IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

RoboPrec: Enabling Reliable Embedded Computing for Robotics by Providing Accuracy Guarantees Across Mixed-Precision Datatypes

Alp Eren Yilmaz[1], Thomas Bourgeat[2], Lillian Pentecost[3], Brian Plancher[4,5], and Sabrina M. Neuman[1]

***Abstract*—Mobile robots demand power efficiency as well as accuracy and high performance in their computations. Embedded microcontrollers and FPGAs can consume as much as $1000\times$ less power than large CPUs and GPUs, however, these power-efficient platforms often lack full floating-point support and rely on fixed-point computations to deliver performance. This is a challenge as most robotics software uses floating-point datatypes (*double*, *float*) to conservatively ensure accuracy, and prior works that use fixed-point types employ unreliable ad hoc approaches to select the datatype precision (i.e., quantity and allocation of bits). We address this challenge with the RoboPrec framework, where we: (i) develop a transpiler that integrates code transformations and robot-specific code generation with traditional numerical stability analysis methods (which calculate error bounds), and adapts them to be practical for robotics software; and then leverage this to (ii) generate guaranteed-accuracy fixed-point code that is deployable to embedded computing platforms. We use rigid body dynamics, a fundamental robotics workload, as a motivating case study. We find that RoboPrec-generated 32-bit fixed-point code can be up to $8\times$ faster than *float* and $122\times$ faster than *double* on embedded processors while, critically, also providing guaranteed accuracy bounds with lower worst-case error than *float*.**

***Index Terms*—Embedded Systems for Robotic and Automation, Software Tools for Robot Programming, Software-Hardware Integration for Robot Systems**

I. Introduction

MODERN robots must process complex computations with high accuracy and efficiency while operating under tight power constraints. Although high-performance CPUs and GPUs support floating-point arithmetic with high speed and precision, they can consume tens to hundreds of watts of power [1], [2] which can make them undesirable for long-duration, real-world mobile robotic deployments. By contrast, embedded computing platforms such as microcontrollers and FPGAs can operate at power levels that are orders of magnitude lower [2], [3], enabling practical long-range autonomy for both large mobile robots and small resource-constrained robots with limited battery life, like nano-UAVs and insect-scale platforms [4], [5].

However, the shift to embedded computing introduces a fundamental challenge: many embedded platforms do not have floating-point units (FPUs) and do not natively support their associated high-dynamic-range datatypes (i.e., *float*, *double*). Instead, developers must rely on *fixed-point* arithmetic, which requires careful tuning of bit allocations to balance numerical range and precision. Incorrect bit configuration of fixed-point arithmetic can lead to numerical instability, degraded performance, or outright functional failure of code. This presents particular challenges for robotic applications, which often require safety-critical operation. Promising prior robotics work using fixed-point for efficient robotics computing (e.g., on FPGAs [6], [7], [8], [9], [10], [11], [12]) addresses this challenge through trial-and-error, selecting a precision that appears adequate based on a limited representative set of inputs. An analogous approach has been attempted for quantization of neural networks [13], [14], [15]. However, such ad hoc approaches offer none of the formal guarantees that are needed for real-world, safety-critical robotic deployments. While formal methods exist to determine the required fixed-point precision analytically [16], the underlying analysis is manual, requires significant expertise, and is difficult to scale to more complex algorithms.

To address these safety and performance concerns, we introduce *RoboPrec*, a framework for deploying robotics algorithms to embedded systems using fixed-point arithmetic with *provable numerical accuracy* through a three stage process. First, RoboPrec transforms programs by unrolling loops, vectors, matrix operations, and precomputing conditional statements to yield a simplified code structure. Second, the framework performs static analysis using developer-specified datatypes to provide precise error bounds over each return variable. Finally, RoboPrec generates high-performance C code that is guaranteed to run in those error bounds for that datatype.

We use rigid body dynamics, a fundamental robotics workload and key bottleneck in control applications [6], as a motivating case study to evaluate the RoboPrec framework and empirically investigate tradeoffs between numerical precision (e.g., *double* vs. *float* vs. fixed-point) and software performance on diverse computing hardware platforms. We find that for certain kernels, 32-bit fixed-point arithmetic can surprisingly offer *not only superior numerical accuracy* compared to 32-bit floating-point arithmetic, *but also improved computation speed*, even on powerful CPU hardware. Furthermore, on embedded platforms lacking hardware FPUs, optimized fixed-point representations yield significant performance gains, achieving up to $122\times$ speedup compared to optimized programs with *double* datatypes. These results underscore the importance of systematic selection of datatypes for different computing hardware constraints. We release our code at: github.com/robomorphic/roboprec

Manuscript received: July 1, 2025; Revised October 1, 2025; Accepted November 12, 2025.

This paper was recommended for publication by Editor Soon-Jo Chung upon evaluation of the Associate Editor and Reviewers' comments.

[1]AY, SN are with Boston University, USA {yilmaz, sneuman}@bu.edu

[2]TB is with EPFL, Switzerland thomas.bourgeat@epfl.ch

[3]LP is with Amherst College, USA lpentecost@amherst.edu

[4]BP is with Dartmouth College, USA plancher@dartmouth.edu

[5]BP is also with Barnard College, Columbia University, USA

©2026 IEEE

II. Background and Related Work

A. Fixed-Point Arithmetic

Fixed-point arithmetic represents real numbers by storing values as scaled integers with an implicit decimal point, allocating bits to the integer and fractional parts. We refer to keeping the position of the decimal point and the total number of bits constant across all variables as *uniform-precision*. If either the decimal point location or the total number of bits differs between variables, we call it *mixed-precision*. Addition and subtraction between fixed-point numbers with the decimal point in the same location reduces to simple integer operations, which are faster and more energy-efficient than their floating-point counterparts [17]. When the location of the implicit decimal point differs, a shift operation is required to align the numbers before integer math can be performed. Multiplication and division introduce complexity: multiplying two n-bit fixed-point values produces a $2n$-bit result, and division similarly requires casting the dividend to a larger bit width to preserve precision.

