A Vectorial and Phasor-Based Model of Programming: From Abstract Semantics to Quantum and Electrical Implementations

Lucio Guerchi
lugu16f@gmail.com

September 9, 2025

Abstract

We present a representation of programming semantics in which assignments, operations, data types, and control structures are expressed as vectors and phasors within defined vector spaces. Assignments and arithmetic are modeled as affine/linear transformations; loops and decisions are represented by phasors, where iteration corresponds to angular displacement. We show how this formalism naturally translates into matrix algebra, and demonstrate mappings to quantum-circuit and electrical-circuit implementations. This unifies abstract programming constructs, linear-algebraic reasoning, and hardware realization in a single geometric framework.

Keywords: vector semantics, phasors, programming model, quantum circuits, electrical circuits, matrix algebra

1 Introduction

We propose a geometric programming model VPL (Vector-Phasor language) in which program state, data, operators and control flow are expressed as vectors or rotations (phasors) in a partitioned vector space. This view provides a compact linear-algebraic substrate for reasoning about programs and a pathway to physical implementations in quantum and electrical hardware.

2 Formal model

Let the program state be represented in homogeneous form as a column vector:

$$S = \begin{bmatrix} x \\ 1 \\ p_x \\ p_y \end{bmatrix}, \tag{1}$$

where x is the magnitude of a numeric variable, the constant 1 allows affine translations, and (p_x, p_y) encodes a control phasor in the plane.

2.1 Assignments and operators

An assignment or arithmetic operation is modeled as an affine linear operator acting on S. Example: adding a constant 2 to x is

$$A_{+2} = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad S' = A_{+2} S. \tag{2}$$

2.2 Loops and decisions as phasors

Iteration and decision-making are modeled by planar rotations (phasors). A rotation by angle
theta acting on the control plane is

$$R(theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\theta & -\sin\theta \\ 0 & 0 & \sin\theta & \cos\theta \end{bmatrix}. \tag{3}$$

We adopt the convention $+90^\circ$ (or $+\pi/2$) represents a logical "true" phasor and -90° a logical "false" phasor; iteration corresponds to repeated application of a rotation $R(\theta)$.

3 Strings as Vectors in VPL

In the Vectorial Programming Language (VPL), strings are represented as vectors within the *String type vector space*. Each character corresponds to a basis vector, similar to ASCII encoding, but treated as a proper mathematical vector. For instance, the string

`"Hello World"`

is represented as

$$S = \mathbf{h} \oplus \mathbf{e} \oplus \mathbf{l} \oplus \mathbf{l} \oplus \mathbf{o} \oplus \ \oplus \mathbf{w} \oplus \mathbf{o} \oplus \mathbf{r} \oplus \mathbf{l} \oplus \mathbf{d},$$

where each symbol $\mathbf{c}$ is a vector in the String subspace $\mathbb{V}_{\text{string}}$. This makes type casting (e.g. converting numbers to strings) a matter of *vector translation between subspaces*.

4 Data Structures as Vector Compositions

In VPL, all data structures are represented as compositions of vectors within their respective type subspaces. This provides a uniform algebraic foundation for both simple and complex structures.

4.1 Arrays and Lists

An array of n elements is a concatenation of vectors:

$$A = [a_1, a_2, \ldots, a_n], \quad a_i \in \mathbb{V}_{\text{type}}.$$

Each element a_i belongs to the vector subspace corresponding to its type (e.g., numbers, strings). Lists are treated similarly, with additional operators for dynamic insertion and removal as displacements within the sequence.

4.2 Tuples and Records

Tuples are direct sums of heterogeneous vectors:

$$T = \mathbf{x} \oplus \mathbf{y} \oplus \mathbf{z}, \quad \mathbf{x} \in \mathbb{V}_{\text{int}},\ \mathbf{y} \in \mathbb{V}_{\text{string}},\ \mathbf{z} \in \mathbb{V}_{\text{bool}}.$$

This corresponds to records or structs in traditional programming languages. Type casting is modeled as translation between subspaces.

4.3 Trees

A tree is expressed as a recursive tensor product:

$$\mathcal{T} = \mathbf{r} \otimes (\mathcal{T}_1 \oplus \mathcal{T}_2 \oplus \cdots \oplus \mathcal{T}_k),$$

where $\mathbf{r}$ is the root node vector, and $\mathcal{T}_i$ are child subtrees. Traversals are phasor-controlled iterations through this structure.

