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

Lucio Guerchi
lugu16f@gmail.com

September 14, 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 Vector Space Model of Instructions

In the Vector Programming Language (VPL), control flow itself can be expressed as a vector in an *instruction space*. Instead of modeling instructions as sequential tape positions, each instruction is a basis vector in a finite-dimensional Hilbert space $\mathcal{I}$. The execution of a program is then represented by a control state $|C\rangle \in \mathcal{I}$ that evolves under linear operators.

7.1 Instruction Space

We define the instruction space as:

$$\mathcal{I} = \text{span}\{|I_0\rangle, |I_1\rangle, \ldots, |I_{N-1}\rangle\},$$

where each basis state corresponds to a program instruction. Execution proceeds by applying linear maps $M : \mathcal{I} \to \mathcal{I}$ that implement jumps, conditionals, and terminations.

7.2 Data Space

Program variables live in a separate vector space $\mathcal{D}$, whose basis represents classical values. For instance, an 8-value register uses

$$\mathcal{D} = \text{span}\{|0\rangle, |1\rangle, \ldots, |7\rangle\}.$$

The joint program state lies in the tensor product

$$\mathcal{J} = \mathcal{I} \otimes \mathcal{D}.$$

7.3 Operators

Execution is defined by sparse operators acting on $\mathcal{J}$:

- **Initialization:** $M_{init} = |I_1\rangle\langle I_0| \otimes I_{\mathcal{D}}$.
- **Comparison (Projection):** $M_{cmp} = |I_{true}\rangle\langle I_1| \otimes P + |I_{false}\rangle\langle I_1| \otimes (I - P)$, where P is a projector selecting valid data states.
- **Increment (Shift):** $M_{inc} = |I_{next}\rangle\langle I_k| \otimes S_{+c}$, with S_{+c} a shift matrix adding a constant c.
- **Jump:** $M_{jump} = |I_j\rangle\langle I_k| \otimes I_{\mathcal{D}}$.
- **Halt:** $M_{halt} = |I_{halt}\rangle\langle I_{halt}| \otimes I_{\mathcal{D}}$.

The full program operator is the sum of all instruction clauses:

$$M = M_{init} + M_{cmp} + M_{inc} + M_{jump} + M_{halt}.$$

7.4 Why Control Instructions Are Vectors

In the classical model of computation, control flow is an abstract sequencing mechanism. In VPL, however, we treat every instruction—including control instructions such as `if`, `while`, and `goto`—as basis vectors in the same instruction space. This is justified by the following:

1. **Uniform Representation:** By encoding both arithmetic and control operations as vectors, we avoid a dual system where some operations live in data space and others in an external controller. All computation reduces to state evolution in a single linear space.
2. **Linear Composition:** Control instructions transform execution by redirecting program flow. Representing them as vectors allows these redirections to be modeled as linear maps between basis states. For example, a conditional jump is simply a projection onto one subspace (true branch) or its orthogonal complement (false branch).
3. **Tensor Product with Data:** Once instructions are vectors, they can be tensored with data vectors. This makes the joint system $\mathcal{J} = \mathcal{I} \otimes \mathcal{D}$ capture both "what is the current instruction" and "what is the current data" in a unified state.

4. **Geometric Interpretation:** Treating control instructions as phasors (rotations in the instruction space) aligns with the intuition that loops and conditionals correspond to angular displacements. A `while` loop is then a repeated rotation until a measurement (projection) halts it. In this convention, a decision is encoded as an angular displacement:

   - $+90°$ corresponds to **true** (continue execution),
   - $-90°$ corresponds to **false** (halt or exit branch).

   This makes conditionals and loops interpretable as rotations in the complex plane, with branching behavior emerging from angular phase selection.

Thus, control instructions are not external directives but intrinsic vectors within $\mathcal{I}$, ensuring that the entire program, including flow control, remains representable as linear algebra.

8 Function Calls in VPL

In the vector space model, functions are treated as dedicated instruction subspaces, and a call is represented as a transition between the caller's instruction state and the callee's. This provides a uniform way to encode procedural abstraction without leaving the linear algebraic framework.

8.1 Instruction Space Extension

We extend the instruction space to include `call` and `return` basis states:

$$\mathcal{I} = \text{span}\{|I_{main}\rangle, |I_{call}\rangle, |I_{body}\rangle, |I_{ret}\rangle, |I_{halt}\rangle\}.$$

A call is a transition into the function's instruction subspace, while a return maps control back to the calling instruction.

8.2 Operators

- **Call Operator:**
$$M_{call} = |I_{body}\rangle\langle I_{call}| \otimes C_{args},$$
which transfers control to the function body while casting the input arguments from the caller's data space to the callee's data space.

