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

Lucio Guerchi

Abstract

We introduce the Vectorial Programming Language (VPL), a formalism that unifies programming constructs under vector space and phasor representations. In VPL, assignments, operations, and data types are expressed as vectors in distinct spaces; loops and decisions are represented as phasors; and control is modeled by vector evolution in instruction space. This framework allows algorithms to be mapped naturally to both quantum and electrical circuits, providing a unified perspective for reasoning about concurrency, parallelism, and execution flow.

1 Introduction

Traditional models of programming describe execution in terms of sequential control flow, branching, and iteration. The Vectorial Programming Language (VPL) reformulates these constructs in terms of linear algebra and phasor dynamics. Assignments become displacements in vector spaces; conditionals become projections; loops correspond to phasor rotations; and control itself is a vector that evolves in instruction space.

This abstraction is motivated by three goals:

1. To provide a mathematical semantics for programming grounded in vector spaces.

2. To enable direct mappings to quantum computation, where linear operators and superposition are natural.

3. To permit hardware realizations in analog and digital circuits, where phasors, oscillators, and linear transformations naturally arise.

2 Formal Model

We define distinct vector spaces for different entities:

- Variables live in data vector spaces, partitioned by type (numeric, string, structured).
- Operations are linear operators acting on these vectors.
- Control flow resides in an instruction vector space $\mathcal{I}$, where each basis vector corresponds to an instruction.
- Loops and conditionals are modeled as phasors or projections acting on the control vector.

A program state at time t is described by the tuple

$$(\vec{d_t}, \vec{c_t}, \phi_t),$$

where $\vec{d_t}$ encodes data vectors, $\vec{c_t}$ encodes the control vector, and ϕ_t encodes loop phasors. Execution corresponds to the stepwise application of operators that update these components.

3 Control Vectors

In VPL, control flow is represented as a *control vector* living in the instruction space $\mathcal{I}$. Each instruction corresponds to a basis vector $\hat{i}_k \in \mathcal{I}$, and program execution is modeled as the evolution of the control vector $\vec{c_t}$.

3.1 Sequential Execution

In the simplest case, $\vec{c}_t$ is one-hot, pointing to a single instruction at time t. Execution corresponds to advancing this vector step by step:

$$\vec{c}_{t+1} = U\vec{c}_t,$$

where U is the instruction update operator.

3.2 Branching

Conditionals are represented as projections:

$$\vec{c}_{t+1} = \begin{cases} P_{\text{true}}\vec{c}_t, & \text{if condition is true,} \\ P_{\text{false}}\vec{c}_t, & \text{if condition is false.} \end{cases}$$

3.3 Loops as Phasors

Loops are modeled as phasors, encoding cyclical evolution:

$$\phi_{t+1} = \phi_t + \Delta\theta,$$

with exit determined by a projection threshold.

3.4 Parallel Execution

The control vector may be a linear combination of instruction states:

$$\vec{c}_t = \alpha\hat{i}_k + \beta\hat{i}_m.$$

This encodes parallel execution: both instructions $\hat{i}_k$ and $\hat{i}_m$ are active simultaneously, evolving under the update operator:

$$\vec{c}_{t+1} = \alpha U\hat{i}_k + \beta U\hat{i}_m.$$

This models concurrency as a natural feature of the vector space formalism.

4 Functions and Function Calls

Practical programming requires modularity. In VPL, functions are modeled as *subspaces* of the instruction space $\mathcal{I}$.

4.1 Functions as Subspaces

A function f is defined by a sequence of basis vectors $\{\hat{i}_1^f, \dots, \hat{i}_n^f\}$ spanning a subspace $\mathcal{I}_f \subset \mathcal{I}$. Entering a function projects the control vector into $\mathcal{I}_f$.