Compared to floating-point operations, integer operations are significantly more efficient in power, speed, and circuit area. For example, 32-bit floating-point addition on CPUs can consume up to $9\times$ more power than integer addition, and multiplication up to 19% more power [17].

B. Challenges in Using Numerical Verification for Robotics

Prior work has sought to enable generation of provably reliable software by developing static analysis tools (e.g., PRECiSA [18], Fluctuat [19], Daisy [20], Satire [21], Real2Float [22], FPTaylor [23], Gappa [24]) to analyze numerical programs and produce provable bounds. For example, Daisy [20], which we selected as a back-end in our RoboPrec framework, contains methods for robust finite-precision analysis of both floating-point and fixed-point numbers, and has a limited ability to generate C code for different datatypes.

However, leveraging traditional verification tools for robotics algorithms faces a number of challenges. First, the tooling for such verification typically requires extensive code re-implementation in highly constrained Domain-Specific Languages (DSLs) [18]. These DSLs frequently lack robust support for common programming constructs like loops and conditionals [18], offer limited handling of vectors and matrices [25], and prevent the use of external libraries for tasks like parsing URDF files [25]. Furthermore, these DSLs may not be executable [20], which makes it harder to debug the code before launching time-consuming analysis runs.

Furthermore, numerical analysis techniques like interval and affine arithmetic often over-approximate, producing conservative error bounds due to their inability to capture nonlinear variable relationships [20]. This can trigger false alarms and lead to inefficient code, e.g., unnecessary bit-width increases or spurious division-by-zero hazards. Furthermore, unnecessary bit-width increases may affect performance, undermining optimization efforts for embedded systems [26].

Finally, some analysis tools [20] lean exclusively on aggressive mixed-precision datatypes and lack native support for the analysis of uniform-precision datatypes that are more commonly deployed in real-world fixed-point code for embedded systems. While mixed-precision can demonstrate what is theoretically optimal in terms of tailoring the numerical range on a per-variable basis, it is desirable to also explore the possibilities of well-provisioned uniform-precision arithmetic, since it significantly reduces the need for additional shift and casting operations in the final code.

III. The RoboPrec Framework

In this work, we address the challenges in Sec. II with a framework that harnesses the power of traditional static analysis tools by expanding their usability and scope to make them practical for complex robotics algorithms. The RoboPrec framework is designed to bridge the gap between modern robotics software and low-level static analysis tools that perform numerical precision verification. RoboPrec enables users to develop robotics code in modern programming languages, without the need for wholesale re-implementation of code in restrictive DSLs. It also expands the error analysis and code generation processes to avoid the inefficiencies, hazards, and impracticalities described in Sec. II.

The RoboPrec framework (Fig. 1) has three stages: (1) *Transpiler:* Decouples the user-facing language interface from code transformations into an analysis-specific DSL (Sec. III-A); (2) *Error Analysis:* Performs an expanded analysis of numerical range and error propagation; (Sec. III-B); and (3) *Code Generation:* Generates optimized code for a chosen datatype, deployable across diverse computing platforms, including microcontrollers and FPGAs (Sec. III-C). This modular design allows RoboPrec to provide robotics-specific usability features at the front-end while interfacing with a powerful analysis back-end. Note that while the initial version of RoboPrec utilizes the Daisy [20] analysis tool, the same framework can be ported to any number of similar tools [18], [19], [21], [22], [23], [24].

A. Transpiler: From User-Friendly Robotics Code to DSL

The pipeline begins with the Transpiler stage (Fig. 1), which takes as an input a robotics program written in a modern programming language (Rust) with a user-friendly linear algebra library. Then, as an output it produces code that has been transpiled, including static optimizations and robot-specific code specialization, into the analysis language (e.g., a tool-specific DSL) for downstream analysis.

Enabling Interoperability with Robotics Libraries: To apply analysis tools to standard robotics software, developers need to be able to leverage familiar programming paradigms and integrate external libraries (e.g., URDF parsers), which is often infeasible within existing DSL environments. To address this, we make the crucial design choice to build RoboPrec's frontend in the Rust programming language, as a linear algebra library, making use of intuitive Scalar, Vector, and Matrix *structs* whose role is primarily to track the sequence of operations performed on them (see example with rotation and translation matrices in Fig. 2). Rust is a high-performance alternative to C/C++ that is growing in popularity in the software community (including robotics [27]) because of its reliability