- **Function Body:** Within the callee, computation is encoded as usual by arithmetic or logical operators acting on the data register.

- **Return Operator:**
$$M_{ret} = |I_{after}\rangle\langle I_{ret}| \otimes C_{res},$$
which transfers control back to the caller and casts the result into the caller's data space.

8.3 Example: Increment Function

Consider the following VPL-style function:

```
function f(X) {
    return X + 1;
}

Y = f(2);
```

Instruction Space.

$$\mathcal{I} = \text{span}\{|I_{main}\rangle, |I_{callf}\rangle, |I_{fbody}\rangle, |I_{retf}\rangle, |I_{halt}\rangle\}.$$

Operators.

$$\begin{aligned} M_{init} &= |I_{callf}\rangle\langle I_{main}| \otimes |2\rangle, \\ M_{callf} &= |I_{fbody}\rangle\langle I_{callf}| \otimes C_{args}, \\ M_{fbody} &= |I_{retf}\rangle\langle I_{fbody}| \otimes S_{+1}, \\ M_{retf} &= |I_{halt}\rangle\langle I_{retf}| \otimes C_{res}. \end{aligned}$$

Execution. Starting with

$$|\Psi_0\rangle = |I_{main}\rangle \otimes |0\rangle,$$

application of M_{init}, M_{callf}, M_{fbody}, and M_{retf} yields

$$|\Psi_{final}\rangle = |I_{halt}\rangle \otimes |Y = 3\rangle.$$

8.4 Properties

This construction shows:

- Functions are just linear subspaces in the instruction vector space.
- Function calls are transitions into these subspaces; returns are transitions back.
- Recursion is naturally modeled as loops in the instruction subspace.
- Concurrency is possible by preparing a superposition of multiple function calls, evolving in parallel.

8.5 Imports in VPL

In conventional programming languages, `import` or `include` statements bring in external functions and data types. In VPL, importing a package is expressed as an *extension of the vector spaces* for instructions and data.

Instruction Space Extension. Suppose a program imports a package `P`. The instruction space of the program expands by a direct sum:

$$\mathcal{I}_{total} = \mathcal{I}_{prog} \oplus \mathcal{I}_P,$$

where $\mathcal{I}_{prog}$ encodes the program's native instructions and $\mathcal{I}_P$ encodes the imported package instructions.

Data Space Extension. If the package introduces new types, the data space is likewise extended:

$$\mathcal{D}_{total} = \mathcal{D}_{prog} \oplus \mathcal{D}_P.$$

This ensures imported structures (e.g., numbers, strings, or graphics objects) reside in well-defined subspaces.

Operators from Imports. Each imported function corresponds to a new operator acting on the extended spaces. For instance, if the `Math` package provides a sine function, we represent it as

$$M_{\sin} = |I_{next}\rangle\langle I_{\sin}| \otimes S_{\sin},$$

where $S_{\sin}$ is the operator implementing the sin transformation on the data register.

Example. Consider the following VPL code:

```
import Math;

X = 0.5;
Y = sin(X);
measure(Y);
```

The instruction space becomes

$$\mathcal{I} = \text{span}\{|I_{init}\rangle, |I_{\sin}\rangle, |I_{halt}\rangle\}.$$

The operators are:

$$M_{init} = |I_{\sin}\rangle\langle I_{init}| \otimes |0.5\rangle,$$
$$M_{\sin} = |I_{halt}\rangle\langle I_{\sin}| \otimes S_{\sin}.$$

The final state is

$$|\Psi_{final}\rangle = |I_{halt}\rangle \otimes |\sin(0.5)\rangle.$$

Interpretation. In this framework:

- Imports correspond to extensions of the vector spaces,
- Imported functions are operators on the new subspaces,
- Scope resolution corresponds to projection into the relevant subspace.

Thus, importing a package in VPL is not a symbolic convenience but a concrete operation that enlarges the program's vectorial universe.

Example: Parallel Graph Traversal

Consider a graph with three nodes A, B, C where A connects to both B and C. A classical depth-first traversal would branch into two separate recursive calls, requiring explicit stack management. The number of possible paths grows exponentially with graph size.

In VPL, the traversal is expressed as a superposition of instruction states:

$$|\Psi_0\rangle = |I_A\rangle \otimes |data_A\rangle.$$

When the program evaluates the neighbors of A, instead of forking explicit processes, it performs a projection into the neighbor subspace:

$$|\Psi_1\rangle = \alpha_B |I_B\rangle \otimes |data_B\rangle + \alpha_C |I_C\rangle \otimes |data_C\rangle,$$

where α_B, α_C represent traversal weights (uniform if breadth-first, biased if heuristic-guided).

