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 $\{\hat{i}_1^f, \ldots, \hat{i}_n^f\}$ 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

$$\text{"Hello"}$$

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:

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

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.

9.5 Numerical Example: fully explicit matrix computations

We illustrate the VPL semantics with a fully explicit numeric example and show *every* linear-algebra operation in detail.

Program

```
X = 0;
while (X < 3) { X = X + 1; }
measure(X);
```

Finite state choices and bases

Choose small finite spaces so we can write concrete matrices.

$$\text{Control basis } \mathcal{C} = \{\, |init\rangle, |loop\rangle, |add\rangle, |halt\rangle \,\}, \qquad \text{Data basis } \mathcal{D} = \{\, |0\rangle, |1\rangle, |2\rangle, |3\rangle \,\}.$$

Dimensions: $\dim \mathcal{C} = 4$, $\dim \mathcal{D} = 4$. The joint space is $\mathcal{H} = \mathcal{C} \otimes \mathcal{D}$ with dimension 16.

We adopt the natural *control-major* ordering for the 16-component joint vectors: first the 4 data components for 'init', then the 4 for 'loop', then 'add', then 'halt'. Concretely the 16 indices correspond to

$$\begin{array}{rl} 0: & (init, 0), \quad 1 : (init, 1), \quad 2 : (init, 2), \quad 3 : (init, 3), \\ 4: & (loop, 0), \quad 5 : (loop, 1), \quad 6 : (loop, 2), \quad 7 : (loop, 3), \\ 8: & (add, 0), \quad 9 : (add, 1), \quad 10 : (add, 2), \quad 11 : (add, 3), \\ 12: & (halt, 0), \quad 13 : (halt, 1), \quad 14 : (halt, 2), \quad 15 : (halt, 3). \end{array}$$

Data operators (explicit numeric matrices)

Identity on the data space:

$$I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Projector for the predicate $X < 3$:

$$P_{<3} = \mathrm{diag}(1,1,1,0) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Adder A that implements $|k\rangle \mapsto |k+1\rangle$ for $k = 0, 1, 2$ and saturates at $|3\rangle$:

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

Compute the combined operator that is used when the predicate is true:

$$AP \;=\; A\,P_{<3}.$$

Because $P_{<3}$ is diagonal, multiplying on the right scales the columns of A by the diagonal entries of $P_{<3}$. Concretely, column j of AP equals column j of A multiplied by P_{jj}. Thus (columns numbered 0..3)

$$AP = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Check by element formula:

$$(AP)_{i,j} = \sum_{k=0}^{3} A_{i,k} P_{k,j} = A_{i,j} P_{j,j},$$

so column 3 of AP is column 3 of A times $P_{33} = 0$, producing the zero column.

Also compute the complement projector

$$I_4 - P_{<3} = \operatorname{diag}(0,0,0,1) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Control (rank-1) matrices and Kronecker products

Each control transition is encoded by a rank-1 control matrix $|t\rangle\langle s|$ (where s = source control state, t = target control state). For example the control matrix $|loop\rangle\langle init|$ is the 4×4 matrix with a single 1 at row position 'loop', column 'init' and zeros elsewhere:

$$|loop\rangle\langle init| = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

The global contribution of a transition is the Kronecker product of the control matrix with the corresponding data operator. E.g.

$$(|loop\rangle\langle init|) \otimes I_4$$

is a 16×16 matrix whose block structure places I_4 in the block (row block = 'loop', column block = 'init') and zeros elsewhere. Concretely, Kronecker with a rank-1 control matrix inserts the data matrix into a single 4×4 block of the global 16×16 matrix.