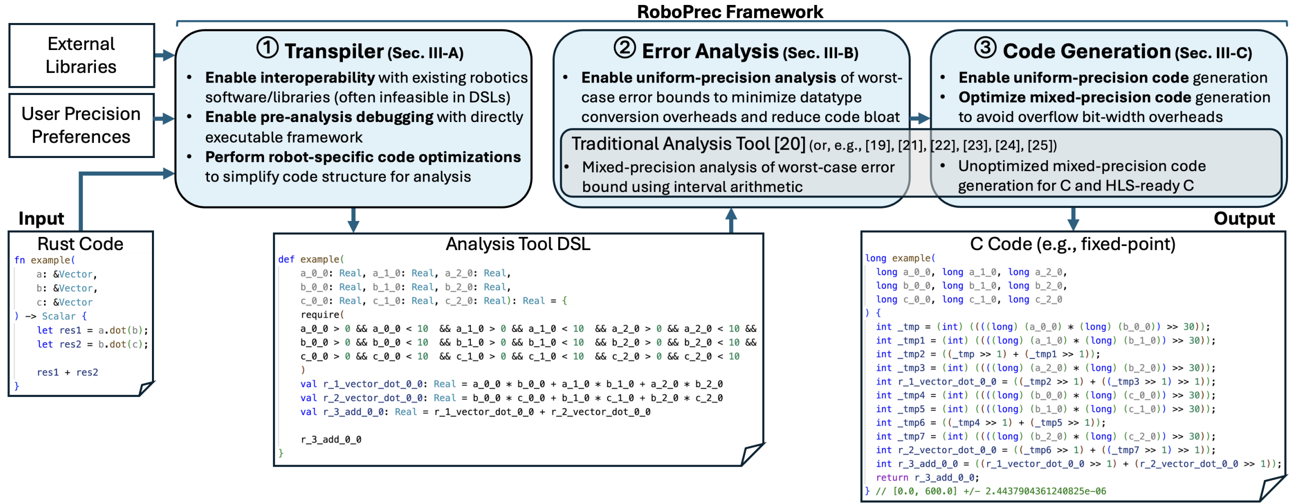


Fig. 1: Overview of the RoboPrec framework for generating bounded-accuracy robotics code for embedded computing. (1) *Transpiler:* Transforms user-provided Rust code into analysis DSL, using robot-specific optimizations. (2) *Error Analysis:* Utilizes a static analysis tool back-end (e.g., [20]) for mixed-precision analysis, improved in RoboPrec to include efficient uniform-precision analysis. (3) *Code Generation:* Generates C code or HLS-ready code for FPGAs, optimized by RoboPrec.

benefits due to strong typing and safe memory usage. Enabling users to input code in a modern fully-featured programming language represents a significant difference from the restrictive DSL-centric approaches common in prior verification tools (e.g., highly-constrained functional languages [18]), and can facilitate integration of RoboPrec with existing robotics software and toolchains (e.g., ROS [28]). This is a critical step in promoting practical adoption and usability in the robotics software community. We also note that RoboPrec's output is C-code that is easy to integrate into existing C/C++ robotics codebases.

Enabling Executability and Debugging: RoboPrec Rust code is also directly executable, enabling standard debugging and testing workflows to be applied to the resulting code *before* the time-consuming verification process, streamlining development cycles and avoiding the complex post-analysis debugging often required with analysis tool DSLs. RoboPrec's use of a modern programming language enhances usability by allowing the same code to be both ready for analysis and reusable as a standard linear algebra library. Furthermore, the transpiler provides informative compile-time error checking in Rust, providing clear diagnostics.

Performing Robot-Specific Code Optimizations: For robotics applications where the robot's structure is known (e.g., via a URDF), the transpiler performs robot-specific code specialization by unrolling loops and precomputing conditional branches based on this structure, building on prior software work, e.g., for rigid body dynamics [7], [29]. This yields an optimized, straight-line intermediate representation, well-suited for the subsequent analysis phase (and beneficial downstream for producing code suitable for FPGA deployment in particular). These unrolling optimizations currently require conditionals to be statically determined at compile time, but we plan to extend RoboPrec to cover dynamic program analysis in future work.

```
fn apply_rotation_translation(
    rotation: &Matrix, // rotation matrix
    translation: &Vector, // translation vector
    p: &Vector, // position vector
) -> Vector {
    let rotated_p = rotation.matmul_vec(p);
    let translated_p = rotated_p + translation;

    translated_p
}
```

Fig. 2: Example using RoboPrec's Rust library for rotation and translation matrices. Users develop in a debuggable high-performance language, and RoboPrec automatically transpiles code to an optimized DSL for back-end analysis.

B. Error Analysis: Broadening Functionality of Verification

RoboPrec performs error analysis (Fig. 1) by utilizing a verification tool back-end that we augment with additional functionality for easier deployment on embedded platforms by enabling the use of *uniform-precision* fixed-point (Sec. II).

Selection of Verification Analysis Back-End: For the initial version of RoboPrec, we selected the Daisy verification tool [20] as our back-end because it provides an extensible analysis pipeline and support for multiple analysis methods. For robotics workloads, we found that the *interval arithmetic* analysis method provides a suitable balance between the tightness of error bounds and computational efficiency compared to more complex, time-consuming methods (e.g., affine arithmetic), that we found to not provide meaningful improvements in error bounds for robotics workloads. Based on developer-specified input ranges, this stage analyzes the roundoff errors for a target numerical precision, and computes the worst-case variable error bounds. Again, we note that RoboPrec's additions are extensible to other similar tools (see Sec. II) due to its modular framework.

Enabling Uniform-Precision Verification Analysis: RoboPrec adds support to perform the error analysis described above for uniform-precision datatypes (Sec. II), in addition to default mixed-precision datatypes. Mixed-precision allocates the minimum precision required for each variable

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

independently to prevent overflows. Uniform precision, on the other hand, enforces a consistent bit-width across all variables. While we observe in our evaluation (Sec. IV) that mixed-precision enables more flexibility and ultimately tighter bounds, it inherently requires more operations to shift between the various precisions. This makes mixed-precision programs slower than uniform-precision ones, highlighting a key tradeoff to consider in robotics workloads aiming for maximum performance. We note that this reduction in conversion overhead may be particularly relevant for resource-constrained embedded targets, making it important functionality to include in the RoboPrec framework.

C. Code Generation

Finally, code generation (Fig. 1) produces deployable code. RoboPrec outputs optimized C code with a streamlined structure due to the transpiler's processing of loops and conditionals (Sec. III-A), which facilitates compiler optimizations. For developers targeting FPGAs, output in *ap_fixed* format is provided, suitable for use in high-level synthesis (HLS), i.e., compiling high-level code into Verilog. The fixed-point C code generation includes implementation of the bit shifts and type casts required to correctly emulate fixed-point arithmetic using standard integer types.

Optimizing Mixed-Precision Code: Generating efficient code, especially for mixed-precision representations where the binary point shifts between operations, requires careful management. For example, multiplication of two n-bit fixed-point numbers may require intermediate calculations using $2n$-bit integers to prevent overflow, adding computational cost (division can lead to similar widening). Our implementation includes refinements over the baseline analysis tool aimed at optimizing this process over the standard approaches, which can be inefficient. For the multiplication issue described above, for example, while prior tools [20] will default to inefficiently widening all variables to $2n$-bits, in RoboPrec we widen variables to $2n$-bits only when necessary in the arithmetic, and then return to n-bits. For an FPGA design with custom datapaths, this can prevent a nearly $2\times$ unnecessary penalty in area and power.

