the given problem. The chain establishes tight error bounds on the output of given code with respect to the true continuous solution it approximates, quantifying all sources of error. We demonstrate this approach by establishing the correctness of a pseudospectral solver for a simple 1-dimensional Poisson problem.**

***Index Terms*—Numerical analysis, partial differential equations (PDEs), scientific computing, SPIRAL, large language models (LLMs)**

I. Introduction

Large Language Models (LLMs) have increasingly become a widespread tool for neural code generation, despite their tendency towards hallucinations and mistakes [1], [2]. While improvements have been made to their ability to generate semantically correct code, this is often insufficient, especially when generating code in the space of scientific computing. Here, numerical errors in algorithm implementation can be easy to make, hard to catch, and even more difficult to diagnose, especially with the added complexity of evaluating floating-point code. This can be a problem not just when attempting to generate new scientific kernel code as explored by Valero-Lara et al. [3], but also when translating code written in languages such as FORTRAN to more modern codebases as is the goal of projects such as TRACTOR [4].

Leaning on the well-known insight in mathematics and computer science that proving a solution correct is far easier than finding one, we propose a multi-step, end-to-end system to verify the numerical correctness of given code.

This material is based upon work supported by the U.S. Department of Energy, Office of Science under Award No. DE-SC0025645, by the National Science Foundation under Award No. 2431265, and by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112490517. Approved for public release: distribution is unlimited.

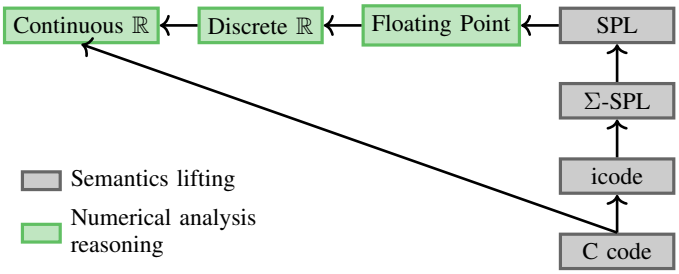

Fig. 1: Logical overview of the core steps in the pipeline connecting C code to the true, continuous solution of the given problem over the real numbers extending prior work [5].

Previous work [5] leveraged the SPIRAL system [6] to perform semantics lifting based on symbolic execution and program transformations. By restricting the problem domain to those understood by SPIRAL, Fast Fourier Transforms (FFTs) can be lifted from recursive C implementations. Reversing SPIRAL's rules-based, multi-tiered code generation system that traditionally lowers high level semantics into functioning code, the system can iteratively fuse operations into higher level functional blocks, until finally leveraging SPIRAL's database of linear transforms to recognize the components of a FFT. Due to the symbolic execution used for the fusion, minimal ambiguity is introduced by floating-point operations, and equivalency can be established. The methods introduced in this paper extend that work to support more complex mathematical algorithms consisting of a network of functional blocks.

Also utilizing the SPIRAL system, the pipeline follows a similar framework of exploiting symbolic execution and the database to recognize even higher level operations. However, complex scientific computing code comprised of numerous mathematical operations beyond ubiquitous operations like the FFT are infeasible to merely lift, as SPIRAL would need to be populated with an enormous database of all possible operations and their variations. Moreover, those operations may be implemented in mathematically distinct manners.

We propose a pipeline that can recognize and verify these more complex operations by constructing reasoning chains as follows. In Section III, the chain starts with basic C code and uses previous work to extract and verify basic

functional blocks. Next, the mathematical principles of the pseudospectral solver for the given problem are introduced in Section IV. Then, Section V describes a chain equating the lifted operators directly with the equation purportedly being solved by the code, quantifying and bounding all sources of error. Afterwards, techniques described by Zaliva in [7] can be leveraged to formalize the derivation directly in SPIRAL using theorem provers. We showcase all this by walking through the verification of a pseudospectral solver for 1-dimensional Poisson equations, a partial differential equation (PDE).