4.4 Graphs

Graphs generalize trees by allowing cycles and arbitrary adjacency:

$$\mathcal{G} = \{\mathbf{v}_i\} \oplus \{E_{ij}\},$$

where $\mathbf{v}_i \in \mathbb{V}_{\text{nodes}}$ and edges $E_{ij} \in \mathbb{V}_{\text{relations}}$ represent connections. Graph traversal corresponds to phasor-driven exploration across edge vectors.

4.5 Implications

By defining data structures as vector compositions, VPL unifies memory representation, traversal, and operations under a single vectorial and phasor framework. This abstraction is particularly suited for parallel execution, since independent substructures (e.g., subtrees or graph components) can be processed concurrently via tensor products.

5 Parallel Execution in VPL

Parallelism in VPL arises naturally because control flow is a vector pointing to multiple instructions simultaneously. A control vector may be expressed as a superposition of instruction vectors:

$$C = \alpha \cdot I_1 + \beta \cdot I_2,$$

where I_1 and I_2 are instructions and α, β weight their concurrent execution. This allows both SIMD-style data parallelism and task parallelism.

For data parallelism, an operator applied to a vector distributes over elements:

$$f([x_1, x_2, \ldots, x_n]) = [f(x_1), f(x_2), \ldots, f(x_n)].$$

6 Tensor Products for Concurrency

Independent tasks in VPL can be expressed as tensor products of their respective vector spaces:

$$\mathcal{S} = \mathcal{T}_1 \otimes \mathcal{T}_2 \otimes \cdots \otimes \mathcal{T}_m,$$

where each $\mathcal{T}_i$ is a task represented as a vector in its own subspace. Execution proceeds in parallel, and synchronization corresponds to projection back into a shared subspace at measurement.

This formalism allows clear modeling of concurrent execution units (e.g. GPU threads, SIMD lanes) while keeping semantics consistent with the vectorial and phasor-based foundation of VPL.

7 Worked example: matrix compilation

Consider the program fragment:

```
X = 0;
while (X < 5) {
    X = X + 2;
}
measure(X);
```

Start state (choose control phasor initially pointing along +x):

$$S_0 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{4}$$

One iteration (body then rotate) operator is:

$$O(\theta) = R(\theta) A_{+2}. \tag{5}$$

For $\theta = 90°$, and three iterations (0 to 2 to 4 to 6), the composed operator is O^3 and

$$S_{final} = O^3 S_0 = \begin{bmatrix} 6 \\ 1 \\ 0 \\ -1 \end{bmatrix}, \tag{6}$$

so $x = 6$ and the phasor lies at angle $-\pi/2$ (equivalently $3\pi/2$).

Matrix multiplication steps visualized

$$S_0 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad A_{+2} S_0 = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad R(\pi/2) A_{+2} S_0 = \begin{bmatrix} 2 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$A_{+2} R(\pi/2) A_{+2} S_0 = \begin{bmatrix} 4 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$R(\pi/2) A_{+2} R(\pi/2) A_{+2} S_0 = \begin{bmatrix} 4 \\ 1 \\ -1 \\ 0 \end{bmatrix}$$

$$A_{+2} \cdots = \begin{bmatrix} 6 \\ 1 \\ -1 \\ 0 \end{bmatrix}$$

$$O^3 S_0 = \begin{bmatrix} 6 \\ 1 \\ 0 \\ -1 \end{bmatrix}$$

Figure 1: Stepwise linear-algebra execution of the loop (one unrolled representation). Read from left to right one line at a time.

8 Mapping to quantum circuits

To obtain a quantum realization we choose an encoding. In a basis (computational) encoding, a 3-qubit register represents integer X in the range 0..7. The addition-by-2 operation

becomes a modular adder unitary U_{+2}: $|x\rangle \mapsto |x + 2 (\mathrm{mod})\, 8\rangle$. The phasor state is encoded in an auxiliary qubit whose phase is rotated by $R_z(\pi/2)$ each iteration. A controlled-adder implements the conditional semantics if $X < 5$ then $X \leftarrow X + 2$.

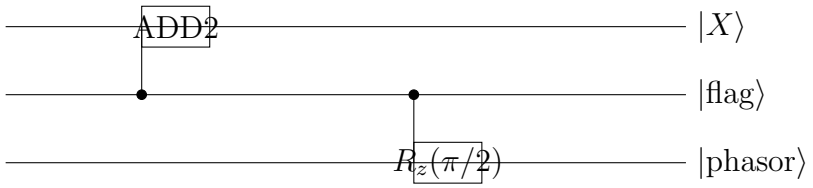

Figure 2: Quantum-circuit fragment: conditional add and phasor rotation.