Enabling Uniform-Precision Code: RoboPrec enables generation of uniform-precision code, in addition to the default mixed-precision in prior works [20]. This is important for reducing code bloat: uniform precision avoids shift operations related to moving decimal points. This reduction in operations can lead to higher-performance code at the potential cost of reduced numerical accuracy. We compare performance and accuracy of both uniform and mixed-precision code in our evaluation (Sec. IV).

IV. Evaluation: Rigid Body Dynamics Case Study

We evaluate the performance and accuracy benefits of RoboPrec-generated code using standard rigid body dynamics algorithms [30] as a case study, as such computations are common bottlenecks in many robotics applications [6]. Across three benchmark algorithms and four robots (Sec. IV-A), we find that RoboPrec-generated 32-bit fixed-point code can not only be up to $8\times$ faster than *float* and $122\times$ faster than *double* on embedded processors (Fig. 3a, Sec. IV-B), but it also provides guaranteed accuracy bounds that surpass *float* (Fig. 4, Sec. IV-D). We also demonstrate using RoboPrec to successfully execute a pick-and-place task on a real manipulator robot using a low-power embedded processor and reduced-precision fixed-point kinematics calculations (Fig. 6, Sec. IV-G). These results indicate that tools like RoboPrec are a promising pathway to delivering *guaranteed accuracy as well as high performance* for robotics on diverse computing platforms, especially low-power embedded devices.

A. Methodology

We evaluate RoboPrec across three common rigid body dynamics algorithms [30]: forward kinematics (*FK*), Recursive Newton-Euler Algorithm (*RNEA*), and first-order derivatives of RNEA (*RNEA Deriv*). Each of the algorithms take as inputs for each joint the acceleration, velocity, and the sine and cosine of position. We analyze these algorithms applied to four robots with increasing complexity of degrees-of-freedom (dof): 4-dof RoArm-M2 and 5-dof RoArm-M3 [31], 6-dof Indy7 [32], and 7-dof Franka Emika Panda [33].

We evaluate RoboPrec across a diverse set of real computer processors to analyze the impact of different datatypes executed on a range of architectures with varied computing capabilities, including level of support for vectorization and for hardware floating point units. Our platforms range from the low-power embedded-scale Raspberry Pi Pico RP2350 and RP2040 (under 1 Watt), to the Raspberry Pi 5 BCM2712 (12 W), to the high-performance, but power-inefficient, Intel i7-14700K (253 W). For all platforms, we compiled our code with `g++` for cross-platform portability, using `-O3` optimizations for high performance.[1] The i7 and BCM2712 CPU governors are set to performance mode and we disable hyperthreading and turboboost on the i7 [2] to avoid variance and nondeterminism in timing measurements from these mechanisms (e.g., OS thread scheduling interrupts, thermal and workload transients).[3] For runtime results, we run each algorithm for 100k iterations and take the mean. The embedded platforms have no operating system (OS), but for non-embedded platforms we additionally average the means of 10 runs of 100k iterations to insulate from the effects of OS interrupts. Standard deviation across all runs was low, under 3%. As a baseline, we compare the performance of RoboPrec's output on the i7 to code generation outputs of the state-of-the-art Pinocchio [29] library. To validate our code, we verify

[1] We note that running experiments under `-O2` decreased performance across all results with no observed impact on code stability, so we present `-O3` results here. This is also in keeping with the use of `-Ofast` or `-O3` in other state-of-the-art robotics computing work (e.g., Pinocchio dynamics codegen [29], high-performance collision checking [34]). We do not use `-Ofast` or `fast-math`, to avoid undefined behavior and code instability.

[2] This methodology is common in computer systems research, and is in keeping with robotics computing work reporting results for a fixed clock frequency [29], [35], or taking added steps to reduce nondeterminism [36].

[3] Additional testing with these features enabled does not change conclusions in Sec. IV: All i7 runtimes decrease, but relative improvements from RoboPrec remain similar or increase (e.g., 1.98× to 2.38× speedup in one case). However, run-to-run variance rises substantially (standard deviation of results increases up to 18%), so we disable these features for Sec. IV.

that both RoboPrec and Pinocchio produce identical numerical outputs using the *double* datatype. Later, accuracy of the final RoboPrec-generated code (in any given numerical precision) is reported to the user by RoboPrec.

B. Datatype Performance on Embedded Low-Power CPUs

Performance Results: Using RoboPrec (Sec. III), we investigate performance tradeoffs using different datatypes on embedded low-power (mW) processors, and find that *careful selection of datatype even within embedded CPUs is essential for fast runtimes, achieving speedups from* $8.81\times$ *to* $122.92\times$ *over the standard* double *type* (see Fig. 3a). Note that in Sec. IV-D, we analyze the accompanying guaranteed accuracy tradeoffs for all datatypes to complete the picture.

Processor Architecture Matters for Power and Speed: The ARM cores within the RP2350 microcontroller have half-precision floating-point units (FPU) that only support *float*, while the RP2040 cores and the RP2350's RISC-V cores have no FPU at all. This has two consequences: First, the RP2350 running with FPU-enabled ARM cores has a peak power around $3.16\times$ larger than the no-FPU RP2040, which can make a significant difference to the battery life of a resource-constrained robot. Second, the lowest-runtime datatype selected for the RP2350-ARM and the RP2040 are of course different (*float* and analytically-tuned fixed-point, respectively), and both offer substantial speedups (around an order of magnitude) over the standard *double* type. This is because when the processor does not natively support *double* types, it instead relies on slow and costly software-based emulation, negatively impacting overall performance.