Contributions. This paper makes the following contributions towards an end-to-end pipeline:

- Describing an end-to-end pipeline for verifying the correctness of (LLM-generated) code.
- Demonstrating how to determine equivalency between a sequence of lifted semantics and the equation the code purports to solve.
- Sketching proofs for tight error bounds on the output of said code versus the true solution over the real numbers.
- Showcasing the high computational requirements involved in determining equivalency.

II. Background and Related Work

SPIRAL. SPIRAL is a code generation system capable of translating certain high level mathematical expressions into high-performance code by lowering it through multiple stages of abstraction. During these steps, SPIRAL uses a library of breakdown rules to perform substitutions that do not break correctness but can introduce parallelism or recursion, with the target hardware specifications guiding SPIRAL in these breakdown choices. For many of the earlier stages, SPIRAL keeps algorithms fully symbolic, allowing equivalent substitutions to be made and verified exactly.

As mentioned in Section I, SPIRAL was recently used to perform semantics lifting of C code into higher level math operations [5]. This was accomplished by reversing the usual SPIRAL workflow. Instead, code was iteratively lifted to higher and higher levels of abstraction using the same substitution rules in reverse.

Formal verification. Formally verifying the correctness of numerical software is extremely challenging, especially when floating-point code is involved. Appel and Kellison [8] presented a framework for verifying some floating-point programs written in C by using Rocq (formally Coq). They demonstrated the difficulty in formalizing the relationship between the outputs of a C program with finite precision and the true continuous solution over the real numbers, even when restricting the scope of supported data types and operations.

The HELIX project presented by Zaliva [7] addresses some of the problems that arise when formally proving floating-point code. It too builds upon SPIRAL to create a formally proven high-performance code generation system. Instead of using the system directly, the author builds a parallel pipeline with similar design characteristics. The algorithm is kept in the symbolic domain until most of the optimizations and transformations are applied, and the final code is generated. Even then, one of the transformations not formally verified was the step that introduces finite precision floating-point numbers, with the author citing the proof would necessitate the application of a broad range of numerical analysis techniques that would not easily generalize.

Pseudospectral methods. PDEs are equations involving partial derivatives of multivariate functions, and can be seen as a superset of ordinary differential equations (ODEs), which deal with derivatives of univariate functions. Typically, it is challenging or even infeasible to directly compute their solutions, requiring the use of numerical methods to find approximations.

There are multiple distinct approaches to approximating the operations and computing a solution, each with their own trade-offs in terms of accuracy and computational efficiency [9, Sec. 1]. Pseudospectral methods are one such category of approaches which aim to approximate differentiation as sums of smooth basis functions when given a problem on a simple—usually rectangular—grid.

Clearly, the choice of basis is, therefore, critical to the accuracy of the results. As described by Fornberg [9, Sec. 2.1], they should generally meet three requirements: rapid convergence of the function approximation for smooth functions, ease of determining the coefficients for the derivative of the function being approximated, and fast computation of the approximation at a given point. Depending on the boundary conditions imposed on the problem, different function classes may meet these requirements, with the FFT (and its sine and cosine versions) allowing trigonometric expansions and orthogonal Jacobi polynomials to meet the third requirement.

III. LLM Code Generation

We start with Listing 1, a block of C code that purports to implement a pseudospectral Poisson solver for a 1-dimensional problem with periodic boundary conditions.

Similar code could easily be generated with the help of ChatGPT using prompts like *"Hey ChatGPT, please generate C code for solving the Poisson equation in 1D with periodic boundary conditions using the pseudospectral method with FFTs and using nothing beyond the standard libraries."* Even with such a simple example, verifying its correctness is non-trivial; mistakes could be present in the implementation details of the FFTs, the computation of the convolution factors in Fourier space, or even the indexing. Moreover, even if approaches like fuzzy testing could be leveraged to determine correctness, merely comparing the output of such code will often not give a clear indication regarding the root cause of any errors.

Instead, semantics lifting from previous work [5] mentioned in Section II is employed to detect and lift the basic building blocks of the algorithm. Reversing the typical code generation pipeline in SPIRAL, code is repeatedly lifted to higher level abstractions via a combination of symbolic execution and transformations.