9 Electrical circuit embodiment

A practical classical implementation uses well-known digital primitives:

- a 3-bit register (binary counter) storing X,
- an adder stage that computes $X + 2$ and parallel-loads it,
- a comparator that implements the predicate $(X < 5)$,
- a 2-bit phase register selecting one of four quadrature outputs (0°,90°,180°,270°), and
- an R-2R DAC to output analog measurements of X when desired.

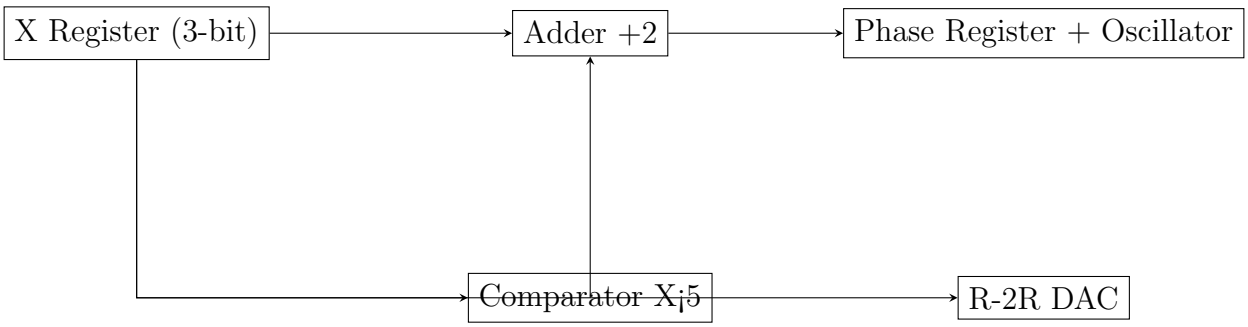

Figure 3: Block-level electrical realization.

10 Discussion

The vectorial-phasor model yields compact algebraic representations for programs; matrix compilation demonstrates how execution can be represented as operator products acting on an initial state. While predicates and branching introduce nonlinearity (runtime-dependent operator selection), many core constructs map naturally to linear or unitary maps.

11 Conclusion

We presented a geometric programming semantics with concrete compilation into matrix operators and practical mappings to quantum and electrical implementations. This framework invites further study on richer typing partitions, reversible embeddings, and performance trade-offs for hardware embodiments.

The Vector/Phasor Programming Language (VPL) introduces a novel way of expressing programming semantics through vectors and phasors. By treating assignments, operations, and data types as vectors, and control structures such as loops and decisions as phasors, VPL reframes computation in purely geometric terms. This provides a unified model that bridges abstract programming concepts with physical realizations in quantum systems, analog hardware, and electrical circuits.

VPL shines in several areas. As a theoretical framework, it unifies programming and linear algebra, offering an elegant geometric representation of data flow and control flow. As a compiler intermediate representation (IR), it could serve as a bridge between high-level programming languages and specialized backends such as quantum circuits, neuromorphic hardware, or GPU kernels. VPL also has clear potential as an educational tool, helping students understand that computation can be seen as trajectories in vector spaces, with loops as rotations and conditions as angular displacements. Finally, VPL is especially promising for unconventional computing paradigms, since its vector and phasor abstractions are native to quantum mechanics, signal processing, and oscillator-based analog computation.

Despite these advantages, VPL has limitations. Expressing symbolic or discrete logic requires embedding into vector spaces, which may be unintuitive or inefficient. Certain nonlinear operations do not fit neatly into a purely linear or phasor-based framework, requiring approximations or extensions. Moreover, the practical use of VPL will depend on building a supporting toolchain: compilers, interpreters, and runtime systems capable of lowering vector/phasor constructs into executable code or hardware mappings.

Future work should focus on developing a compiler pipeline for VPL, where high-level constructs are translated into a vector/phasor intermediate representation and then targeted to different backends such as CPUs, GPUs, FPGAs, or quantum computers. Another research direction involves building educational platforms to help students and practitioners visualize computation as geometric motion, which could transform how programming is taught. Finally, exploring VPL as a design language for emerging hardware models—including analog, neuromorphic, and quantum computing—could provide a unified framework for algorithm design across radically different computational substrates.

Acknowledgments

Since this is an early draft... this is void.

References

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2010.

[2] D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997.

[3] D. Hestenes, *New Foundations for Classical Mechanics*, Kluwer Academic, 1986.

[4] M. M. Mano and M. D. Ciletti, *Digital Design*, Pearson, 2017.