Subsequent evolution applies simultaneously:

$$|\Psi_2\rangle = M_{step} |\Psi_1\rangle,$$

with M_{step} advancing each instruction/data pair according to the graph structure. All paths are explored in parallel as components of the same vector state, without combinatorial explosion.

Interpretation. In classical algorithms, each traversal path is a separate process requiring explicit memory and control flow. In VPL, traversal states are naturally superposed vectors, and the program evolves them linearly. This provides a compact, geometric representation of parallel exploration, well-suited for search problems, AI, and symbolic reasoning.

9 Example: Parallel Execution in VPL

In the vector space model, parallelism arises by placing the control state into a superposition of multiple instructions. For instance, consider two loops that operate independently on different variables:

```
X = 0;   while (X < 5) { X = X + 2; }
Y = 1;   while (Y < 8) { Y = Y + 3; }
measure(X, Y)
```

Instruction Space. We define

$$\mathcal{I} = \text{span}\{|I_X\rangle, |I_Y\rangle, |I_{halt}\rangle\},$$

where $|I_X\rangle$ controls the loop on X, $|I_Y\rangle$ controls the loop on Y, and $|I_{halt}\rangle$ is the shared measurement stage.

Parallel Control State. Instead of evolving $|I_X\rangle$ or $|I_Y\rangle$ alone, we prepare a superposed state:

$$|C\rangle = \alpha|I_X\rangle + \beta|I_Y\rangle,$$

with coefficients α, β controlling relative scheduling weights. This models concurrent execution of both loops.

Joint Data Space. Variables are represented as a tensor product:

$$\mathcal{D} = \text{span}\{|0\rangle_X, \dots\} \otimes \text{span}\{|0\rangle_Y, \dots\}.$$

Execution. Each loop uses its own comparison projector and increment operator, but acts only on its respective subspace of $\mathcal{D}$. For example:

$$M_X = |I_X\rangle\langle I_X| \otimes (P_{X<5} S_{+2}^{(X)} + (I - P_{X<5})),$$
$$M_Y = |I_Y\rangle\langle I_Y| \otimes (P_{Y<8} S_{+3}^{(Y)} + (I - P_{Y<8})).$$

The combined program operator is:

$$M = M_X + M_Y + M_{halt}.$$

Interpretation. At each step, both X and Y evolve in parallel, with the control phasor oscillating between $|I_X\rangle$ and $|I_Y\rangle$. This models parallel execution not as interleaving, but as true simultaneous state evolution in the tensor product space.

10 Detailed Example: While Loop in VPL

We revisit the example:

$$X = 0; \quad \texttt{while } (X < 5)\{X = X + 2;\} \quad \texttt{measure}(X).$$

This section illustrates in full detail how the program is represented in VPL, how its execution unfolds step by step, how the control phasor operates, and why compact operators are preferred when mapping to circuits.

10.1 Fully Explicit Expansion

We define two vector spaces:

- Data space $\mathcal{D}$: the register X taking values $|0\rangle, |2\rangle, |4\rangle, |6\rangle, \dots$.
- Instruction space $\mathcal{I}$: states for comparison ($|I_{\text{cmp}}\rangle$), addition ($|I_{\text{add}}\rangle$), and halting ($|I_{\text{halt}}\rangle$).

The joint state lives in $\mathcal{H} = \mathcal{I} \otimes \mathcal{D}$.
At each step, the machine applies a joint operator M such that:

$$M(|I_{\text{cmp}}\rangle \otimes |x\rangle) = \begin{cases} |I_{\text{add}}\rangle \otimes |x\rangle, & x < 5 \\ |I_{\text{halt}}\rangle \otimes |x\rangle, & x \geq 5 \end{cases}$$
$$M(|I_{\text{add}}\rangle \otimes |x\rangle) = |I_{\text{cmp}}\rangle \otimes |x + 2\rangle.$$

Execution trace. Starting from $|I_{\text{cmp}}\rangle \otimes |0\rangle$, the steps evolve as:

$$\begin{aligned}
&|I_{\text{cmp}}\rangle \otimes |0\rangle \;\mapsto\; |I_{\text{add}}\rangle \otimes |0\rangle \\
&\mapsto\; |I_{\text{cmp}}\rangle \otimes |2\rangle \;\mapsto\; |I_{\text{add}}\rangle \otimes |2\rangle \\
&\mapsto\; |I_{\text{cmp}}\rangle \otimes |4\rangle \;\mapsto\; |I_{\text{add}}\rangle \otimes |4\rangle \\
&\mapsto\; |I_{\text{cmp}}\rangle \otimes |6\rangle \;\mapsto\; |I_{\text{halt}}\rangle \otimes |6\rangle.
\end{aligned}$$

Measurement yields $X = 6$.

