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

Lucio Guerchi

Abstract

This work introduces a rigorous mathematical and computational framework for representing programs as vector evolutions in finite-dimensional linear spaces. At its core is the *Vector Programming Language* (VPL), a formal language whose semantics are defined by the *Vector Evolution Machine* (VEM). In this model, both control flow and data are encoded in a unified state vector, and program execution corresponds to iterative applications of affine transformations combined with phasor-based halting conditions. This results in a mathematically transparent formulation of imperative computation.

The VEM formalism extends classical computation models by embedding algorithmic behavior directly in linear algebraic structures. Conditionals, loops, and function calls are encoded as explicit control vectors and linear operators, enabling programs to be analyzed as deterministic or convergent dynamical systems. This representation supports parallelism, symbolic reasoning, and the possibility of extending beyond the Church–Turing paradigm.

Beyond its theoretical contributions, this work also develops a complete compilation pipeline for VPL. Source programs written in C are automatically translated into VPL intermediate representations, expanded into control and data matrices, and transformed into phasor-based forms suitable for mathematical simulation. The resulting structures can be executed in a purely algebraic simulator that evolves the program state step by step, fully exposing the underlying vector dynamics.

Finally, the framework demonstrates mappings from VPL representations to hardware implementations, including both digital and analog circuits (e.g., netlist generation for simulation in NGSPICE). These applications illustrate how VPL bridges the gap between high-level programming, formal mathematical models, and physical computational architectures. The combination of rigorous theoretical foundations and practical tooling establishes VPL and VEM as a novel paradigm for computation, analysis, and implementation.

Contents

1 Introduction

Since the mid-20th century, several foundational paradigms have shaped our understanding of computation. The *Turing machine* formalism and λ-calculus, established by pioneers such as Turing and Church, gave rise to the *Church–Turing thesis*, which remains a central theoretical cornerstone of computer science. In parallel, developments in control theory and linear systems introduced mathematical tools for describing dynamical systems using matrices, state-space representations, and phasors. Later, the emergence of quantum computation reframed information processing in terms of linear operators acting on complex Hilbert spaces.

These three intellectual trajectories — symbolic computation, continuous dynamical systems, and quantum linear algebra — have historically remained largely separate. Classical programming models are rooted in discrete symbol manipulation; control theory operates in continuous state spaces; and quantum computation relies on unitary linear transformations. This separation has created a conceptual gap between program semantics and physical computation, particularly in the context of hybrid or emerging architectures.

Vector Programming Language (VPL) and its execution model, the *Vector Evolution Machine* (VEM), aim to bridge this gap. Instead of representing computation as discrete strings on tapes, VPL encodes both control and data

as structured vectors in finite-dimensional vector spaces. The evolution of program state is expressed through linear transformations, unifying symbolic execution with linear algebraic dynamics. Control flow is represented by *control vectors*, while data resides in *data vectors*, and their interaction is governed by a global transition operator. This provides a mathematically rigorous but computationally flexible foundation.

A core conceptual innovation in this work is the introduction of *phasor dynamics* into program semantics. Phasors enable the representation of oscillatory or periodic structures, such as loops and conditionals, using algebraic transformations rather than imperative constructs. This allows the execution of control structures like `while` or `for` loops to be captured as recurrent linear transformations in vector space, effectively modeling time evolution of program control states.

Parallelism emerges naturally in this framework. Because multiple control paths can coexist as linear superpositions of control vectors, the evolution of parallel or branching programs does not require explicit concurrency primitives. Instead, it is embedded in the mathematical structure of the execution model. This represents a shift from *control-driven* to *state-driven* parallelism, aligning more closely with physical processes.

The compilation pipeline for VPL illustrates the practical implications of this theoretical model:

- **Digital hardware:** Vector operators can be compiled into deterministic control structures and traditional logic circuits.
- **Analog hardware:** Phasor-based semantics can be mapped to networks of analog components, and automatically exported to SPICE netlists for simulation or fabrication.
- **Quantum hardware:** Because the core model uses linear operators, VPL programs can also be mapped into unitary transformations and quantum circuits, allowing execution on quantum computing platforms.

This dual theoretical–practical nature sets VPL and VEM apart from traditional computation models. On the theoretical side, they generalize the notion of computation as vector state evolution, potentially extending beyond the classical Church–Turing framework. On the practical side, they provide a unified intermediate representation that can target digital, analog, and quantum architectures.

The remainder of this paper develops the formal foundations of the model, presents the mathematical execution semantics, and demonstrates the framework through explicit numerical and algorithmic examples. It also outlines how this representation can serve as a common language between abstract algorithms and physical realizations, supporting new computational paradigms in hybrid analog–digital and quantum-classical architectures.

2 Formal Model

2.1 Program Representation

We consider programs written in a structured, imperative subset of the Vector Programming Language (VPL), consisting of assignments, conditional branches, and loop constructs such as `while` and `for`. Let

$$P = (S, V)$$

denote a program with:

- $S = \{s_0, s_1, \ldots, s_{n_s-1}\}$ the set of *control statements*;
- $V = \{v_0, v_1, \ldots, v_{n_v-1}\}$ the set of *data variables*.

Program execution evolves through a control pointer that moves along S, while the data vector evolves according to arithmetic or logical transformations defined by the statements. Each statement in S corresponds to a control block and each variable in V corresponds to a data register.

2.2 Control–Data Vector Space

Each program is embedded into a finite-dimensional real vector space:

$$\mathcal{H} = \mathbb{R}^{n_c} \oplus \mathbb{R}^{n_d}$$

where:

- $\mathbb{R}^{n_c}$ encodes the *control state*;
- $\mathbb{R}^{n_d}$ encodes the *data state*;
- $n_c = |S|$ or a fixed upper bound N;

- $n_d = |V|$.

An element of $\mathcal{H}$ at time step t is:

$$x(t) = \begin{bmatrix} c(t) \\ d(t) \end{bmatrix}$$

with:

- $c(t) \in \mathbb{R}^{n_c}$ a one-hot or phasorial encoding of the active control block;
- $d(t) \in \mathbb{R}^{n_d}$ the vector of data values at time t.

The initial state is:

$$x(0) = \begin{bmatrix} e_{s_0} \\ d_0 \end{bmatrix}$$

where e_{s_0} is the canonical basis vector associated with the entry statement.

2.3 Phasor Semantics of Statements

Each statement $s \in S$ is represented by a linear (or piecewise-linear) operator:

$$\Phi_s : \mathcal{H} \to \mathcal{H}$$

composed of two components:

- $T_s \in \mathbb{R}^{n_c \times n_c}$: control transition,
- $A_s \in \mathbb{R}^{n_d \times n_d}$: data transformation.