C. Datatype Performance on Non-Embedded CPUs

Performance Results: We compare datatypes across large (10s to 100s of Watts) non-embedded CPUs (Fig. 3b) and find that *counterintuitively, fixed-point datatypes on these platforms can sometimes achieve similar or faster (e.g.,* $1.01\times$ *to* $1.42\times$ *speedups) performance than the fully-supported* double *type.* In particular, RoboPrec's uniform-precision *fixed32* (Sec. III-B, Sec. III-C) improves performance on the i7 over *double*. This indicates that deliberate choice of datatype can be beneficial beyond embedded-scale robotics computing– these processors are comparable to those onboard larger robots with a significant onboard power budget, e.g., the Boston Dynamics Spot quadruped [37].

Validation of RoboPrec Code Generation: To validate RoboPrec's code generation, we compare its *double* outputs to the state-of-the-art Pinocchio dynamics library [29] and its code generation (Sec. IV-A). In Fig. 3b we also compare runtime to Pinocchio as a performance baseline. RoboPrec code manages similar ($1.01\times$) or faster (up to $1.98\times$) performance to Pinocchio on the Intel i7. We manage speedups in some cases, likely because Pinocchio's code generation reportedly can have difficulties with vectorization [38], while RoboPrec's transpiler optimizations (Sec. III-A) create a simplified code structure that is amenable to vector operations.

D. Analytical Accuracy Guarantees Across Datatypes

Guaranteed Error Bound Results: RoboPrec enables the analytical calculation of guaranteed error bounds across datatypes for our collection of real robotics algorithms (Fig. 4). While *double* gives the tightest error bounds because it has 64 bits to express numbers with, *the counterintuitive result is that, despite common assumptions,* fixed32 *types can give better numerical accuracy than* float *types, even though both use the same number of bits (32) in their representation.*

By using bits to represent the exponent, *float* can represent a larger range of values, $[-3.4\text{e}38, 3.4\text{e}38]$, compared to *fixed32*, which can only represent $[-2^{31}, 2^{31}-1]$, but this comes at the cost of precision for *float*. Therefore, if the dynamic range of numbers in an algorithm does not require *float*'s wider range, an appropriately-tuned *fixed32* can represent more fine-grained fractional values.

Challenges in Guarantees for Complex Algorithms: The most complex algorithm, RNEA-Deriv, poses challenges for providing good analytical error bounds because the greater number of cascaded operations than FK and RNEA leads to greater accumulation of numerical error. This challenge is compounded by robot complexity (dof) which also increases the number of sequential operations in these algorithms. For example, the worst scenario in Fig. 4 is RNEA-Deriv for 7-dof, where variables have a worst-case range spanning 10^{10}.

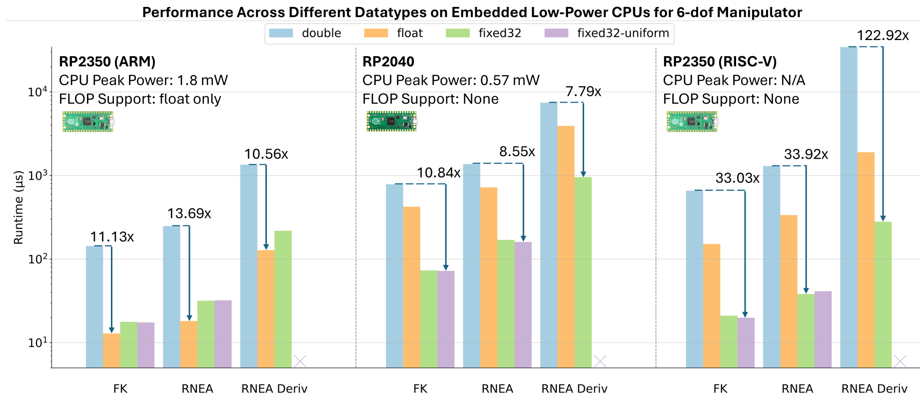
Practical Impact: These bounds represent low-level numerical correctness under worst-case analysis, which *does not* necessarily suggest that robot states or errors will approach, e.g., 10^{10}, in physical units as the bounds are unitless. Instead, the practical effect of these bounds is at the lowest level of computational processing: they apply to intermediate arithmetic in the fixed-point pipeline, and determine the minimum integer word length needed to provably avoid overflow. While current bounds are intentionally conservative, they do provide deterministic numerics and thus support safe operation under quantization, a prerequisite for pursuing higher-level end-to-end safety guarantees for robotics computing.

Co-Design of Datatype and Algorithm: Given the spread in guaranteed error bounds, it is critical that robotics practitioners use analysis tools like RoboPrec to systematically tune the datatype that best fits the application-level error tolerance of their algorithm. If an application, e.g., feedback control, already has some degree of built-in error tolerance, then datatypes such as a well-tuned *fixed32* may be an option to reliably migrate code to a low-power embedded device.

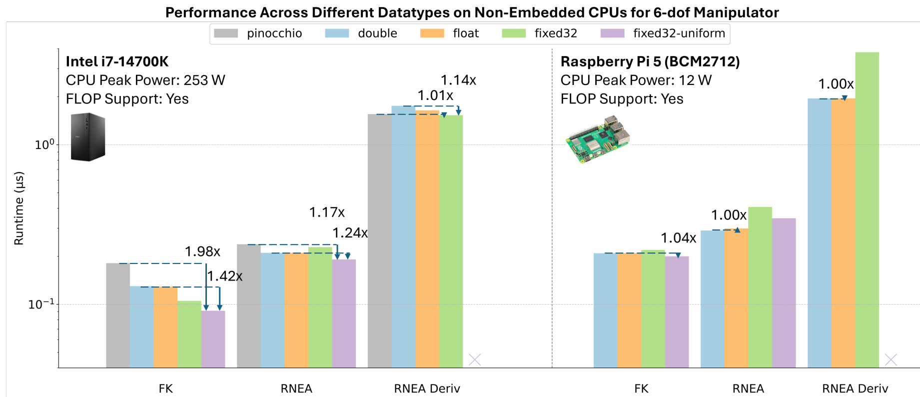
E. Empirical Measurements Vs. Analytical Accuracy Bounds

The analytically-derived accuracy bounds from RoboPrec (Fig. 4) represent guarantees on the worst-case errors of each algorithm (Sec. III-B). However, because these are conservative bounds (i.e., over-approximations of error), we also perform experiments to study experimentally-measured error values and compare to the trends in worst-case analytical errors. For measurements, we sample 100k points from the input space of each function, and use a *double* implementation as ground truth to calculate output error.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.