Full program operator (block form)

We assemble the full operator M as the sum of the relevant control$\otimes$data terms:

$$\begin{aligned} M \;=\; & |loop\rangle\langle init| \;\otimes\; I_4 \;+\; |add\rangle\langle loop| \;\otimes\; (AP_{<3}) \\ & +\; |halt\rangle\langle loop| \;\otimes\; (I_4 - P_{<3}) \\ & +\; |loop\rangle\langle add| \;\otimes\; I_4 \;+\; |halt\rangle\langle halt| \;\otimes\; I_4. \end{aligned}$$

Interpretation of each term:

- $|loop\rangle\langle init| \otimes I_4$: init → loop, data unchanged.
- $|add\rangle\langle loop| \otimes (AP)$: when in ‘loop‘ and predicate true, apply AP and go to ‘add‘.
- $|halt\rangle\langle loop| \otimes (I - P)$: when in ‘loop‘ and predicate false, go to ‘halt‘.
- $|loop\rangle\langle add| \otimes I_4$: after ‘add‘, return to ‘loop‘.
- $|halt\rangle\langle halt| \otimes I_4$: ‘halt‘ is absorbing.

Explicit 16×16 **numeric matrix** M

Write M as a 4×4 block matrix with 4×4 blocks. Using the control order $(init, loop, add, halt)$ and data order $(0, 1, 2, 3)$, the nonzero blocks are:

$$\begin{aligned} B_{1,0} &= I_4, \\ B_{2,1} &= AP, \\ B_{3,1} &= I_4 - P_{<3}, \\ B_{1,2} &= I_4, \\ B_{3,3} &= I_4, \end{aligned}$$

all other 4×4 blocks equal the zero matrix.

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 \\ I & 0 & I & 0 \\ 0 & AP & 0 & 0 \\ 0 & I - P & 0 & I \end{bmatrix},$$

If M is expanded a row-by-row (16 rows, each row has 16 entries) contains:

- Rows 0–3 are all zero (no transitions target ‘init‘).
- Rows 4–7 contain two identity blocks (placed at columns corresponding to ‘init‘ and ‘add‘), giving the ‘init→loop‘ and ‘add→loop‘ transitions.
- Rows 8–11 contain the AP block in the column for ‘loop‘.
- Rows 12–15 implement the ‘loop→halt‘ and ‘halt→halt‘ behavior.

Initial joint state vector

Initialization assigns $X = 0$ and places control in 'init'. The joint vector is the 16×1 one-hot vector:

$$v_0 = |\Psi_0\rangle = |init\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

(1 in index 0, zeros elsewhere).

Step-by-step multiplications (explicit element arithmetic)

We will compute $v_{t+1} = Mv_t$ for each step and show the nonzero arithmetic operations explicitly. Because v_t is one-hot at each step, each matrix multiply reduces to selecting a single column of M (the column with the 1 in v_t).

Step 1: $v_1 = Mv_0$.

Because v_0 has a single nonzero entry at index $j = 0$, the product $v_1 = Mv_0$ equals the first column of M. Inspect column 0 of M: the only nonzero entry is at row $i = 4$ with value 1 (this comes from the I_4 block in $B_{1,0}$). Concretely:

for each row i:

$$(v_1)_i = \sum_{j=0}^{15} M_{i,j}(v_0)_j = M_{i,0} \cdot 1.$$

All $M_{i,0} = 0$ except $M_{4,0} = 1$. Hence

$$v_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = |loop\rangle \otimes |0\rangle.$$

Step 2: $v_2 = Mv_1$.

Now v_1 has its single 1 at index $j = 4$ (which is '(loop,0)'). So v_2 equals column 4 of M. To find nonzero entries in column 4, examine the blocks in block-column 1 (the 'loop' column): the nonzero block there are $B_{2,1} = AP$ and $B_{3,1} = I - P$. Their columns provide the nonzeros.