The full phasor matrix for a statement is block structured:

$$\Phi_s = \begin{bmatrix} T_s & 0 \\ B_s & A_s \end{bmatrix}$$

where B_s encodes data updates conditioned on control activation.

2.4 Program Evolution Operator

The global program evolution operator is defined as:

$$M = \sum_{s \in S} \Phi_s,$$

where the sum is taken over mutually exclusive control activations. At each discrete time step:

$$x(t+1) = Mx(t).$$

This equation defines a deterministic linear or piecewise-linear evolution in the program vector space.

2.5 Loop Structures and Cyclic Trajectories

A `while` or `for` loop is represented phasorially as:

$$\Phi_{\text{loop}} = \Phi_{\text{condition}} \cdot \Phi_{\text{body}} \cdot \Phi_{\text{backedge}},$$

where:

- $\Phi_{\text{condition}}$ evaluates the loop condition,
- Φ_{body} performs the body transformation,
- Φ_{backedge} returns control to the condition or exit block.

Loop execution corresponds to cyclic trajectories in control space, with linear or affine transformations on data space:

$$x(t+k) = \Phi_{\text{loop}}^{k}\, x(t).$$

Depending on the loop semantics, trajectories may:

- converge to a fixed point,
- form periodic cycles,
- diverge (non-terminating loops).

2.6 Fixed vs. Dynamic Dimensionality

Earlier versions of the model used a fixed control dimension $n_c = N$ (e.g., $N = 16$) to regularize the algebra:

$$M \in \mathbb{R}^{(N+n_d)\times(N+n_d)}.$$

Current implementations adopt a *dynamic dimensionality* strategy:

$$n_c = |S|,$$

resulting in a minimal matrix representation:

$$M \in \mathbb{R}^{(|S|+|V|)\times(|S|+|V|)}.$$

Both forms are semantically equivalent. The fixed-dimension matrix can be seen as an embedding of the minimal matrix:

$$\tilde{M} = \begin{bmatrix} M & 0 \\ 0 & I_{\text{pad}} \end{bmatrix}.$$

2.7 Relation to Control Graphs and State Machines

The control subspace corresponds to the state set of a finite state machine (FSM). Each Φ_s acts as a transition operator:

- activates one or more next control states,
- modifies the data vector.

This provides a linear-algebraic encoding of control flow graphs. Unlike classical FSMs, phasorial encoding supports superposition, spectral analysis, and algebraic manipulation of trajectories.

2.8 Simulation and Vector Evolution

Program execution corresponds to repeated application of the evolution operator:

$$x(t+1) = Mx(t),$$

until the control subspace reaches an absorbing state or halting condition.

The set $\{x(t)\}$ defines a discrete trajectory in $\mathcal{H}$. The simulator explicitly records:

- $c(t)$: control state vector at each step,
- $d(t)$: data vector at each step,
- $x(t)$: full program state,
- M^t: cumulative operator for spectral and structural analysis.

2.9 Phasorial Semantics and Circuit Mapping

The block structure of M naturally maps to both digital and analog circuits:

- In digital hardware, control subspace corresponds to a finite state machine, and data subspace to registers and ALUs.
- In analog hardware, phasorial components can be realized through networks of op-amps or dynamic systems that reproduce the state evolution as continuous signal flows.

This correspondence establishes a direct path from abstract program semantics to concrete circuit implementations.

2.10 Summary

The formal model embeds imperative programs into a finite-dimensional real vector space. Execution is represented by iterated application of a structured linear operator (phasor matrix). Control flow corresponds to steering basis vectors through a graph, while data flow corresponds to linear or affine transformations. This representation provides a bridge between software semantics, algebraic models, and circuit realizations.

3 Vector Evolution Machine (VEM)

The *Vector Evolution Machine* (VEM) is the abstract execution model that realizes the semantics of a VPL program through linear transformations on a structured state space. Unlike a traditional control-flow interpreter, the VEM is not based on stepwise instruction dispatching but on *vector space dynamics*: program state is represented as a point in a finite-dimensional vector space, and program execution is realized as successive applications of linear operators derived from the program's structure.

3.1 State Representation

Let $\mathcal{H}$ denote the total state space of the program, decomposed as

$$\mathcal{H} = \mathcal{H}_C \oplus \mathcal{H}_D$$

where $\mathcal{H}_C$ represents the *control subspace* (encoding active control blocks and program flow) and $\mathcal{H}_D$ represents the *data subspace* (encoding variables and intermediate values). Each program point (control block) is mapped to a unique basis vector in $\mathcal{H}_C$, while each scalar or structured data variable corresponds to a dimension in $\mathcal{H}_D$.

A full program configuration at time t is represented as a state vector

$$X(t) \in \mathcal{H},$$

with

$$\dim(\mathcal{H}) = n_C + n_D,$$

where n_C and n_D are the number of control and data dimensions, respectively.

3.2 Phasorial Transition Model

Each statement or basic block of the program is compiled into a linear transition operator acting on $\mathcal{H}$. More specifically, the entire program induces a global transition operator

$$M \in \mathbb{C}^{(n_C+n_D)\times(n_C+n_D)},$$

such that

$$X(t+1) = MX(t) + c,$$

where c is a constant vector representing affine offsets (e.g., initializations or increments). This representation naturally extends classical linear models by admitting *phasorial* control: transitions between control blocks can be weighted by complex phasors, representing oscillatory or rotational evolution in the control subspace. This property allows the VEM to encode both conventional deterministic control flow and continuous or periodic evolution.

3.3 Execution Semantics

Program execution corresponds to iterated application of the transition operator:

$$X(t) = M^t X(0) + \sum_{k=0}^{t-1} M^k c.$$

The initial state $X(0)$ encodes the entry control block and initial data variable values. Loops are modeled not as explicit control constructs but as eigenstructure in the operator M: a loop corresponds to a non-trivial invariant subspace or cyclic orbit under repeated application of M.

Branching is represented by superposition or projection of control phasors; in the simplest deterministic case, transitions are modeled as 0–1 matrices, but probabilistic or phasorial weighting can also be applied, allowing smooth interpolation between paths. This allows the VEM to simulate control-flow graphs as linear dynamical systems.

3.4 Halting and Observation

A program halts when the system reaches a subspace $\mathcal{H}_{\text{halt}} \subseteq \mathcal{H}_C$ corresponding to terminal control blocks. Let P_{halt} denote the orthogonal projector onto $\mathcal{H}_{\text{halt}}$. The halting condition is

$$P_{\text{halt}} X(t) \neq 0.$$

Upon halting, the final values of data variables are read from the projection of $X(t)$ onto $\mathcal{H}_D$.