Small loops and code segments are symbolically executed to determine their exact characteristics, then small sections

```c
int i, j;
double k, *ksq, *data, *phi;
data = calloc(2*N, sizeof(double));
ksq = malloc(N * sizeof(double));
phi = malloc(N * sizeof(double));

// Initialize the data
initialize_rhs(data);
// Numerical Recipes in C, Third Edition
    [10]: FFT Function: four1()
four1(data, N, 1);

// Construct solver in Fourier space
for (i = 0; i < N; i++) {
  k = (i <= N / 2) ? i : i - N;
  // Construct lambda'
  ksq[i] = -(4 * (M_PI * M_PI) / (L * L)) * ( k *
      k );
}

// Apply the constructed solver to the input
for (i = 0; i < N; i++) {
  // At 0,0 don't normalize
  if (ksq[i] != 0) {
    data[2*i] *= 1/ksq[i];
    data[2*i+1] *= 1/ksq[i];
  }
}

// Zero out first component
data[0] = data[1] = 0.0;

// Perform inverse FFT
four1(data, N, -1);

// Copy data array back to phi and normalize
for (i = 0; i < N; i++) phi[i] = data[2*i] / N;
```

Listing 1: Sample C code implementing a 1D Poisson solver of size N. FFT code has been omitted to save space.

are fused into equivalent operations. For instance, a loop multiplying the respective elements of two arrays may be fused into a vector product, which can then replace the original loop.

Using SPIRAL's vast library of rule transforms originally created to allow the system to break apart large operations in parallelizable chunks, operations can be recursively fused until high level semantics are extracted. Here, this step could determine the presence and exact correctness of the two FFTs, as well as the pointwise multiplication in Fourier space. The latter could be represented as a multiplication with a diagonal matrix of factors.

As it stands, in a large codebase (e.g., one implementing a PDE solver), lifting can only identify, at the function level, what each code block represents. For example, it can recognize that a specific block implements linear operations such as FFTs or matrix multiplication. However, at the global level, current lifting techniques offer minimal insight into the semantics of how these blocks interact or what their combination represents. Theoretically, the rules-based system in SPIRAL that enabled lifting so far could be expanded by adding more rules that equate a specific sequence of blocks with a 1D Poisson solver. This approach, however, comes with numerous downsides.

Recognizing even just a subset of PDE solvers would require building a huge database. Previously, lifting was used to find a small set of widely-used kernels which represent the core building blocks for constructing a large range of scientific computing operations. Basic combinatorics reveals how quickly the number of possible compound operations grows. Given the number of PDEs, the variations introduced by boundary conditions, and higher dimensions, even restricting lifting to only those still leaves an enormous search space.

Furthermore, this system would be extremely brittle in the face of superficial changes in implementation or approach. PDEs can be solved in multiple distinct ways, and even the same type of solver may be implemented in slightly different ways. Accounting for all those variants with just pattern recognition would entail adding all possible variants to the database as well.

Lastly, the resulting system would also fail to provide any feedback beyond a match. Properties of the algorithm would need to be hard-coded, and the system would not provide much feedback for any deviation or mistake beyond a simple "no match". Without understanding how the solver in question is constructed, the system cannot robustly confirm its correct implementation or provide guarantees on the output.

IV. Derivation of the Solver

As discussed in Section II, the core of the pseudospectral solver is the representation of a function as a sum of basis functions. This can be showcased using the 1D Poisson equation on the interval $[0, L]$,

$$\Delta\varphi = g. \tag{1}$$

Trigonometric expansions are typically valid basis functions for the pseudospectral method [9, Sec. 2.1]. Assuming sufficiently smooth functions and periodic boundary conditions, both φ and g can therefore be represented as a sum of Fourier coefficients, e.g.,

$$\varphi(x) = \sum_{k=-\infty}^{\infty} \hat{\varphi}_k e^{2\pi i x(k/L)}. \tag{2}$$