Compute the contribution of $B_{2,1} = AP$ (placed at global rows 8..11):

- Column index within the block: $j_{\text{in}} = 4 \bmod 4 = 0$. So we take column 0 of AP:

$$\text{col}_0(AP) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}.$$

Therefore global rows 8..11 receive values $[0, 1, 0, 0]^T$. The $B_{3,1} = (I - P)$ block's column 0 is $[0, 0, 0, 0]^T$, so it contributes nothing. All other blocks corresponding to column 4 are zero. Hence column 4 of M has a single 1 at global row 9. Explicitly:

$$v_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = |add\rangle \otimes |1\rangle.$$

This corresponds to: from 'loop' with $X = 0$, predicate true, apply A giving $X \mapsto 1$, and move control to 'add'.

Step 3: $v_3 = Mv_2$.

Now v_2 has a 1 at index $j = 9$ ('add',1). Column 9 of M is in block column 2 (the 'add' column). Nonzero block in that column is $B_{1,2} = I_4$ (placed at global rows 4..7). Take column index in-block $j_{\text{in}} = 9 \bmod 4 = 1$ (this selects column 1 of I_4, which is $[0, 1, 0, 0]^T$). So column 9 of M equals a vector with entries $4..7 = [0, 1, 0, 0]^T$ and zeros elsewhere. Thus

$$v_3 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = |loop\rangle \otimes |1\rangle.$$

(Which is the intended "return from add to loop, data=1".)

Step 4: $v_4 = Mv_3$.

Now v_3 has 1 at $j = 5$ ('loop',1). Consider column 5 of M (block column 1, in-block column 1). The AP block column 1 is $[0, 0, 1, 0]^T$, so rows 8..11 will get $[0, 0, 1, 0]^T$ (i.e. a 1 at global row 10). $I - P$ column 1 is zeros. So

$$v_4 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = |add\rangle \otimes |2\rangle.$$

This corresponds to applying A to $X = 1$ producing $X = 2$.

Step 5: $v_5 = Mv_4$.

From $j = 10$ (‘add‘,2). Column 10 lies in block column 2, in-block column 2. The I_4 block at $B_{1,2}$ has column 2 equal to $[0, 0, 1, 0]^T$, so rows 4..7 get that vector and we obtain

$$v_5 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = |loop\rangle \otimes |2\rangle.$$

Step 6: $v_6 = Mv_5$.

Now $j = 6$ (‘loop‘,2). Column 6 is block column 1, in-block column 2. Column 2 of AP equals $[0, 0, 0, 1]^T$ (recall AP’s third column); so rows 8..11 receive $[0, 0, 0, 1]^T \rightarrow$ a 1 at global row 11:

$$v_6 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = |add\rangle \otimes |3\rangle.$$

(So $X = 2 \xrightarrow{A} X = 3$.)

Step 7: $v_7 = Mv_6$.

From $j = 11$ (‘add‘,3). Column 11 is in block column 2, in-block column 3. The I_4 block at $B_{1,2}$ has column 3 equal to $[0, 0, 0, 1]^T$, so rows 4..7 get $[0, 0, 0, 1]^T \to 1$ at global row 7:

$$v_7 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = |loop\rangle \otimes |3\rangle.$$

Step 8 (predicate false): $v_8 = Mv_7$.

Now $j = 7$ (‘loop‘,3). Column 7 is in block column 1, in-block column 3. Column 3 of AP is $[0, 0, 0, 0]^T$ (because $P_{33} = 0$ zeroed that column), so the ‘add‘ branch contributes nothing. The ‘halt‘ branch uses $I - P$: column 3 of $I - P$ equals $[0, 0, 0, 1]^T$, so rows 12..15 receive $[0, 0, 0, 1]^T$ giving a 1 at global row 15. Also there is an absorbing $B_{3,3} = I_4$ block but that affects subsequent iterations (it keeps halt absorbing). Thus

$$v_8 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = |halt\rangle \otimes |3\rangle.$$

Once in ‘halt‘, further applications of M leave the state unchanged because of the $|halt\rangle\langle halt| \otimes I_4$ term (identity block at $B_{3,3}$).