3.5 Relation to Classical and Analog Execution Models

The VEM generalizes both classical control-flow execution and analog signal evolution. When M is a Boolean permutation matrix, the model reduces to a deterministic control-flow machine. When M contains real-valued or phasorial coefficients, the model corresponds to an analog computational system, suitable for continuous-time approximation or physical realizations in electronic or mechanical systems.

3.6 Quantum Specialization

The same vector evolution framework can also be specialized for quantum circuits by imposing unitarity on M. Specifically, if $M = U$ and $U^\dagger U = I$, and if $X(0)$ is a normalized vector in a complex Hilbert space, then the VEM semantics correspond to unitary evolution:

$$|\psi(t)\rangle = U^t|\psi(0)\rangle.$$

Measurement corresponds to projection onto subspaces, analogous to halting conditions. This specialization allows VPL programs to be interpreted as quantum circuits under appropriate constraints.

3.7 Advantages of the VEM Approach

- **Unified execution model:** The same formalism captures digital, analog, and quantum execution semantics.
- **Deterministic or phasorial control:** Phasorial transitions allow modeling of cyclic and oscillatory behaviors naturally.
- **Linear algebraic structure:** Execution is fully described by matrix algebra, enabling symbolic analysis, compilation, and simulation.
- **Hardware mapping:** The linear structure allows systematic translation into both digital circuits (as state machines) and analog circuits (as differential or phasor networks).
- **Composability:** Control flow composition corresponds to block structure and matrix composition, allowing modular compilation.

The Vector Evolution Machine thus provides the foundational computational model for VPL, supporting both efficient simulation and hardware realization, and offering a natural bridge between discrete computation, analog dynamics, and quantum evolution.

3.8 Comparison with Turing Machine

The Turing machine, introduced by Alan Turing in 1936, is a formal abstraction of computation based on a discrete tape, a finite set of states, and

a transition function that determines the next action based on the current state and symbol. It serves as the foundation of the Church–Turing thesis, which states that any function that can be effectively computed can be computed by a Turing machine. This model has been central to computer science, complexity theory, and the formal study of algorithms.

In contrast, the Vector Evolution Machine (VEM) represents computation through linear algebraic structures acting on combined control–data vector spaces. Instead of a tape and head, the system evolves a state vector $\mathbf{X}$ according to an evolution operator $\mathbf{M}$ and a control structure embedded in phasorial and algebraic transformations:

$$\mathbf{X}(t+1) = \mathbf{M}\mathbf{X}(t) + \mathbf{c}. \tag{1}$$

This provides a fundamentally different view of computation. Where the Turing machine advances one symbolic step at a time, the VEM applies global transformations to a vector space at each iteration. Loops, conditionals, and control flow are encoded algebraically into the structure of $\mathbf{M}$, while halting conditions are represented as constraints, often through phasorial projections or vector subspace selections.

Structural Comparison. The main structural differences can be summarized as follows:

- **Representation of State.** A Turing machine stores its state in a finite control state and a tape with a head position. A VEM represents the entire program and data state as a vector $\mathbf{X} \in \mathbb{R}^n$ or $\mathbb{C}^n$, where n is determined by the number of control blocks and data variables.

- **Transition Mechanism.** A Turing machine uses a discrete transition function $\delta(q, s) = (q', s', d)$. A VEM uses a linear operator $\mathbf{M}$ and a bias vector $\mathbf{c}$ to evolve the state, corresponding to $\mathbf{X}(t+1) = \mathbf{M}\mathbf{X}(t) + \mathbf{c}$.

- **Control Flow.** In a Turing machine, control flow is sequential and determined by the current symbol and state. In a VEM, control flow is encoded as subspace transformations and phasorial decision functions that can be applied in parallel or superposition.

- **Halting.** Turing machines halt upon reaching a designated halting state. VEMs halt when the evolving state vector reaches a subspace

satisfying halting constraints, typically expressed as

$$\mathbf{C}_p\mathbf{X} = \mathbf{c}_p,$$

or through phasorial phase-locking conditions in the control subspace.

Computational Power and Extensions. Every Turing machine can be encoded as a VEM by mapping tape symbols and states into a vector basis and encoding the transition function into the matrix $\mathbf{M}$. This demonstrates that VEMs are at least Turing-complete.

However, the VEM formalism is not limited to discrete, symbolic computation. Because $\mathbf{M}$ and $\mathbf{X}$ may be continuous or complex-valued, the model also accommodates:

- **Analog computation**, where continuous time and real numbers are used.
- **Quantum computation**, by restricting $\mathbf{M}$ to unitary operators and incorporating superposition and interference effects.
- **Phasorial control**, which embeds halting and control decisions in the geometry of the state space.

This broader representational capacity suggests that VEM extends beyond the traditional Church–Turing paradigm under certain assumptions (e.g., infinite precision or physical realizability of continuous operators). This connects naturally to models such as the Blum–Shub–Smale machine and continuous-time quantum computation.

Summary. While the Turing machine provides a minimal symbolic model of computation, the VEM framework describes computation as vectorial state evolution under linear or unitary transformations, with control embedded in algebraic and geometric structures. Turing machines are a subset of this broader class. As such, VEM provides both a theoretical generalization and a practical bridge between digital, analog, and quantum models of computation.

Aspect	Turing Machine	Vector Evolution Machine
State	Tape + head + control state	Vector $\mathbf{X}$
Transition	Discrete δ function	Linear/phasorial evolution $\mathbf{M}$
Control	Symbolic stepwise	Geometric / subspace driven
Halting	Designated halting state	Constraint subspace / phasor locking
Precision	Discrete	Discrete or continuous
Computational model	Symbolic	Symbolic, analog, or quantum

Table 1: Comparison between Turing machine and Vector Evolution Machine.

3.8.1 Encoding a Turing Machine as a VEM

We give a direct constructive encoding showing how a deterministic Turing machine can be represented as a Vector Evolution Machine. This demonstrates that VEMs subsume classical Turing computation.

Deterministic Turing machine (standard definition). A deterministic Turing machine (TM) is a tuple

$$\mathcal{T} = (Q, \Gamma, b, \Sigma, \delta, q_0, q_H),$$

where:

- Q is the finite set of control states (including initial q_0 and halting q_H),
- Γ is the tape alphabet (including blank symbol b),
- $\Sigma \subseteq \Gamma$ is the input alphabet,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the (deterministic) transition function,
- the tape is bi-infinite with a single read/write head.

A TM configuration can be viewed as the triple (q, τ, h) where $q \in Q$ is the current state, $\tau : \mathbb{Z} \to \Gamma$ is the tape content, and $h \in \mathbb{Z}$ is the head position.