Following the steps laid out by Fuka [11], the Poisson equation can then be reduced to finding those coefficients,

$$-\left(\frac{2\pi k}{L}\right)^2 \hat{\varphi}_k = \hat{g}_k. \tag{3}$$

From (2) it can be derived that $e^{2\pi i x(k/L)}$ are eigenfunctions of the Laplacian operator, with the corresponding eigenvalues $\lambda_k = -(2\pi k/L)^2$. Unfortunately, λ_k in the order $k = 0, 1, \ldots, n$ are not periodic on $[0, L]$, so a shift that reverses the order of the second half of the eigenvalues must be introduced,

$$\lambda'_k = \begin{cases} \lambda_k & k < \lfloor n/2 \rfloor - 1 \\ \lambda_{n-k} & k \geq \lfloor n/2 \rfloor - 1. \end{cases} \tag{4}$$

With all the parts in place, the full method is fairly straightforward: compute $\hat{g}$ using the discrete Fourier transform (DFT), divide pointwise by λ', then return to the time domain using the inverse discrete Fourier transform (IDFT).

$$\begin{bmatrix} 20\pi^2/16 & -4\pi^2/16 & -12\pi^2/16 & -4\pi^2/16 \\ -4\pi^2/16 & 20\pi^2/16 & -4\pi^2/16 & -12\pi^2/16 \\ -12\pi^2/16 & -4\pi^2/16 & 20\pi^2/16 & -4\pi^2/16 \\ -4\pi^2/16 & -12\pi^2/16 & -4\pi^2/16 & 20\pi^2/16 \end{bmatrix}$$

(a) The operator computed by taking the direct pseudoinverse of the discrete Laplacian.

$$\begin{bmatrix} -36\pi^2/16 & 1\pi^2/16 & 28\pi^2/16 & 4\pi^2/16 \\ 4\pi^2/16 & -36\pi^2/16 & 4\pi^2/16 & 28\pi^2/16 \\ 28\pi^2/16 & 4\pi^2/16 & -36\pi^2/16 & 4\pi^2/16 \\ 4\pi^2/16 & 7\pi^2/16 & 4\pi^2/16 & -36\pi^2/16 \end{bmatrix}$$

(b) The operator extracted from the pseudospectral solver.

Fig. 2: Comparing the operators resulting from inverting the forward operator versus applying the pseudospectral method for $n = 4$ on the domain $[0, 1]$.

Using SPIRAL, an operator combining all these steps in matrix form can be explicitly constructed and compared with the pseudoinverse of the discrete Laplacian operator. Figure 2, however, shows that these operators are far from identical.

It should be noted that using this algorithm is equivalent to computing a convolution of g and the time-domain version of the function from which λ' is sampled. In fact, this function is the Green's function for the Laplacian in the respective dimensionality, though this method never requires it to be explicitly constructed. Green's functions will not be discussed in detail here, though their properties will be briefly used in Subsection V-C. The main aspect of note relating to this paper is that convolving with them is an approach to solving PDEs and ODEs, i.e. given the Green's function $G(x, s)$ corresponding to (1) and the requisite boundary conditions,

$$\varphi(x) = G(s) \star g = \int G(x, s)g(s)ds. \tag{5}$$

V. Quantifying the Error

Having been able to identify key components of the solver via semantics lifting as discussed in Section III and equipped with the mathematical framework underlying the pseudospectral method, the next task of the pipeline is verifying the correctness of the actual solver. Continuing with the Poisson equation, this section will walk through quantifying the precise relationship between the true underlying solution of a PDE of this type and the output of the implemented discrete solver.

A. Discretization

The first divergence from the true solution arises from the discrete nature of the solver. This is unavoidable given the discrete operator, though the pseudospectral method and the use of the FFT imply that the functions are sampled uniformly.

Beyond the inherent information loss incurred by discretization, error can also be introduced should this process involve projecting samples onto an insufficient base. Fortunately, as noted in Section IV, the Fourier coefficients are eigenfunctions of the Laplacian operator at the heart of the Poisson PDE when constrained by periodic boundary conditions [12, pp. 35–37]. Hence, the Fourier transforms represent no distortion of the result not also present in the Laplacian.