(a) Runtimes across embedded CPUs: two with no *float* or *double* (FLOP) support and one with *float* support only (RP2350-ARM). On the RP2040, processor architecture simplifications (e.g., not providing FLOP support) enable $3.16\times$ lower peak power over the RP2350-ARM, and RoboPrec's *fixed32* code demonstrates speedups of $7.79\times$ to $10.84\times$ over *double*. The RP2350-RISC-V exhibits greater speedups for *fixed32* types, from $33\times$ to $122\times$ over *double*. Note: Analytical error bounds grow with algorithm complexity (see Fig. 4) so, e.g., RoboPrec often cannot produce a provably safe program for RNEA-Deriv using *fixed32-uniform*.



(b) Runtimes on non-embedded CPUs with peak power of 10s to 100s of Watts. Counterintuitively, RoboPrec *fixed32* achieves similar or better performance than RoboPrec *double* on these FLOP-supported devices. We validate against the state-of-the-art Pinocchio library [29] (not supported on embedded systems), and RoboPrec code can achieve comparable ($1.01\times$) or faster performance ($1.17\times$ to $1.98\times$).

Fig. 3: Performance of different datatypes across embedded (mW of power) and non-embedded (10s to 100s of Watts) CPUs.

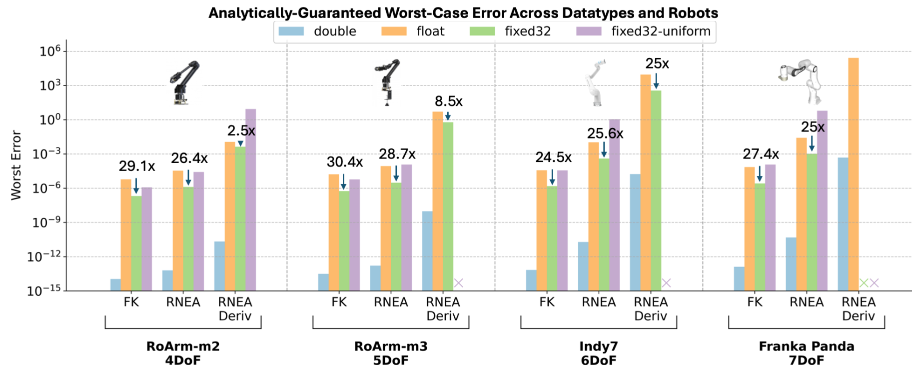
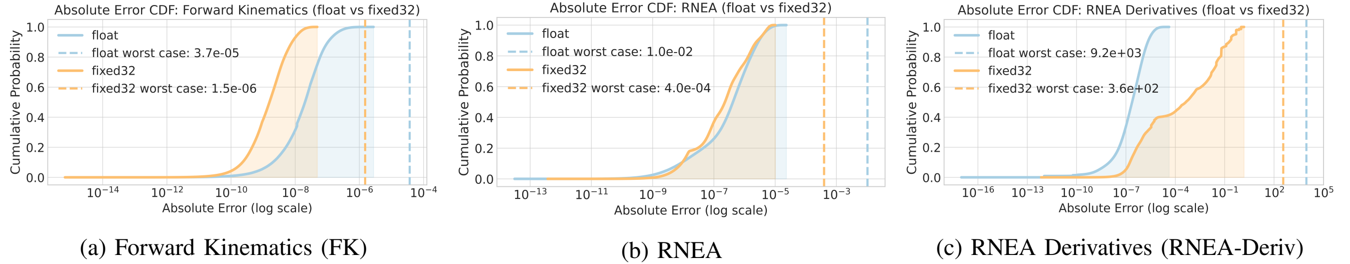


Fig. 4: Analytical error bounds computed by RoboPrec increase with robot and algorithm complexity, but they also vary across datatypes, exposing a performance (Fig. 3) versus accuracy tradeoff space for robotics practitioners to leverage. Surprisingly, despite conventional use of *float* for high accuracy, we find that *fixed32* consistently achieves a lower worst-case guaranteed error bound, giving improvements in numerical accuracy ranging from $2.5\times$ to $30.4\times$ over *float*. For the most complex algorithm, RNEA-Deriv, RoboPrec reveals that worst-case error causes overflow for *fixed32* in many cases.

We find that the predictive quality of analytical worst-case error scales deteriorates with increased complexity of the algorithm (Fig. 5), which is as expected due to the accumulation of estimated error over longer chains of cascaded operations for complex algorithms (see scaling trends in Sec. IV-D and Fig. 4). For example, in our simplest algorithm, FK (Fig. 5a), the predictive quality of the analytical worst-case bound is strongest: While absolute values of the errors for *float* and *fixed32* are both conservative by about 10^3, the relative error between the two types is well-modeled

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.



(a) Forward Kinematics (FK) (b) RNEA (c) RNEA Derivatives (RNEA-Deriv)

Fig. 5: Empirical cumulative density functions (CDF) of average-case errors for 6-dof robot, randomly sampling input space. Dashed lines are RoboPrec's analytical worst-case error. Worst-case bounds capture empirical trends for FK, but do not scale well to more complex algorithms. Tightening error bounds during analysis (Sec. V) can enable less conservative guarantees.

by the worst-case errors, i.e., *fixed32* is estimated as giving a $24.7\times$ improvement in error over *float*, and measured as giving a $17.1\times$ improvement in median error. However, for the more complex RNEA (Fig. 5b) and RNEA-Deriv (Fig. 5c) algorithms, the measured trends differ significantly from the estimates. In Sec. V we discuss pathways to tighten worst-case error bounds for more complex robotics algorithms, to enable less conservative guarantees in future systems.