Explicit arithmetic in selected steps

We now show the small matrix multiplications used above in raw arithmetic form (row-by-column sums) for two representative cases.

(i) Compute AP explicitly:

$$AP = AP_{<3} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Row-by-column checking (example entry $(3, 2)$):

$$(AP)_{3,2} = \sum_{k=0}^{3} A_{3,k}P_{k,2} = A_{3,2}P_{2,2} + A_{3,3}P_{3,2} = 1 \cdot 1 + 1 \cdot 0 = 1.$$

(ii) Example multiplication producing v_6: We had $v_5 = |loop\rangle \otimes |2\rangle$ which is the standard basis vector with 1 at index $j = 6$. To compute $v_6 = (Mv_5)$, take column $j = 6$ of M. The only nonzero block in column 1 (columns 4..7) that contributes is AP. Column within block $j_{\text{in}} = 6 \bmod 4 = 2$ (the third column). Column 2 of AP is $[0, 0, 0, 1]^T$. Writing the multiplication explicitly as sums of products for the global row $i = 11$ (which became 1):

$$(v_6)_{11} = \sum_{j=0}^{15} M_{11,j}(v_5)_j = M_{11,6} \cdot 1.$$

But $M_{11,6}$ equals the entry $(AP)_{3,2} = 1$ (indexing from 0), so $(v_6)_{11} = 1$. All other rows $i \neq 11$ have $M_{i,6} = 0$ so their components are zero. Thus the product reduces to the single multiplication shown above.

Measurement and final result

After reaching the halting joint state $v_8 = |halt\rangle \otimes |3\rangle$, a measurement of the data register returns the basis index 3, i.e. $X = 3$, which matches the program semantics.

Remarks and interpretation

- Every step of program execution corresponds to selecting a column of M (because v_t is one-hot) and copying that column into the new state vector. This is why the block/factored form is convenient: we only need the small data operator associated to the active control column.
- The construction is completely explicit and finite: all matrix entries are 0/1 in this toy example, and all arithmetic reduces to sums of products of 0/1 with vector entries (hence selection of columns).
- For larger data spaces the same pattern holds: blocks become larger, but the semantics is still block-sparse and compositional.

10 Applications

While the previous section presented foundational examples of how VPL encodes basic algorithms, in this section we demonstrate how the formalism naturally extends to more advanced domains. We illustrate this with two key applications: linear algebra (via BLAS routines) and spectral methods (via the Fast Fourier Transform, FFT).

10.1 BLAS: Vector Operations

Consider the classical BLAS Level 1 operation:

$$y \leftarrow \alpha x + y,$$

where $\alpha \in \mathbb{R}$, and $x, y \in \mathbb{R}^n$ are vectors.

VPL Representation. In VPL, this becomes:

$$\vec{y}_{t+1} = \alpha \cdot \vec{x}_t + \vec{y}_t,$$

where $\vec{x}$ and $\vec{y}$ are elements of the numeric vector space. The scalar α is a magnitude applied uniformly across the basis vectors of $\vec{x}$.

Control Vector. No branching control is needed, as this operation is purely linear. The control vector simply advances one step to indicate the completion of the update.

Hardware Mapping. In hardware, this corresponds to:

- multipliers applying α to each component of x,
- adders summing the results with y, and
- registers storing the updated y.

This shows how VPL describes classical linear algebra routines in a unified vectorial language.

10.2 Fast Fourier Transform (FFT)

The FFT decomposes a vector of N samples into frequency components via recursive butterfly operations.

VPL Representation. The FFT recursion can be expressed in VPL as:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}.$$

Here, the exponential factor is a phasor, naturally matching the VPL formalism where loops and decisions are represented as rotations in the complex plane.

Butterfly Operation. The core step,

$$(u, v) \mapsto (u + \omega v, u - \omega v),$$

is directly modeled in VPL as a pair of vector updates, where ω is a phasor $e^{-i2\pi k/N}$.