In fact, since they share the same base, the discretization operation can be pulled out of the solver entirely and performed beforehand, yielding an equivalent formulation. Now the solver no longer introduces that error, mirroring the fact that the code accepts already discretized input vectors, not continuous function descriptions.

B. Numerical Equivalency to the Inverse Operator

So far, it has been verified that no unaccounted numerical error slipped in when constructing the framework for the solver or its base components. However, it must also be verified that the solver actually produces a correct solution. In short, the solver should be able to solve the equation.

To accomplish this, SPIRAL can explicitly construct the forward operator central to the PDE in question—the discrete Laplacian—in matrix form. Then, we use SPIRAL's built-in computer algebra system (CAS) functionality to determine its exact symbolic inverse.

Here it should be noted that periodic boundary conditions are insufficient to fully constrain the solution space, meaning that the operator is rank-deficient and has no pure inverse. Instead, we compute the pseudoinverse, which projects all solutions to a space where they have a mean of zero. Therefore, multiplying the constructed forward and inverse operators should not yield an identity matrix but rather a matrix of the following form,

$$\begin{bmatrix} (n-1)/n & -1/n & \dots & -1/n \\ -1/n & (n-1)/n & \dots & -1/n \\ \vdots & \vdots & \ddots & -1/n \\ -1/n & \dots & & (n-1)/n \end{bmatrix} \tag{6}$$

Taking the sequence of operations lifted from the raw C implementation—an FFT, a multiplication with a diagonal matrix whose main diagonal is composed of the λ's, followed by an inverse FFT—SPIRAL's symbolic engine can compare them to the constructed pseudoinverse, as well as to the forward operator. Given no indication so far that errors are introduced by utilizing the pseudospectral method when compared to a discretized version of the true solution, the two inverse operators should be identical. However, as already showcased in Figure 2, they are not. Curiously, both operators do correctly invert the forward operator, both producing the same matrix (6).

Upon closer inspection, the operators are in fact equivalent, merely separated by a rotation in their eigenspaces. This creates not only superficially different operators but also ones with seemingly incongruent eigenspaces (i.e. different eigenvalues and eigenvectors). The rotation matrix can be constructed in closed form, or directly by SPIRAL using Gaussian elimination, and automatically verified to be a valid

$$-417/4 + 18 \cdot \mathrm{E}(32) + 10 \cdot \mathrm{E}(32)^2 + 22 \cdot \mathrm{E}(32)^3 + 4 \cdot \mathrm{E}(32)^4 + 8 \cdot \mathrm{E}(32)^5 + \cdots - 22 \cdot \mathrm{E}(32)^{13} - 10 \cdot \mathrm{E}(32)^{14} - 18 \cdot \mathrm{E}(32)^{15}$$

Fig. 3: A single entry in the base change matrix for $n = 16$, a cyclotomic polynomial with $\mathrm{E}(n)$ being the nth root of unity.

base change. Since the base change is invertible, it adds no error and has no impact on the correctness of the output produced by the operators, hence why both successfully cancel the forward operator.

Here, another advantage of using a proper CAS such as SPIRAL emerges. SPIRAL supports cyclotomic polynomials, allowing them to be operated on symbolically. These polynomials are irreducible, with their roots being the primitive roots of unity, which appear in the base change and can emerge in Green's functions (though not for the 1D Poisson case). As seen in Figure 3, they rapidly grow in complexity, with most entries in the base change matrix having as many terms as the problem size n. Without robust support, the approximation error can grow rapidly, making the base change potentially non-invertible and exact verification impossible.

C. Lipschitz Bounds

Though the correctness of the output points by the solver implemented in C has now been established, no bounds have been placed on the range of continuous functions that could be represented by these points. Between these points, there are no limits on what values the true solution could take.

The main operation of the solver is a convolution—computed via the FFT for performance reasons—with a set of values originating from the DFT itself. In the time domain, these values are samples from the Green's function, though it is never explicitly constructed.