10.2 The Control Phasor

The instruction subspace $\{|I_{\text{cmp}}\rangle, |I_{\text{add}}\rangle\}$ can be viewed as a phasor plane. As long as the predicate $X < 5$ holds, the control vector rotates cyclically:

$$|I_{\text{cmp}}\rangle \;\to\; |I_{\text{add}}\rangle \;\to\; |I_{\text{cmp}}\rangle \;\to \cdots$$

This rotation is interrupted only when the projection operator $P_{\geq 5}$ projects the control onto $|I_{\text{halt}}\rangle$. Thus, the loop itself is a phasor cycle in $\mathcal{I}$.

10.3 Compact Block Representation

Instead of expanding M into a full $(mn) \times (mn)$ operator on $\mathcal{H}$, we can factor the dynamics as block operators acting on $\mathcal{D}$, gated by control states in $\mathcal{I}$:

$$M = \begin{bmatrix} P_{<5} & 0 & 0 \\ A_{+2} & 0 & 0 \\ P_{\geq 5} & 0 & I \end{bmatrix}.$$

Here:

- $P_{<5}$: projection onto states with $X < 5$.
- $P_{\geq 5}$: projection onto states with $X \geq 5$.
- A_{+2}: increment operator $X \mapsto X + 2$.

This form is block-sparse and directly corresponds to physical circuit components.

10.4 Why Expansion Is Not Needed for Circuits

Although the expanded joint operator is useful for formal analysis, real circuits never implement the full transition matrix. Instead:

- Each block (adder, comparator, register) is a physical primitive.
- The control phasor (FSM state) selects which block acts in each step.
- Projection operators are realized as comparators feeding the FSM.

Thus, the compact form already matches hardware: adders, comparators, registers, and state machines. The expanded operator encodes the global truth table, but circuits only ever use the local transitions. The phasor interpretation remains intact: the FSM state register is the rotating phasor, and the comparator projection halts it.

11 Expansion and Circuit Mapping

11.1 Why Expansion Happens

In the Vectorial Programming Language (VPL), the program state is not defined solely by the data register but also by the control register, which keeps track of which instruction is currently active. Formally, the state space is a tensor product

$$\mathcal{H} = \mathcal{I} \otimes \mathcal{D},$$

where $\mathcal{I}$ is the instruction/control vector space and $\mathcal{D}$ is the data vector space.

To describe execution as a single linear operator M, the operator must act on the joint state. For example, when the control state is $|I_{\text{add}}\rangle$, the operator must map

$$M\left(|I_{\text{add}}\rangle \otimes |x\rangle\right) = |I_{\text{cmp}}\rangle \otimes A_{+2}\,|x\rangle\,,$$

where A_{+2} is the operator that adds 2 to the data register. Encoding this behavior requires embedding A_{+2} as a block inside M. Repeating this construction for every possible control state inflates M into a block matrix of dimension

$$\dim(M) = (mn) \times (mn),$$

where $m = \dim(\mathcal{I})$ and $n = \dim(\mathcal{D})$. This phenomenon is what we refer to as *expansion*.

The expansion is conceptually attractive because it unifies the semantics of the program into a single operator. Iteration corresponds to M^k, halting can be modeled as projection, and superpositions of control/data states can be described. In this sense, expansion shows that programs are fundamentally linear dynamical systems.

However, the cost of expansion is the dramatic growth in operator size. A 32-bit data register already requires 2^{32} basis states, and if the control space has dozens of states, the joint operator M becomes an astronomical $(mn) \times (mn)$ matrix. This is mathematically well-defined but computationally intractable to store or manipulate explicitly.

11.2 Why Expansion Is Not Needed for Circuits

Despite the mathematical appeal of expansion, real hardware does not require constructing the full operator M. Electrical circuits operate locally: the control register selects which functional block (e.g., adder, comparator, or register update) acts on the data register at each step. In this model, only the relevant block is applied, rather than an embedded version of all blocks inside a massive joint operator.

In practice, this means that instead of instantiating an $(mn) \times (mn)$ operator, a circuit simply consists of:

- a finite-state machine (FSM) to hold and update the control state,
- functional blocks such as adders or comparators to act on data,
- multiplexers that route data based on the control state, and

- registers to store intermediate results.