Control Vector. The recursion tree of FFT maps to a structured control vector path, with parallel execution across independent butterfly operations.

Hardware Mapping. Mapping the FFT to hardware under VPL yields:

- registers storing intermediate values,
- complex multipliers implementing ω,
- adders/subtractors forming butterflies, and
- a parallel control vector distributing execution across independent branches.

This shows how VPL provides a compact yet expressive formalism that unifies numeric linear algebra and spectral algorithms within the same framework.

11 Electrical Circuit Mappings

One of the most powerful aspects of VPL is its ability to map abstract vector-based computations directly to realizable hardware components. This section describes how algorithms expressed in VPL can be implemented as electrical circuits using standard digital and analog primitives.

11.1 Mapping Principles

The mapping follows these correspondences:

- **Variables as Registers:** Each program variable corresponds to a hardware register or memory cell holding the current vector state.
- **Operators as Functional Units:** Arithmetic and logical vector operations map to adders, multipliers, comparators, and multiplexers.
- **Phasors as Oscillators:** Control phasors are realized as counters or phase accumulators, often coupled with oscillators or clocked finite-state machines.
- **Projections as Comparators:** Decision statements and conditions map to comparators that gate the evolution of data paths.
- **Parallel Execution as Parallel Circuits:** Independent control vectors instantiate parallel circuit paths, naturally realizing concurrency.

11.2 While Loop Example

Consider again the simple loop:

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

The mapping uses:

- a register storing X,
- an adder computing $X + 2$,
- a comparator implementing $(X < 5)$,
- a multiplexer feeding back the updated value when the condition is true,
- a control phasor implemented as a counter that determines when the loop exits, and

- an output projection implemented as a DAC (for analog) or bus connection (for digital).

The flow of data thus mirrors the loop structure: the comparator checks the condition, the multiplexer chooses between updating or holding the register, and the control phasor ensures loop termination.

11.3 FFT Example

For the FFT, the mapping is more involved but still systematic:

- **Butterfly units** consist of adders and subtractors operating in parallel.
- **Twiddle factor multipliers** implement the phasor $\omega = e^{-i2\pi k/N}$ via sine/cosine lookup tables and complex multipliers.
- **Registers** store intermediate results between stages.
- **Control vectors** orchestrate the recursive decomposition, distributing inputs across butterfly units at each stage.

Interpretation. The recursive structure of the FFT maps naturally to a multi-stage network of parallel butterfly circuits, with the VPL control phasor guiding data flow through each stage.

11.4 General Discussion

Unlike traditional hardware description languages, VPL provides a higher-level semantic mapping from algorithms to circuits. Instead of explicitly wiring signals, the programmer expresses computation in terms of vectorial operations and control phasors. The mapping rules then systematically generate the corresponding circuit structure. This allows for:

- compact representations of iterative and recursive structures,
- natural modeling of parallelism,
- and integration of both digital and analog primitives under a unified mathematical formalism.

12 Quantum Circuit Mappings

Beyond classical and electrical hardware, VPL can also be mapped naturally to quantum circuits. The vectorial and phasor-based formalism of VPL aligns closely with the Hilbert space model of quantum computation, where states are vectors and operations are unitary matrices.

12.1 Mapping Principles

The mapping between VPL constructs and quantum circuits follows:

- **Variables as Qubits:** Each variable corresponds to one or more qubits, depending on the precision required. For example, an integer variable X may be stored in a register of n qubits representing its binary value.
- **Operators as Unitaries:** Arithmetic and logical vector operations correspond to unitary gates (e.g., controlled-NOT, Toffoli, quantum adders).
- **Phasors as Phase Gates:** Control phasors are implemented as rotations in Hilbert space, such as $R_z(\theta)$ or controlled-phase gates.
- **Projections as Measurements:** Conditional statements map to projective measurements, collapsing quantum states into classical outcomes.
- **Parallel Execution as Superposition:** Independent control branches can be represented as quantum superpositions, enabling natural modeling of parallel paths.