Green's functions are tied to specific operators and boundary conditions, and are definitionally functions which can be convolved with the input to a PDE to solve the equation. Since the values we use originate from this function, its inherent properties are useful to derive bounds on the output as well.

If a constraint is imposed on the input requiring it to be Lipschitz continuous on the interval of interest, the Dominated Convergence Theorem and Bochner's Theorem [13, pp. 25–30] can be used to show that the solution is also Lipschitz continuous. Since the pseudospectral method makes similar assumptions, this is a reasonable constraint. Then, utilizing the known features of the Green's function ψ,

$$\varphi(z) = \int g(z+x)\psi(x)dx \implies ||\varphi||_{Lip} \leq ||g||_{Lip}$$

Hence, its range between samples is strictly limited by the constraint on its derivative, and thus the space between samples in which it must lie is constrained.

D. Machine Error

The SPIRAL system enables the isolation of numerical sources of error as it is built upon symbolic execution. In normal code, however, operations are not executed symbolically but numerically, which leads to the accumulation of machine error due to finite precision. These errors can be quantified exactly by conducting forward error analysis. For this, interval arithmetic is used, counting the number of finite precision operations in the critical path of the input.

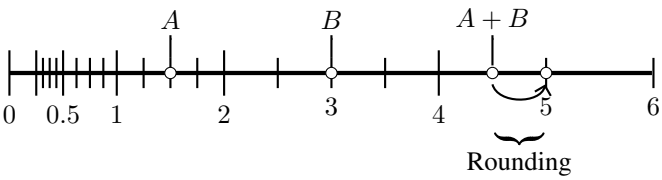

Fig. 4: Example of how error can be introduced by adding two floating-point numbers, encountered when the result moves into a range with coarser precision. This nonuniform distribution of precision is inherent to most typical floating-point formats, shown here by 4.5 not being representable.

The idea behind this stems from the distribution of floating-point numbers along the number line when using traditional representations such as IEEE-754. As showcased in Figure 4, a rounding error can occur after an addition operation involving floating-point numbers due to the nonuniformity of the representation, where less precision is dedicated to larger numbers. That way, two numbers between two different powers of two can yield an exact result that cannot be represented by the format and hence must be rounded.

Different operations interact differently with the error interval. While the add operations introduce a maximum of one rounding error equal to half the distance between numbers at the lowest precision, which shall be called one "tick", multiplication is different. Ordinarily, these operations would massively expand the error intervals, but since all constants are in $[0, 1]$, they cannot. In fact, these multiplications merely add a maximum of one tick, caused by the rounding of the result post-operation.

Hence, the total maximum error incurred by finite precision can be determined by counting the number of additions and multiplications performed in the critical path. Each FFT contains four such operations per butterfly component, with the entire FFT being composed of $\log n$ butterflies. Since two FFTs are used, we double that number and add three for the complex multiplication done in frequency space. In total, this means that the machine error is $8\log(n) + 3$ times the size of a tick.

VI. Results

To recap the pipeline, it begins with Listing 1, a snippet of code implementing the pseudospectral solver. Its basic semantics can be lifted by techniques in line with those presented in a previous paper [5]. This results in a sequence of symbolic operators, which can then be related to the known forward operator for this equation, namely the discrete Laplacian with periodic boundary conditions. While the extracted operator