Thus, the expansion is a mathematical artifact of modeling everything as a single linear operator. For implementation purposes, one can factor the program into smaller operators applied conditionally, avoiding the exponential blow-up. The compact presentation of VPL is therefore sufficient for hardware mapping, since circuits realize conditional branching and control sequencing directly without ever materializing the expanded block matrix.

11.3 Expansion and Quantum Circuits

In the context of quantum computing, the question of expansion takes on a dual character. Formally, quantum mechanics already describes state evolution in terms of linear operators acting on a Hilbert space. If we model a program in VPL as a state in the joint space

$$\mathcal{H} = \mathcal{I} \otimes \mathcal{D},$$

then the correct description of execution is a unitary operator U on $\mathcal{H}$. This operator is, in principle, exactly the "expanded" form: a single global matrix acting on all components of the state.

However, just as in the classical electrical case, direct expansion is computationally infeasible. A register of n qubits lives in a 2^n-dimensional Hilbert space, and the corresponding operator is a $2^n \times 2^n$ matrix. For even modest n, this representation becomes impossible to handle explicitly.

Quantum hardware avoids this problem by implementing the global operator indirectly. A circuit is specified as a sequence of local gates (typically one- and two-qubit operations), along with controlled operations that condition the application of one unitary on the state of another subsystem. These gates decompose the global unitary into a product of simple components:

$$U = U_k \cdot U_{k-1} \cdots U_1,$$

where each U_i acts on only a small portion of the total space. In this way, the full expansion is never instantiated; instead, it is realized implicitly through composition.

Thus, in quantum circuits the situation mirrors the VPL–to–classical circuit mapping. *Expansion is essential at the semantic level*, since the physics is linear and global operators are the ground truth. *Expansion is not necessary at the implementation level*, because decomposition into localized gates suffices for actual execution. This duality emphasizes that while the expanded formalism provides clarity and unification, practical realizations must exploit factorization to avoid exponential blow-up.

12 Potential Applications of VPL

The Vector Programming Language (VPL) introduces a radically different model where assignments, loops, decisions, and even imports are all expressed as vectors or phasors in unified vector spaces. This uniformity gives VPL natural advantages in problems that are traditionally hard in classical computing.

12.1 Concurrency and Parallelism

In classical programming, parallel execution requires explicit scheduling, synchronization primitives, and careful bookkeeping. In VPL, parallelism arises naturally as tensor products of instruction and data subspaces:

$$\mathcal{J} = \mathcal{I}_1 \otimes \mathcal{D}_1 \quad \oplus \quad \mathcal{I}_2 \otimes \mathcal{D}_2.$$

This makes it straightforward to model independent threads of execution, synchronization barriers, and concurrent state updates.

12.2 Combinatorial Branching

Branching control flow often leads to a combinatorial explosion of possible execution paths. Classical methods use explicit path enumeration or backtracking. In VPL, a decision is simply a rotation in the instruction space:

$$\texttt{if}(\mathit{cond}) \quad \longrightarrow \quad R(\pm 90^\circ),$$

where the phasor points to the *true* or *false* subspace. No combinatorial blow-up occurs, since all branches are vectors within the same space. This compactly represents symbolic execution and speculative evaluation.

12.3 Loops and Oscillations

Loops are inherently circular in nature, and VPL models them as phasors under repeated rotation. This allows loops to be analyzed geometrically: finite loops correspond to bounded angular displacement, while infinite loops correspond to stable oscillations. This geometric perspective provides a new lens for studying termination and convergence.

12.4 Graph and Search Problems

Dynamic graph traversal, search trees, and symbolic AI often face exponential growth in the number of paths. In VPL, traversal states can exist in superposition:

$$|\Psi\rangle = \alpha_1 |I_{node1}\rangle \otimes |d_1\rangle \; + \; \alpha_2 |I_{node2}\rangle \otimes |d_2\rangle + \cdots$$

Each path is a component vector, and evolution applies simultaneously to all. This eliminates the need for explicit stacks or queues.

12.5 Quantum-Inspired Algorithms

Because VPL uses vector spaces, phasors, and projections as first-class elements, it is naturally aligned with quantum computation. While not a quantum language, it can directly simulate quantum-inspired algorithms (e.g., amplitude amplification, parallel search) more transparently than classical imperative languages.

Summary. The problems most difficult for classical approaches—those with *combinatorial explosion*, *parallel branching*, and *deep recursion*—are precisely the ones VPL makes easier. By reducing them to rotations, projections, and tensor products in linear algebra, VPL offers a compact representation where complexity is geometric rather than combinatorial.

13 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.

14 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.