12.2 While Loop Example

Reconsider the simple loop:

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

Quantum Mapping.

- X is stored in a quantum register of qubits.
- The operation $X = X + 2$ is implemented as a quantum adder circuit.
- The condition $(X < 5)$ is a comparator circuit controlling whether the adder applies, which can be expressed as a controlled operation.
- The phasor controlling the loop corresponds to a phase rotation encoding loop continuation or exit.
- Finally, `measure`(X) is realized as a projective measurement of the quantum register.

12.3 FFT Example

The FFT has a particularly elegant quantum counterpart, the Quantum Fourier Transform (QFT). Its mapping in VPL highlights the synergy between the formalisms.

Quantum Mapping.

- Input vector x is encoded into a quantum register.
- Each butterfly operation is replaced by Hadamard and controlled-phase gates.
- The twiddle factors $e^{-i2\pi k/N}$ correspond directly to quantum phase rotations, which are native in QFT.
- The recursive decomposition of FFT maps to a quantum circuit with depth $O(\log^2 N)$, leveraging VPL's parallel phasor interpretation.

Interpretation. VPL's phasor-based representation makes the QFT nearly identical to its classical FFT description, except that quantum superposition provides exponential compression. Thus, VPL serves as a natural bridge between classical and quantum Fourier analysis.

12.4 General Discussion

Quantum circuit mappings highlight the universality of VPL:

- **Control phasors** become phase rotations,
- **vector updates** become unitary operators,
- **conditionals** map to measurements or controlled gates,
- and **parallel execution** is modeled either as concurrency or as superposition.

This dual compatibility with classical and quantum hardware suggests that VPL can function as a unifying intermediate representation for heterogeneous computing systems.

13 Discussion

VPL provides a unifying framework for representing programs as vectors and phasors, enabling seamless mappings to classical, electrical, and quantum hardware. This abstraction highlights several strengths:

- **Universality:** Control, data, and I/O operations are expressed in the same mathematical formalism.
- **Parallelism:** Both concurrency and superposition are naturally represented through tensor products and phasors.
- **Hardware Transparency:** Programs can be interpreted directly as circuit structures without requiring intermediate compilation layers.

However, challenges remain. The state space can expand combinatorially in explicit form, requiring compact operators to remain tractable. Moreover, noise and resource constraints limit the scalability of electrical and quantum mappings. These issues point to directions for refinement.

14 Conclusion

We presented the Vector Phasor Language (VPL), a programming model where assignments, data types, loops, and control flow are expressed as vectors and phasors. VPL unifies program semantics with mathematical objects, allowing straightforward mappings to electrical circuits and quantum circuits.

The compact operator view of control and data evolution makes VPL expressive yet hardware-friendly. By treating control as a vector, VPL naturally accommodates parallel execution, recursion, and concurrency. Applications such as BLAS routines and FFT demonstrate the practicality of the approach.

15 Future Work

Future research should explore:

- **Noise analysis:** quantifying limits in analog and quantum implementations.
- **Libraries:** mapping standard computational libraries (e.g., linear algebra, signal processing, machine learning) into VPL.
- **Optimization:** developing algebraic simplifications to minimize circuit complexity during mapping.
- **Toolchains:** building compilers from high-level languages into VPL and onward to hardware descriptions.
- **Applications:** studying whether problems in information theory or cryptography gain clearer formulations under VPL.

References

- Guerchi, L. *A Vectorial and Phasor-Based Model of Programming*, this manuscript.
- Nielsen, M. A., & Chuang, I. L. (2010). *Quantum Computation and Quantum Information*. Cambridge University Press.

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Hennessy, J. L., & Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.
- Golub, G. H., & Van Loan, C. F. (2013). *Matrix Computations*. Johns Hopkins University Press.