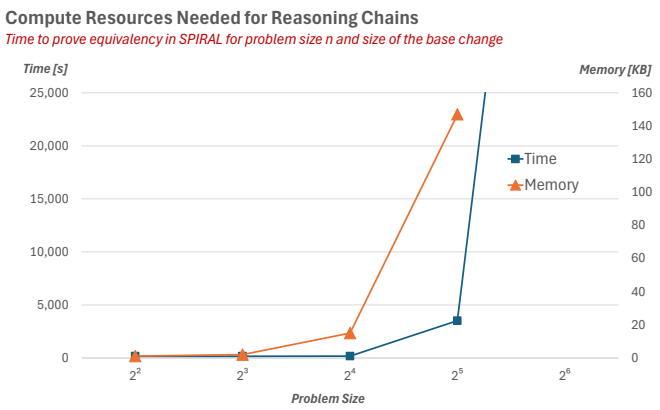

Fig. 5: Compute resources needed for determining the equivalency of the lifted operator sequence and the forward operator. Memory is the approximate space required for the cyclotomic polynomials of the base change, which grow rapidly with problem size. For $n = 2^6$, execution exceeded 24 hours and was then stopped. Cases 2^2–2^5 were run on a Regular Memory node with 512 GB RAM, case 2^6 on an Extreme Memory node, all at the Pittsburgh Supercomputing Center [14].

differs from the direct pseudoinverse of the forward operator, this is the result of the eigenspace rotation discussed in Subsection V-C. Therefore, SPIRAL can compute a change of basis between the operators, then have it verify that the result matches the requirements of a base change.

The base change can also be constructed directly using a closed-form expression. If this fails to match, meaning the operator is either incorrect or implemented in a radically different fashion, running the much slower Gaussian elimination would generate the base change and determine whether the operator is correct. This demonstrates the degree of robustness—even towards mathematically different solution methods—enabled by using a symbolic execution system like SPIRAL.

A major cost of this type of correctness checking, however, is the extreme computational resources it requires for larger problem sizes. Figure 5 shows how long it can take for SPIRAL to match these (already lifted) operators, and the scaling behind it. It also reveals the high memory scaling of just the exact base change, since this is composed of rapidly growing cyclotomic polynomials.

Though shortcuts like generating the base change directly can help speed up computation, these are limited tricks that must be manually added to the chain for a given operator. Even when operating on a supercomputer node, it remains intensive.

VII. Discussion

Key to the proposed pipeline is the automation of the reasoning chains discussed throughout this paper. Given a piece of code and a mathematical expression within a problem domain representable by SPIRAL (particularly PDEs), verification should be possible without problem-specific manual tweaks. For the 1D Poisson equation, the pipeline has been described here, including how to direct SPIRAL's existing systems to mostly automate it.

Critically, for a given (small) compile time n, the system can lift most of the operations (especially the FFTs) into the symbolic domain. A SPIRAL script exists that then confirms numerical equivalency with the forward operator. Using the IGen system [15], the forward error analysis described in Subsection V-D can be completed automatically and the exact intervals returned. In the future, the HELIX system [7] could be leveraged to formalize the entire reasoning chain directly in SPIRAL using Rocq, generating dependent types with inherent guarantees and correctness bounds.

Beyond that, crucial next steps involve expanding the problem domain beyond the Poisson equation and periodic boundary conditions. The insights gained from constructing this first chain and automating its components should generalize to more PDE types and boundaries, especially those that use sine and cosine transforms.

While the problem demonstrated here is rather simple, complexity in PDE solvers rises rapidly with dimensionality and boundary conditions. For instance, work by Fedeli et al. on the WarpX project [16] demonstrates the extreme complexity introduced by the interplay of higher dimensions, more complicated equations, and intricate boundary conditions. Here, while the underlying theory remains similar, manual verification quickly becomes intractable, and even automated experimental testing is difficult. Therefore, leveraging that common theory and iteratively extracting semantics can reduce these problems to similar collections of components, which then become part of reasoning chains.

The process of first creating the chain of reasoning demonstrated here required using both the CAS features built into SPIRAL and Mathematica [17]. Primarily, the latter was useful in exploring the different eigenspaces of the direct pseudoinverse and the pseudospectral operator. Mathematica allows for rapid computation of the entire eigenspace of an operator and its variants, as well as supporting a broader range of general mathematical functions in a friendlier interface. This streamlined the process of understanding the rotation and required base change. However, to actually construct the chain, SPIRAL alone suffices, as it is capable of generating the base change, either via direct construction or using Gaussian elimination, and confirming its properties.

Acknowledgment

The authors would like to thank Dr. Brian Van Straalen, Dr. Het Mankad, and Dr. Phillip Colella for many insightful discussions related to this project.