F. Floating-to-Fixed-Point Conversion and Program Design

Users must carefully consider how much of a program to execute in fixed-point arithmetic versus floating-point arithmetic, since conversion between the two families of datatypes introduces latency overheads. On embedded platforms such as the RP2350-ARM and RP2040, we find that if users employ a naive "convert-one-single-function-only" strategy (i.e., for a single function: convert every input from *double* to *fixed32*, execute the function in *fixed32*, then convert every output from *fixed32* to *double*), then conversion between *double* and *fixed32* can take a large percentage of function runtime (Fig. 3a) across our three case study dynamics functions, with averages of 197.5%, 27%, 10.15% for FK, RNEA, RNEA-Deriv respectively. (Note that RP2350-RISC-V gave worse conversion rates, i.e., 363%, 97%, 25%, respectively, likely due to inefficiencies in the RISC-V compiler back-end compared to the more mature compiler flows for the ARM architectures of the other embedded CPUs.) This indicates that on embedded devices, *there are significant performance advantages to converting an entire program into fixed-point arithmetic, rather than only one subroutine in isolation, in order to amortize conversion latency.* That being said, in Sec. IV-G we demonstrate that even using a naive single-function strategy with conversion costs can deliver meaningful speedups for a real robot experiment.

G. Real Robot Demonstration Using Embedded Fixed-Point

We demonstrate the use of RoboPrec to successfully execute an inverse kinematics pick-and-place demo on a low-cost RoArm-M3 manipulator using a Raspberry Pi Pico RP2040 embedded processor. Analyzing the full inverse kinematics pipeline presents a challenge due to its inherently variable number of iterations, which complicates end-to-end static range and precision verification. However, inverse kinematics calls forward kinematics in every iteration, so we made FK our target algorithm and we separately optimized and verified that function. Despite only targeting a single function (FK) taking 8.27% of the original overall application runtime, and paying full floating-to-fixed-point conversion costs (Sec. IV-F), we were able to *successfully execute our desired task using a low-power embedded processor for the calculations* (Fig. 6) and we *reduced total execution time by* 5% *(*$42\ ms$ *to* $40\ ms$*) by moving FK to fixed precision.* Note that the runtime of FK itself reduced 46.3% ($218\ us$ to $117\ us$) using RoboPrec's uniform-precision *fixed32*.

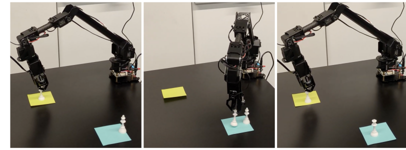


Fig. 6: RoArm-M3 pick-and-place demonstration. Inverse kinematics computed on embedded RP2040 with RoboPrec-generated *fixed32* FK code. Positions sent via serial interface.

We compared two configurations for the internal forward kinematics subroutine, a baseline using *double* precision, and an optimized version using *fixed32* with 5-bits for integer and 27-bits for decimal, as determined by analysis using RoboPrec: We ran forward kinematics through RoboPrec, and used its code generation to develop C code that we deployed to our robot. Again, RoboPrec's output code executed the task successfully and with speedups, despite the naive implementation (Sec. IV-F). Finally, we note that further performance gains are also achievable by converting the entire inverse kinematics algorithm to reduced precision.

V. Conclusion and Future Work

In this work, we present RoboPrec, an open-source framework to streamline formal verification for robotics programs, and perform robot-specific code generation across multiple numerical precisions. Our framework reduces developer effort and provides rigorous error and range guarantees for both embedded and non-embedded processors. We show that fixed-point arithmetic (e.g., *fixed32*) can be both faster and more accurate than *float*, in both power-constrained devices like microcontrollers and FPGAs, and modern power-hungry processors like the Intel i7-14700K. We also show that our robot-specific code generation may outperform traditional approaches by efficiently using data types and compilers.

We provide RoboPrec as an open-source tool to the research community so that it can be a platform for future work. For example, there are opportunities for future efforts to address scalability challenges to apply analysis to larger workloads (Sec. IV-D) with less conservative estimates (Sec. IV-E), and to further increase performance through optimally balancing type conversion overheads (Sec. IV-F).

Scaling to larger workloads with more practical bounds is essential to expanding results across the full robotics pipeline (including computationally expensive perception systems), and working towards end-to-end safety certification of robot systems, e.g., to enable safe interaction of humans and robots. A promising approach would be to leverage domain-specific knowledge to restrict specific variable ranges based on physical constraints, e.g., robot structures like joint limits, or through application specific analysis in order to significantly tighten worst-case range estimations. In pursuit of safety, there are also opportunities to augment the framework with interpretability, e.g., principal component analysis-style passes to improve understanding of how overflow in certain variables can impact physical robot failure behavior.

To optimize performance in the face of cross-datatype conversion overheads, one promising direction can be using the error bounds from RoboPrec to automatically perform design space exploration to find optimal operating points in the tradeoff space, e.g., a balance of mixed- and uniform-precision computations. The performance of lower-precision datatypes can also be increased through support of compiler primitives for packed-vectorization of computations.

These enhancements are expected to improve both the accuracy and speed of complex kernels, paving the way for broader application of formal verification methods in real-world robotics, and enabling deployment of accurate, reliable robotics software on embedded computing devices.

References

[1] B. Betkaoui, D. B. Thomas, and W. Luk, “Comparing performance and energy efficiency of fpgas and gpus for high productivity computing,” in *IEEE Int. Conf. on Field-Programmable Technology*, 2010.

[2] L. Scarciglia, A. Paolillo, and D. Palossi, “A map-free deep learning-based framework for gate-to-gate monocular visual navigation aboard miniaturized aerial vehicles,” in *(ICRA)*, 2025.

[3] Z. Wan, A. Lele, B. Yu, S. Liu, Y. Wang, V. J. Reddi, C. Hao, and A. Raychowdhury, “Robotic computing on fpgas: Current progress, research challenges, and opportunities,” in *IEEE (AICAS)*, 2022.