VEM state vector design. We construct a finite-dimensional VEM state vector that encodes a *bounded* portion of the TM tape (sufficient for the simulated computation), the head position within that window, and the control state. Choose a tape window of length W (indices $0, \ldots, W-1$); for formal completeness one can consider an arbitrarily large but finite window (or an encoding that grows dynamically).

Define the VEM state vector X by concatenating three sub-blocks:

$$X = \begin{bmatrix} C \\ H \\ T \end{bmatrix},$$

with

1. $C \in \mathbb{R}^{|Q|}$ — *control subspace.* Canonically, control state q is represented by the basis vector e_q (one-hot).

2. $H \in \mathbb{R}^{W}$ — *head position one-hot vector.* If the head is at tape index i (inside the window) then $H = e_i$.

3. $T \in \mathbb{R}^{W \cdot |\Gamma|}$ — *tape cell encoding.* Each tape cell i is represented by a block of $|\Gamma|$ coordinates, one-hot encoding the symbol at that cell. Thus the total data dimension for the tape is $W \cdot |\Gamma|$.

Consequently the full VEM dimension is

$$n = |Q| + W + W \cdot |\Gamma|.$$

Transition operator M. We build M so that one application simulates exactly one TM transition step. Let $\delta(q, a) = (q', a', d)$, with $d \in \{L, R\}$. The matrix M is block-sparse and implements the following actions for every possible combinational input:

- **Control update:** when control block e_q and head position e_i are active and tape cell i holds symbol a, the next control subvector should be $e_{q'}$. This is achieved by setting appropriate rows of the control block of M so that

  $$C_{\text{next}} \leftarrow \text{selector}(q',\, q, i, a).$$

  Operationally, this is done by adding contributions to the rows of M that map the current active triple (q, i, a) into $e_{q'}$.

- **Tape write:** the symbol register of cell i must change from the coordinate for a to the coordinate for a'. This is effected by letting the rows of M corresponding to the symbol coordinates of cell i depend on the conjunction of $C = e_q$ and $H = e_i$ and on the symbol a.
- **Head move:** the head one-hot vector H must shift left or right. If $d = R$ we want $H_{\text{next}} = \text{shift\_right}(H)$. This corresponds to setting subdiagonal/ superdiagonal 1s in the H-to-H block of M.

Concretely, M has the following block structure (rows grouped by output subvector, columns by input subvector):

$$M = \begin{array}{c|c|c|c} & C & H & T \\ \hline C & M_{CC} & M_{CH} & M_{CT} \\ \hline H & M_{HC} & M_{HH} & M_{HT} \\ \hline T & M_{TC} & M_{TH} & M_{TT} \end{array}$$

where:

- M_{CC} can be zero (except for wiring of non-changing control states); most of control update logic is encoded jointly via M_{CT} and M_{CH} so the control next-state is selected only under the correct conjunction of symbol and head-location.
- M_{HH} implements the deterministic shift of the head one-hot vector (a permutation matrix effecting $i \mapsto i \pm 1$ where valid).
- M_{TT} implements persistence of tape cells (identity) for cells not written on the current step; writes at the head are implemented by rows in M_{TC} and M_{TH} that replace the one-hot symbol block for cell i with the new symbol a' when the conjunction (control q, head i) holds.

Implementation detail (selector logic via linear algebra). The conjunction "control $= q$ AND head $= i$ AND tape-cell $i = a$" can be realized by choosing a linear encoding where the involved one-hot coordinates are multiplied via indicator wiring: since VEM evolution is linear, one implements these conjunctions by using *expanded control coordinates* (i.e., include coordinates that explicitly represent the Cartesian product of relevant discrete components), or by encoding the selector as a sparse set of matrix rows that pick the unique basis vector corresponding to the active triple. For example:

- Introduce auxiliary product coordinates (optional): for each (q, i, a) create a basis coordinate $\pi_{q,i,a}$ set to 1 exactly when $C = e_q$, $H = e_i$, and tape-cell i has symbol a. This increases dimensionality but makes selection linear (the next-state rows are simply direct mappings of $\pi_{q,i,a}$ to the appropriate outputs).
- Alternatively, directly program the sparse rows of M so that the single nonzero column entries correspond to the active triple's basis indices; because the active triple is represented by a single 1 across the combined coordinates, the matrix rows map that column to the appropriate next-state coordinates.

Halting. The TM halting state q_H corresponds to the control basis vector e_{q_H}. The VEM halting projector P_{halt} is simply the orthogonal projector onto the subspace spanned by e_{q_H}. Execution halts when the projection is nonzero.

Worked tiny example. Consider a trivial TM with $Q = \{q_0, q_H\}$, $\Gamma = \{b, 1\}$, and $W = 3$ (tape window indices $0, 1, 2$). Then:

- C has dimension 2, H dimension 3, T dimension $3 \cdot 2 = 6$. Total $n = 11$.
- If $\delta(q_0, 1) = (q_H, 1, R)$, then the rows of M that set the next control vector must map the (unique) column representing the current active coordinates (control index of q_0, head index i, and the corresponding symbol coordinate within the tape-block for cell i) to the basis vector e_{q_H}.
- The head-shift block M_{HH} is the permutation matrix that sends $e_0 \mapsto e_1$, $e_1 \mapsto e_2$, etc.

Sparsity and practical considerations. The matrix M constructed by this encoding is extremely sparse: each transition $\delta(q, a)$ affects only the rows associated with:

$$\text{control}, \quad \text{head positions}, \quad \text{tape cell } i.$$

However, the *dimension* grows linearly with the tape window W and multiplicatively with $|\Gamma|$. For an unbounded tape this requires an unbounded-dimensional VEM; practically one either chooses a sufficiently large finite

window or uses a dynamic allocation strategy (growing W as needed) or encodes the tape symbolically into a compact representation (e.g., run-length encoding) and simulates that within VEM.

Correctness. By construction, an application of M to the VEM vector that encodes a TM configuration (q, τ, h) yields the VEM vector encoding the next TM configuration (q', τ', h'). Repeating this t times simulates t TM steps. Therefore deterministic TM execution is faithfully simulated by the VEM.

Remarks.

- The encoding shows that *Turing computability* is contained in the VEM model: any TM-computable function can be realized by an appropriate VEM matrix M (with $c = 0$ for purely permutation-like transitions).
- The price paid is dimensionality: the naive encoding yields a vector space whose dimension grows with tape window and alphabet size. Practical VEM toolchains therefore use optimizations (sparse representations, auxiliary coordinates, dynamic window growth) to manage state-space size.
- This construction also clarifies fixed-size n×n models are discussed and can be viewed as *padded embeddings* of such minimal encodings.

4 Control Vectors

Control vectors form a fundamental component of the Vector Evolution Machine (VEM) architecture. They encode the *control state* of the program within a dedicated subspace of the global vector space, making control flow an intrinsic part of the mathematical structure of computation rather than an external mechanism.