4.2 Function Calls

A function call is a transition from $\mathcal{I}_{\text{caller}}$ to $\mathcal{I}_f$, while storing a return vector $\hat{r}_{\text{caller}}$ to resume execution:

$$\vec{c}_{t+1} = P_f \vec{c}_t + \hat{r}_{\text{caller}}.$$

4.3 Recursion

Recursion corresponds to repeated projections into the same subspace, stacking return vectors. Hardware interpretation: nested oscillatory modes.

4.4 Parallel Function Calls

Superpositions enable parallel calls:

$$\vec{c}_t = \alpha P_{f_1} \vec{c}_t + \beta P_{f_2} \vec{c}_t.$$

Thus, modularity and concurrency are unified in the vectorial framework.

5 Strings as Vectors

In VPL, strings are represented as vectors in a dedicated type subspace. Each character corresponds to a basis vector, with an index analogous to ASCII or Unicode. For example, the string

"Hello"

is expressed as the ordered tuple of character vectors:

$$\vec{s} = (\hat{c}_H, \hat{c}_e, \hat{c}_l, \hat{c}_l, \hat{c}_o).$$

Operations on strings, such as concatenation, are modeled as vector concatenation or direct sums of sub-vectors. Casting between strings and other types (e.g., numbers) corresponds to linear maps between type subspaces.

6 Data Structures

Complex data structures arise as compositions of vector subspaces.

6.1 Arrays

An array of length n of type T is represented as an n-tuple of vectors $\vec{a} = (\vec{t}_1, \vec{t}_2, \ldots, \vec{t}_n)$, where each $\vec{t}_i \in \mathcal{V}_T$.

6.2 Tuples

Tuples combine heterogeneous types. A pair (x, s) where x is numeric and s is a string is modeled as a direct sum:

$$\vec{u} = \vec{x} \oplus \vec{s}.$$

6.3 Graphs

Graphs are naturally expressed as adjacency structures in vector form. A graph $G = (V, E)$ can be represented by its adjacency matrix A, where A_{ij} is the weight of the edge from vertex i to j. Traversal algorithms in VPL operate by propagating control vectors across this adjacency structure.

7 Parallel Execution: Algorithmic Examples

The vectorial model of control naturally supports parallel execution.

7.1 Two While Loops in Parallel

Consider the program:

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

In VPL, this is expressed as two control phasors ϕ_X, ϕ_Y evolving independently. The joint control vector is:

$$\vec{c}_t = \vec{c}_t^{X} \oplus \vec{c}_t^{Y}.$$

Execution of both loops proceeds in parallel, with each subspace updated by its own operator.

7.2 Parallel Graph Traversal

Let G be a graph with adjacency matrix A. A breadth-first traversal starting from node v_0 can be expressed by evolving a control vector over multiple nodes simultaneously:

$$\vec{c}_{t+1} = A\vec{c}_t.$$

This naturally expands the active frontier of the traversal in parallel, without explicit iteration over neighbors. The linearity of A allows parallelism to be expressed compactly.

8 Input and Output in VPL

Input and output (I/O) operations in VPL are modeled as projections between the internal program vector spaces and external environments. Unlike in classical models where I/O is primitive and opaque, in VPL it is a natural extension of the vectorial formalism.

8.1 Input as Injection

An input operation corresponds to injecting an external vector into the program state. Formally, reading a value v into a variable x of type T is modeled as:

$$\vec{d}_{t+1} = \vec{d}_t \oplus \hat{v}_T,$$

where $\hat{v}_T \in \mathcal{V}_T$ is the basis vector representing v in the type space T.

Hardware mapping: input devices serve as sources that project external signals into the internal vector space representation.

8.2 Output as Projection

Output is the dual operation: projecting an internal vector onto an external interface space. Writing a variable x of type T corresponds to:

$$\text{out}(x) = P_{\text{ext}}\vec{x}_T,$$

where P_{ext} is the projection operator into the external I/O subspace.

Hardware mapping: output devices implement this projection, e.g., a DAC projecting numeric vectors into voltage levels.