[4] R. Wood, R. Nagpal, and G.-Y. Wei, “Flight of the robobees,” *Scientific American*, vol. 308, no. 3, 2013.

[5] S. M. Neuman, B. Plancher, B. P. Duisterhof, S. Krishnan, C. Banbury, M. Mazumder, S. Prakash, J. Jabbour, A. Faust, G. C. de Croon, *et al.*, “Tiny robot learning: Challenges and directions for machine learning in resource-constrained robots,” in *IEEE (AICAS)*, 2022.

[6] S. M. Neuman, B. Plancher, T. Bourgeat, T. Tambe, S. Devadas, and V. J. Reddi, “Robomorphic computing: A design methodology for domain-specific accelerators parameterized by robot morphology,” in *ACM Int. Conf. Arch. Support Prog. Lang. Op. Sys. (ASPLOS)*, 2021.

[7] B. Plancher, S. M. Neuman, T. Bourgeat, S. Kuindersma, S. Devadas, and V. Janapa Reddi, “Accelerating robot dynamics gradients on a cpu, gpu, and fpga,” *IEEE Robotics and Automation Letters (RA-L)*, 2021.

[8] F. Xu, Z. Guo, H. Chen, D. Ji, and T. Qu, “A custom parallel hardware architecture of nonlinear model-predictive control on fpga,” *IEEE Transactions on Industrial Electronics*, 2021.

[9] B. Khusainov, E. C. Kerrigan, and G. A. Constantinides, “Automatic software and computing hardware codesign for predictive control,” *IEEE Transactions on Control Systems Technology*, 2018.

[10] A. Ravera, A. Oliveri, M. Lodi, A. Bemporad, W. Heemels, E. C. Kerrigan, and M. Storace, “Co-design of a controller and its digital implementation: The moby-dic2 toolbox for embedded model predictive control,” *IEEE Transactions on Control Systems Technology*, 2023.

[11] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, “The microarchitecture of a real-time robot motion planning accelerator,” in *IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, 2016.

[12] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris, “Robot motion planning on a chip.” in *Robotics: Science and Systems*, 2016.

[13] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.

[14] E. Frantar, S. Ashkboos, T. Hoefler, and D.-A. Alistarh, “Optq: Accurate post-training quantization for generative pre-trained transformers,” in *Int. Conference on Learning Representations (ICLR)*, 2023.

[15] L. Grossman and B. Plancher, “Just round: Quantized observation spaces enable memory efficient learning of dynamic locomotion,” in *IEEE International Conf. on Robotics and Automation (ICRA)*, 2023.

[16] J. Jerez, P. Goulart, S. Richter, G. Constantinides, E. Kerrigan, and M. Morari, “Embedded online optimization for model predictive control at megahertz rates,” *IEEE Trans. Automat. Contr.*, 2014.

[17] M. Horowitz, “Computing’s energy problem (and what we can do about it),” in *IEEE Int. Solid-State Circuits Conference (ISSCC)*, 2014.

[18] L. Titolo, M. Moscato, M. A. Feliu, P. Masci, and C. A. Muñoz, “Rigorous floating-point round-off error analysis in precisa 4.0,” in *International Symposium on Formal Methods*. Springer, 2024.

[19] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine, “Towards an industrial use of fluctuat on safety-critical avionics software,” in *International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 2009, pp. 53–69.

[20] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, “Daisy-framework for analysis and optimization of numerical programs (tool paper),” in *Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2018.

[21] A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, and P. Panchekha, “Scalable yet rigorous floating-point error analysis,” in *(SC20)*. IEEE, 2020, pp. 1–14.

[22] V. Magron, G. Constantinides, and A. Donaldson, “Certified roundoff error bounds using semidefinite programming,” *ACM (TOMS)*, 2017.

[23] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, “Rigorous estimation of floating-point round-off errors with symbolic taylor expansions,” *ACM (TOPLAS)*, 2018.

[24] M. Daumas and G. Melquiond, “Certification of bounds on expressions involving rounded operators,” *ACM Trans. Math. Software*, 2010.

[25] A. Isychev and E. Darulova, “Scaling up roundoff analysis of functional data structure programs,” in *International Static Analysis Symposium*. Springer, 2023, pp. 371–402.

[26] E. Darulova, E. Horn, and S. Sharma, “Sound mixed-precision optimization with rewriting,” in *ACM/IEEE (ICCPS)*. IEEE, 2018.

[27] J. Haviland and P. Corke, “Robotics software: Past, present, and future,” *Annual Review of Control, Robotics, and Auton. Sys.*, 2024.

[28] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[29] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard, “The pinocchio c++ library – a fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives,” in *IEEE Int. Symp. Sys. Integrations (SII)*, 2019.

[30] R. Featherstone, *Rigid body dynamics algorithms*. Springer, 2014.

[31] Waveshare, “Roarm-m2 and m3,” Accessed 2025, waveshare.com.

[32] Neuromeka, “Indy7,” Accessed 2025, en.neuromeka.com/cobot-1-1.

[33] Franka, “Emika panda,” Accessed 2025, franka.de.

[34] C. Ramsey, Z. Kingston, W. Thomason, and L. E. Kavraki, “Collision-Affording Point Trees: SIMD-Amenable Nearest Neighbors for Fast Motion Planning with Pointclouds,” in *(RSS)*, 2024.

[35] W. Thomason, Z. Kingston, and L. E. Kavraki, “Motions in microseconds via vectorized sampling-based planning,” in *(ICRA)*, 2024.

[36] S. M. Neuman, T. Koolen, J. Drean, J. E. Miller, and S. Devadas, “Benchmarking and workload analysis of robot dynamics algorithms,” in *(IROS)*, 2019.

[37] Boston Dynamics, “Spot,” Accessed 2025, bostondynamics.com/spot.

[38] J. Carpentier, R. Budhiraja, and N. Mansard, “Proximal and sparse resolution of constrained dynamic equations,” in *Robotics: Science and Systems 2021*, 2021.