4.1 Definition and Representation

Let $\mathcal{C} = \{c_0, c_1, \ldots, c_{n_c-1}\}$ be the set of all control states corresponding to program blocks such as instructions, basic blocks, or loop headers. The

control subspace is defined as

$$\mathcal{H}_C = \mathbb{R}^{n_c}$$

for classical systems, or more generally

$$\mathcal{H}_C = \mathbb{C}^{n_c}$$

when phasor or quantum control is employed. The canonical basis $\{e_{c_i}\}$ corresponds to individual control locations. At each step t, the control vector $C^{(t)} \in \mathcal{H}_C$ encodes the active control state or superposition of control states.

4.2 Global State Structure

The complete program state is represented by the concatenation of control and data subvectors:

$$X^{(t)} = \begin{bmatrix} C^{(t)} \\ D^{(t)} \end{bmatrix},$$

where $C^{(t)}$ represents control flow and $D^{(t)}$ represents data flow. Both components evolve according to a unified linear operator:

$$X^{(t+1)} = MX^{(t)} + c,$$

where

$$M = \begin{bmatrix} M_{CC} & M_{CD} \\ M_{DC} & M_{DD} \end{bmatrix}.$$

Here:

- M_{CC} encodes transitions between control states,
- M_{CD} models control flow affected by data values (e.g., conditional branches),
- M_{DC} models data updates determined by control flow,
- M_{DD} encodes intrinsic data transformations.

4.3 Branching and Looping Semantics

Control vectors provide a direct algebraic representation of control flow constructs. For example:

- **Sequencing:** control transitions are represented by permutation matrices acting on $\mathcal{H}_C$.
- **Branching:** conditional branches are implemented through control transitions modulated by M_{CD}.
- **Looping:** loops correspond to cycles in the control graph encoded within M_{CC}.

In deterministic classical programs, $C^{(t)}$ is a one-hot vector. More generally, phasor or quantum control allows superpositions or continuous mixtures of control states.

4.4 Phasor and Quantum Extensions

Phasor-based control introduces phase and amplitude into control vectors:

$$C^{(t)} = \sum_i a_i e^{j\phi_i} e_{c_i},$$

where a_i and ϕ_i denote amplitude and phase associated with control location c_i. This enables:

- **Phase-dependent control flow** (useful in analog or oscillator-based computation),
- **Coherent superpositions** of control states, enabling hybrid classical-quantum computation,
- **Temporal encoding** of control evolution through phasor rotations.

In the quantum case, control vectors inhabit a Hilbert space and evolve under unitary transformations, preserving norm and coherence.

4.5 Halting States and Control Invariants

A designated absorbing state e_{halt} represents the halting configuration:

$$M_{CC} e_{\text{halt}} = e_{\text{halt}}.$$

Once the halting state is reached, the control vector remains invariant, ensuring well-defined termination in both classical and extended control semantics.

4.6 Role in the VEM Model

Unlike traditional control flow mechanisms in Turing machines or von Neumann architectures, control vectors are embedded in the same linear space as data. This provides:

- A unified algebraic semantics for both control and data,
- Natural extension to parallel, phasor, and quantum computation,
- A direct bridge between abstract computation and physical circuit models (digital or analog).

4.7 Parallel Execution Semantics

A key property of the control vector formalism is its intrinsic ability to encode multiple active control states simultaneously. This enables a natural and algebraically well-defined model of parallel execution.

4.7.1 Superposed Control States

Let

$$C^{(t)} = \sum_{i \in \mathcal{A}(t)} a_i e_{c_i}$$

denote the control vector at time t, where $\mathcal{A}(t)$ is the set of active control states and a_i are the associated coefficients. Unlike classical one-hot control, this representation allows several control points to be active concurrently.

- In the **classical parallel** case, the coefficients a_i may represent weights or relative activation strengths.

- In the **phasor** case, a_i include amplitude and phase, allowing phase-dependent coordination between parallel control flows.
- In the **quantum** case, a_i are complex probability amplitudes, evolving unitarily.

4.7.2 Evolution of Parallel Control

Parallel control states evolve simultaneously under the global transition operator:

$$C^{(t+1)} = M_{CC}C^{(t)} + M_{CD}D^{(t)}.$$

Each active control state contributes to subsequent transitions, enabling parallel block activations without explicit instruction-level synchronization.

This algebraic approach generalizes:

- **Multithreading** in digital computation,
- **Superposed control paths** in quantum circuits,
- **Continuous-time parallel flows** in analog/phasor systems.

4.7.3 Synchronization and Merging

Control vector parallelism also naturally encodes synchronization:

- When multiple active states map to a single successor block, the resulting control vector components sum linearly.
- Conditional synchronization can be expressed through structured transition submatrices in M_{CC} and M_{CD}.
- Halting or joining points correspond to convergence of control amplitudes.

4.7.4 Hardware Implications

In a hardware context:

- Parallel control supports *dataflow-like execution* without centralized instruction sequencing.

- Phasor parallelism corresponds to continuous signal superpositions, allowing analog parallel circuits.
- Quantum parallelism corresponds to true physical superposition of control states.

By treating control flow algebraically, the VEM model unifies sequential, parallel, and superposed execution under the same mathematical structure. Parallel execution does not require special semantics or scheduling; it emerges directly from the evolution of the control vector and the structure of the global operator M.

4.8 Summary

Control vectors elevate control flow to a first-class, algebraically encoded entity. They unify branching, looping, and halting in a single linear transformation framework. By extending their representation from real to complex spaces, they also provide a mathematical substrate for analog and quantum extensions of the Vector Evolution Machine, surpassing the limitations of classical state machines.

5 Functions and Function Calls

5.1 Overview

In the original formulation of the VPL execution model, functions were treated primarily as syntactic control flow elements: named blocks of instructions, parameterized by input arguments, and entered through a `call` instruction and exited through a `return`. This view is sufficient for imperative execution but does not capture the mathematical structure of functions when expressed as transformations in vector space.

In the updated Vector Evolution Machine (VEM) model, functions are formalized as *linear or affine operators* acting on the joint data and control vector spaces. This approach provides a precise algebraic semantics for function invocation, composition, and return, and supports advanced behaviors such as recursion, higher-order functions, and parallel execution.

5.2 Functions as Linear Operators

Let $\mathcal{H}_D$ denote the data vector space and $\mathcal{H}_C$ the control vector space. A function f is represented as an operator

$$F_f : \mathcal{H}_C \otimes \mathcal{H}_D \to \mathcal{H}_C \otimes \mathcal{H}_D,$$

which acts on the joint control-data state

$$\mathbf{x} = \mathbf{c} \otimes \mathbf{d},$$

where $\mathbf{c} \in \mathcal{H}_C$ encodes the control phasor and $\mathbf{d} \in \mathcal{H}_D$ encodes the current data state.