8.3 Streams and Continuous I/O

When inputs or outputs are streams, the projection/injection is repeated at each time step, effectively forming a tensor product of the internal state with a time-indexed external vector:

$$\vec{d}_{t+1} = \vec{d}_t \otimes \hat{v}_{T,t}.$$

8.4 Control Interaction with I/O

Because control vectors may branch or split in parallel, different parts of a program can handle different I/O channels simultaneously. This provides a natural vectorial model for concurrent I/O handling, without requiring explicit threading primitives.

9 Examples

To illustrate how VPL expresses computation, we provide several examples ranging from simple control flow to sorting and graph algorithms. Each example is presented with its VPL encoding, step-by-step explanation, and interpretation in terms of vector spaces, control vectors, and hardware mapping.

9.1 Single While Loop

Consider the classical loop:

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

VPL Encoding.

$$\vec{X}_0 = \hat{0}, \quad \vec{X}_{t+1} = \begin{cases} \vec{X}_t + \hat{2}, & \text{if } P_{<5}(\vec{X}_t) = 1 \\ \vec{X}_t, & \text{otherwise.} \end{cases}$$

Step-by-step Execution.

1. $X = 0$, condition true ($0 < 5$), update $X = 2$.
2. $X = 2$, condition true ($2 < 5$), update $X = 4$.
3. $X = 4$, condition true ($4 < 5$), update $X = 6$.
4. $X = 6$, condition false ($6 \not< 5$), loop exits.

Control Vector. The loop is governed by a control phasor that rotates $+90°$ each time the condition is true and $-90°$ when false. This phasor thus encodes the progression of control flow.

Compact Representation. Rather than expanding all steps, we can express the loop as a single operator:

$$U_{\text{while}} = \prod_{t=0}^{\infty} \left[P_{<5} \cdot (T_{+2}) + (I - P_{<5}) \right],$$

where T_{+2} is the translation operator ($X \mapsto X + 2$) and $P_{<5}$ is the projection operator for the condition.

9.2 Two Parallel While Loops

We now consider two loops running in parallel:

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

VPL Encoding. Each loop is its own phasor:

$$\vec{X}_{t+1} = \vec{X}_t + \hat{1} \quad (\text{until } X \geq 5),$$
$$\vec{Y}_{t+1} = \vec{Y}_t + \hat{2} \quad (\text{until } Y \geq 3).$$

The combined state is the tensor product:

$$\vec{Z}_t = \vec{X}_t \otimes \vec{Y}_t.$$

Control Vectors. Parallel control is expressed as two independent phasors evolving in different subspaces. This naturally models concurrency without explicit threads.

9.3 Bubble Sort

Bubble sort iteratively compares adjacent elements and swaps them if necessary.

VPL Encoding. Each comparison is a projection operator:

$$P_{a>b}(\vec{a}, \vec{b}) = \begin{cases} (\vec{b}, \vec{a}), & \text{if } a > b \\ (\vec{a}, \vec{b}), & \text{otherwise.} \end{cases}$$

Loop Structure. The nested loops of bubble sort are encoded as phasors: the outer loop advances over passes, the inner loop advances over indices.

Interpretation. Sorting is thus expressed as repeated projection-and-swap operators over the array vector space.

9.4 Graph Traversal in Parallel

Graph traversal (e.g., breadth-first search) can be expressed in VPL as parallel updates over a frontier set.

VPL Encoding. Let F_t be the frontier at time t. Then:

$$F_{t+1} = \bigcup_{v \in F_t} \mathrm{Adj}(v) \setminus V_{\text{visited}},$$

with V_{visited} updated in parallel.

Control Vector. The control phasor branches into multiple paths, one for each vertex in the frontier, reflecting true parallel execution.

Interpretation. This shows how VPL naturally expresses concurrency and parallel graph algorithms without explicit synchronization.