This work used Bridges-2 at Pittsburgh Supercomputing Center through allocation cis250137p from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

This paper was prepared as an account of work sponsored by an agency of the United States Government. Neither the

REFERENCES

[1] Z. Zhang, C. Wang, Y. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, and Z. Zheng, "LLM hallucinations in practical code generation: Phenomena, mechanism, and mitigation," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, Jun. 2025.

[2] Y. Tian, W. Yan, Q. Yang, X. Zhao, Q. Chen, W. Wang, Z. Luo, L. Ma, and D. Song, "CodeHalu: Investigating Code Hallucinations in LLMs via Execution-based Verification," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 24, pp. 25 300–25 308, Apr. 2025.

[3] P. Valero-Lara, W. F. Godoy, K. Teranishi, P. Balaprakash, and J. S. Vetter, "ChatBLAS: The First AI-Generated and Portable BLAS Library ," in *Proceedings of the SC '24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '24. Atlanta, GA, USA: IEEE Press, Feb. 2025, pp. 19–24.

[4] "TRACTOR: Translating All C to Rust | DARPA." [Online]. Available: https://www.darpa.mil/research/programs/translating-all-c-to-rust

[5] N. Zhang, S. Rao, M. Franusich, and F. Franchetti, "Towards Semantics Lifting for Scientific Computing: A Case Study on FFT," *arXiv preprint arXiv:2501.09201*, 2025.

[6] F. Franchetti, T.-M. Low, T. Popovici, R. Veras, D. G. Spampinato, J. Johnson, M. Pü schel, J. C. Hoe, and J. M. F. Moura, "SPIRAL: Extreme performance portability," *Proceedings of the IEEE, special issue on "From High Level Specification to High Performance Code"*, vol. 106, no. 11, 2018.

[7] V. Zaliva, "HELIX: From Math to Verified Code," Ph.D. dissertation, Carnegie Mellon University, USA, 2020.

[8] A. Appel and A. Kellison, "VCFloat2: Floating-Point Error Analysis in Coq," in *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2024. New York, NY, USA: Association for Computing Machinery, Jan. 2024, pp. 14–29.

[9] B. Fornberg, *A Practical Guide to Pseudospectral Methods*, ser. Cambridge Monographs on Applied and Computational Mathematics. Cambridge: Cambridge University Press, 1996.

[10] W. Press, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, ser. Numerical Recipes: The Art of Scientific Computing. Cambridge University Press, 2007.

[11] V. Fuka, "PoisFFT – A free parallel fast Poisson solver," *Applied Mathematics and Computation*, vol. 267, pp. 356–364, Sep. 2015.

[12] D. Borthwick, *Spectral Theory: Basic Concepts and Applications*. Cham: Springer International Publishing, 2020.

[13] R. A. Ryan, *Introduction to Tensor Products of Banach Spaces*, ser. Springer Monographs in Mathematics. London: Springer, 2002.

[14] S. T. Brown, P. Buitrago, E. Hanna, S. Sanielevici, R. Scibek, and N. A. Nystrom, "Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research," in *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*, ser. PEARC '21. New York, NY, USA: Association for Computing Machinery, Jul. 2021, pp. 1–4.

[15] J. Rivera, "IGen," May 2024. [Online]. Available: https://github.com/joaoriverd/IGen

[16] L. Fedeli, A. Huebl, F. Boillod-Cerneux, T. Clark, K. Gott, C. Hillairet, S. Jaure, A. Leblanc, R. Lehe, A. Myers, C. Piechurski, M. Sato, N. Zaim, W. Zhang, J.-L. Vay, and H. Vincenti, "Pushing the Frontier in the Design of Laser-Based Electron Accelerators with Groundbreaking Mesh-Refined Particle-In-Cell Simulations on Exascale-Class Supercomputers," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2022, pp. 1–12.

[17] W. R. Inc., "Mathematica, Version 14.2." [Online]. Available: https://www.wolfram.com/mathematica