The function call corresponds to activating a control basis vector associated with the entry point of f, applying F_f to the joint state, and then collapsing the control phasor back to the caller's continuation point at return.

5.3 Entry, Exit and Control Encoding

For each function f, we define:

- $\mathbf{e}_f^{\text{in}} \in \mathcal{H}_C$: control basis vector representing entry to f.
- $\mathbf{e}_f^{\text{out}} \in \mathcal{H}_C$: control basis vector representing return from f.
- F_f: linear or affine map implementing the function body.

A function call is modeled as

$$\mathbf{x}_{\text{out}} = F_f(\mathbf{e}_f^{\text{in}} \otimes \mathbf{d}_{\text{in}}),$$

with

$$\mathbf{x}_{\text{out}} = \mathbf{e}_f^{\text{out}} \otimes \mathbf{d}_{\text{out}}.$$

This corresponds to the execution of the function's internal statements and the update of both data and control vectors.

5.4 Composition of Functions

Let f and g be two functions. Their sequential composition corresponds to operator multiplication:

$$F_{g \circ f} = F_g \cdot F_f.$$

If f and g act on disjoint subspaces, parallel execution can be modeled via the tensor product:

$$F_{f \| g} = F_f \otimes F_g.$$

This provides a clean mathematical foundation for modular and parallel computation.

5.5 Recursion and Fixed Points

Recursive functions can be naturally expressed as fixed points of operator compositions. Let F_f represent a recursive function. Then

$$\mathbf{x}^* = F_f(\mathbf{x}^*)$$

defines a fixed point in $\mathcal{H}_C \otimes \mathcal{H}_D$, corresponding to the stabilized result of a recursive evaluation. Existence and uniqueness conditions for fixed points depend on the properties of F_f (e.g., contractive mappings or finite control).

5.6 Function Parameters and Closures

Function parameters are modeled as additional coordinates in $\mathcal{H}_D$ or as affine offsets applied before evaluation. Closures and higher-order functions are represented as operators that return operators, extending the expressive power of VEM beyond simple procedural execution.

5.7 Integration with the Compiler

The VPL compiler stages handle functions as follows:

Extractor: Identifies function declarations and call sites.

Translator: Maps entry and exit labels to control vectors.

Expander: Expands function bodies as subgraphs in the operator composition tree.

Phasor Transformer: Generates M_f and $\mathbf{c}_f$ for each function block.

Simulator: Composes operators during simulation, supporting recursion and parallelism.

5.8 Parallel and Higher-Order Execution

Functions can be activated simultaneously by assigning non-orthogonal control phasors, allowing multiple active execution threads in the same vector space. This supports parallel execution models and provides a pathway to physical realizations on analog and quantum substrates.

5.9 Summary

By recasting functions as linear or affine operators on the joint control-data space, the VEM formalism provides:

- Precise mathematical semantics for function calls and returns.
- Natural support for composition, recursion, and parallelism.
- A unified representation compatible with both digital and quantum implementations.

This represents a substantial generalization beyond the traditional control flow semantics of imperative languages and positions VPL for advanced computational paradigms.

6 String as Vector

In the VPL / VEM framework, every program element originally expressed as a string in the source language is systematically mapped to a structured vectorial representation. Strings such as variable names, block labels, and conditional expressions are not merely symbolic identifiers; rather, they form the *basis of the program's vector space*.

Let $\mathcal{S}$ denote the set of all strings extracted from the program. Each $s \in \mathcal{S}$ is associated with a canonical basis vector through a bijection

$$\phi : \mathcal{S} \longrightarrow \mathbb{R}^n,$$

where $n = |\mathcal{S}|$ is the total number of unique program strings. This mapping induces two main subspaces:

$$\mathbb{R}^n = \mathcal{C} \oplus \mathcal{D},$$

where $\mathcal{C}$ is the *control subspace* spanned by vectors corresponding to control labels (e.g., blocks, loops, and branch points), and $\mathcal{D}$ is the *data subspace* spanned by vectors associated with variables and symbolic constants.

6.1 Phasorial Interpretation

Conditional expressions are also represented as strings in the original program but are transformed into phasor constraints during the compilation process. This transformation is formalized as

$$\psi : \text{cond\_str} \longrightarrow (C_p, c_p),$$

where C_p and c_p define a linear or affine constraint on the state vector. For example, the condition

```
x < 5
```

is mapped to a hyperplane or region of the state space that constrains the evolution of the control vector associated with the loop block.

The phasorial representation enables conditional branching and loop termination to be handled as *vector space transitions*, rather than as textual comparisons. The evolution of program state is then determined by applying linear transformations and affine offsets derived from the original textual program.

6.2 Role in the Execution Model

This mapping from *strings* to *vectors* and then to *phasorial constraints* is a central step in the VPL compilation and execution pipeline. It enables:

- A fully algebraic execution model for imperative programs.
- Unified representation of control and data within the same vector space.
- Deterministic simulation through linear and affine transformations.
- A natural path toward hardware realization, as both digital and analog systems operate on numeric vectors rather than textual symbols.

6.3 Implications and Extensions

This representation establishes a direct algebraic semantics for program text, in which *the entire program is embedded into a structured vector space*. Program execution corresponds to

$$\mathbf{x}_{t+1} = M\mathbf{x}_t + \mathbf{c},$$

where M and $\mathbf{c}$ are derived systematically from the mapped strings and their associated phasorial constraints. This formulation is compatible with:

- Classical deterministic execution (digital hardware),
- Analog computation through continuous state evolution,
- Quantum extensions through unitary transformations on subspaces.

By treating strings as fundamental generators of the vector space, the VPL / VEM formalism bridges the gap between textual programming languages and algebraic machine models, enabling precise mathematical reasoning, simulation, and eventual hardware implementation.

7 Data Structures in the VPL Formalism

In the Vector Programming Language (VPL), data structures are not treated as abstract containers external to the computational model. Instead, they are embedded directly in the *global program vector space*, allowing their structure and behavior to be represented through algebraic transformations. This unified treatment enables both symbolic reasoning and direct mapping to physical computation substrates, such as digital hardware or analog/quantum implementations.

7.1 Data Structures as Vector Subspaces

Let the global program state at time t be represented by a vector

$$\mathbf{v}_t \in \mathbb{R}^n,$$

where n is the total dimensionality of the program. Each data structure occupies a dedicated subspace of $\mathbb{R}^n$:

$$\mathbf{v}_t = \bigoplus_{i=1}^{k} \mathbf{d}_t^{(i)} \oplus \mathbf{c}_t,$$

where $\mathbf{d}_t^{(i)}$ are data subspaces (arrays, stacks, graphs, etc.) and $\mathbf{c}_t$ encodes the control state.

Unlike classical imperative semantics, there is no separate *memory model*: the structure of the data itself is *isomorphic to a vector structure*, and its evolution is captured by linear or affine transformations.

7.2 Array and Indexed Structures

An array of length m corresponds to a vector subspace

$$\mathbf{a}_t \in \mathbb{R}^m,$$

where each element is mapped to a fixed coordinate. An assignment

$$a[i] \leftarrow x$$

is represented as

$$\mathbf{a}_{t+1} = U_i \mathbf{a}_t + \mathbf{e}_i x,$$

where U_i is an update matrix with identity everywhere except row i, and $\mathbf{e}_i$ is the canonical basis vector selecting element i.

This formulation allows indexed operations to be represented by simple sparse matrices, enabling parallel and symbolic reasoning.

7.3 Dynamic Structures: Stacks, Queues, and Graphs

Dynamic data structures are modeled through structured transformations:

- **Stack (push)**:

  $$\mathbf{s}_{t+1} = P_{\text{push}} \mathbf{s}_t + \mathbf{c}_{\text{push}},$$

  where P_{push} is a shift matrix and $\mathbf{c}_{\text{push}}$ injects the new element.

- **Stack (pop)**:
$$\mathbf{s}_{t+1} = P_{\text{pop}}\mathbf{s}_t,$$
where P_{pop} shifts elements downward and clears the top entry.

- **Queue and FIFO** structures can be implemented analogously with a fixed rotation matrix.

- **Graphs** are represented as adjacency subspaces, with transformations corresponding to traversals or edge activations.

7.4 Constraints and Phasorial Semantics

Data structure invariants such as array bounds, stack depth, or pointer validity are encoded through *phasor constraints*. These constraints live in a dual space to the state vector, typically represented as

$$\mathbf{C}_p\mathbf{v}_t + \mathbf{c}_p \geq 0,$$

where $\mathbf{C}_p$ and $\mathbf{c}_p$ define the geometric region of valid states.

This enables the data structure semantics to be enforced and reasoned about algebraically, providing a foundation for optimization, verification, and physical mapping.

7.5 Parallel Execution on Data Structures

Because data structures are subspaces of the global program vector, multiple operations can be encoded as block-diagonal or block-sparse transformations. This makes:

- element-wise parallel updates,

- simultaneous access,

- distributed computation

natural consequences of the algebraic formulation, without requiring explicit threading or concurrency primitives.

7.6 Implications for Hardware Realization

The vector embedding of data structures allows a straightforward mapping to hardware:

- **Digital:** arrays map to register banks or memory blocks, stacks to shift-register chains, and graph structures to routing matrices.
- **Analog:** data values correspond to continuous signals; structural invariants are enforced by circuit constraints.
- **Quantum:** structures can be embedded as tensor product subspaces, with control vectors mapped to unitary or projective operations.

This tight coupling between structure and vector space enables both software-level reasoning and hardware-level synthesis, offering a uniform foundation for classical, analog, and quantum computation models.

8 Parallel Execution: Algorithmic Examples

8.1 Phasorial and Vectorial Parallelism

In the classical imperative paradigm, parallel execution requires explicit constructs such as threads, tasks, or processes. In the VPL formalism, parallelism is a direct consequence of the structure of the program's state vector and its evolution operator.

Let

$$\mathbf{v}_t \in \mathbb{R}^n$$

denote the full state of the program at time t. If the transition matrix $\mathbf{M}$ has a block diagonal or sparse structure,

$$\mathbf{M} = \begin{bmatrix} M_1 & 0 \\ 0 & M_2 \end{bmatrix},$$

then the evolution

$$\mathbf{v}_{t+1} = \mathbf{M}\mathbf{v}_t + \mathbf{c}$$

naturally decomposes into two (or more) independent sub-evolutions. This is not parallelism by scheduling, but **parallelism by structure**: independent subspaces evolve simultaneously.

8.2 Control Superposition and Phasorial Flows

Parallel control flow is achieved through superpositions in the control vector. If multiple control states are active simultaneously, their influence propagates through $\mathbf{M}$ as concurrent evolutions. For example, if

$$\mathbf{v}_t^{\text{ctrl}} = \alpha \mathbf{e}_i + \beta \mathbf{e}_j,$$

then the next state is the linear combination of the evolutions from both control states:

$$\mathbf{v}_{t+1} = \alpha \mathbf{M}_i \mathbf{v}_t + \beta \mathbf{M}_j \mathbf{v}_t.$$

This enables parallelism without explicit task creation.

8.3 Algorithmic Example: Parallel Accumulation

Consider the following VPL pseudocode:

```
x = 0
y = 0
while (x < 10) { x = x + 1 }
while (y < 10) { y = y + 1 }
```

In an imperative language, this might run sequentially or require explicit parallelization. In VPL, both loops become two **independent subspaces** of the state vector:

$$\mathbf{v}_t = \begin{bmatrix} \mathbf{v}_t^{(x)} \\ \mathbf{v}_t^{(y)} \end{bmatrix}.$$

The matrix $\mathbf{M}$ is block diagonal:

$$\mathbf{M} = \begin{bmatrix} M_x & 0 \\ 0 & M_y \end{bmatrix}.$$

Hence the evolution of x and y proceeds **in parallel** without explicit coordination. The phasorial halting condition for each loop acts independently.

8.4 Phasorial Control of Multiple Loops

The phasorial representation allows each loop to have its own

$$\mathbf{C}_p^{(i)},\ \mathbf{c}_p^{(i)}$$

for halting. If multiple phasorial constraints are active, the global stop condition is

$$\bigwedge_i \left(\mathbf{C}_p^{(i)}\mathbf{v}_t + \mathbf{c}_p^{(i)} \geq 0\right),$$

enabling concurrent, constraint-driven termination.

8.5 Dataflow vs Controlflow Parallelism

Two main patterns emerge naturally:

- **Dataflow parallelism:** Independent variable subspaces evolve simultaneously under separate control.
- **Controlflow parallelism:** Multiple control states coexist in superposition, driving different parts of the computation.

Both patterns are subsumed under the same vectorial framework.

8.6 Example: Parallel Sum and Product

Consider:

```
s = 0
p = 1
for (i=0; i<N; i++) { s += A[i] }
for (j=0; j<N; j++) { p *= B[j] }
```

These two loops correspond to two disjoint control subspaces. Under VPL, the sum and product can be executed in parallel. The matrix structure is:

$$\mathbf{M} = \text{diag}(M_s, M_p),$$

$$\mathbf{v}_{t+1} = \mathbf{M}\mathbf{v}_t + \mathbf{c}.$$

The final output is obtained when both phasorial constraints are satisfied.

8.7 Advantages of the VPL Parallelism Model

- No explicit thread or process management.
- Parallelism is inferred directly from the structure of $\mathbf{M}$ and $\mathbf{C}_p$.
- Compatible with digital, analog, and quantum realizations.
- Natural fit for hardware mapping and systolic/analog architectures.

8.8 Implications for Compilation and Synthesis

The compiler can detect **independent subspaces** during the `phasor_transformer` stage. This enables:

- Automatic parallelization of loops and functions.
- Static scheduling from structural decomposition.
- Direct mapping to parallel hardware circuits.

9 Input and Output in the VPL Formalism

9.1 I/O as Boundary Conditions

In classical models of computation, input and output are defined as discrete operations acting on an external tape or stream. In contrast, the Vector Programming Language (VPL) embeds I/O directly in the global program vector space.

Let

$$\mathbf{v}_t \in \mathbb{R}^n$$

denote the state vector of the program at time t. We partition $\mathbf{v}_t$ into three disjoint subspaces:

$$\mathbf{v}_t = \mathbf{v}_t^{\text{in}} \oplus \mathbf{v}_t^{\text{comp}} \oplus \mathbf{v}_t^{\text{out}},$$

where:

- $\mathbf{v}_t^{\text{in}}$ represents the input space (boundary-controlled),
- $\mathbf{v}_t^{\text{comp}}$ represents the computational core (evolution determined by M and c),
- $\mathbf{v}_t^{\text{out}}$ represents the output space (observable projection).

Input vectors are imposed as boundary conditions at each step:

$$\mathbf{v}_t^{\text{in}} = \mathbf{I}_t,$$

where $\mathbf{I}_t$ is determined by external environment or sensors.

9.2 Linear Projection for Output

Output is realized through a projection matrix

$$\mathbf{y}_t = P_{\text{out}} \mathbf{v}_t,$$

where $\mathbf{y}_t$ is the observed output vector and P_{out} is a linear map selecting (or transforming) the relevant subspace. This formulation allows:

- Continuous or discrete outputs,
- Multiple output channels,
- Analog or digital signal interfacing,
- Quantum measurement operators when P_{out} is projective.

9.3 I/O as Phasorial Interactions

Within the phasorial formulation, input and output can be treated as

$$\mathbf{C}_{\text{in}} \mathbf{v}_t + \mathbf{c}_{\text{in}} = \mathbf{b}_{\text{env}},$$

$$\mathbf{C}_{\text{out}} \mathbf{v}_t + \mathbf{c}_{\text{out}} = \mathbf{y}_t.$$

This turns I/O into a set of algebraic constraints coupling the program vector to its external environment. These constraints are not "executed" but *resolved* as part of the vector evolution.

9.4 Event-Driven vs Continuous Input

Depending on the physical implementation, $\mathbf{I}_t$ can be:

- **Event-driven (discrete):** suitable for classical digital systems, where inputs change at discrete steps.
- **Continuous-time:** suitable for analog or hybrid implementations.
- **Quantum state preparation:** where $\mathbf{I}_t$ corresponds to the initial or intermediate preparation of a subspace.

9.5 Hardware Realization

In hardware terms, input and output subspaces correspond to:

- Digital input/output pins or buses,
- Analog voltage/current ports,
- Quantum I/O ports or measurement operators.

The unification of I/O and computation in the same vector space enables co-simulation and synthesis: I/O becomes a geometric part of the system, not an external procedural construct.

9.6 Compositionality

I/O boundaries allow multiple VPL programs (or VEM instances) to be composed:

$$\mathbf{v}_A^{\text{out}} \equiv \mathbf{v}_B^{\text{in}},$$

yielding a larger machine. This compositional model enables modular design of complex systems, including pipelines, feedback loops, and hybrid analog-digital systems.

10 General Algorithmic Procedure for VPL Encoding

To systematize the transformation of classical algorithms into the Vector Processing Language (VPL) formalism, we present a general compilation procedure. This procedure allows any algorithm—imperative, functional, or hybrid—to be expressed as a unified linear dynamical system in a finite-dimensional vector space.

10.1 Overview

Given an algorithm expressed as a sequence of statements with variables, conditionals, and loops, the goal of the VPL transformation is to construct:

- A global state vector $\mathbf{V} \in \mathbb{R}^n$ (or $\mathbb{C}^n$) containing both data and control variables.

- A transition matrix $M_{\text{full}} \in \mathbb{R}^{n \times n}$ describing the full control-data evolution.
- A constant vector $\mathbf{c}_{\text{full}}$ representing offsets or immediate values.
- A halting or phase constraint (C_p, c_p) defining termination or steady-state behavior.

10.2 Step-by-step procedure

Step 1: Parse and extract the algorithmic structure

1. Identify all variable declarations and initialize them with symbolic or numeric values.
2. Enumerate all statements, loops, and conditional blocks with unique identifiers.
3. Create a hierarchical control graph representing sequential, conditional, and iterative dependencies.

Algorithm $\rightarrow$ Abstract Syntax Tree (AST) $\rightarrow$ Control Flow Graph (CFG)

Step 2: Build the Control Vector Subspace

1. Assign a dedicated control vector component for each program block (e.g., `b_s1`, `b_s2`, ...).
2. Encode control transitions as entries of a matrix M_C, where:

$$(M_C)_{ij} = 1 \text{ iff control flows from block } i \text{ to } j.$$

3. Initialize the control vector $\mathbf{v}_C(0)$ to activate the entry block (e.g., initialization or main).

Step 3: Build the Data Vector Subspace

1. Collect all scalar and array variables into a data vector $\mathbf{x} = [x_1, x_2, \ldots, x_m]^T$.

2. For each assignment statement $(x_i := f(x_j, x_k, \dots))$, define a local transformation matrix M_{D_i} and offset vector $\mathbf{c}_{D_i}$:

$$\mathbf{x}' = M_{D_i}\mathbf{x} + \mathbf{c}_{D_i}.$$

3. Conditional and loop updates are handled as piecewise-affine combinations:

$$\mathbf{x}' = C(\mathbf{x})M_{D_1}\mathbf{x} + (1 - C(\mathbf{x}))M_{D_2}\mathbf{x},$$

where $C(\mathbf{x})$ is the conditional selector.

Step 4: Combine Control and Data Spaces
The full VPL vector is constructed as:

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_C \\ \mathbf{x} \end{bmatrix}, \quad \mathbf{V} \in \mathbb{R}^{n_C + n_D}.$$

The total transformation is represented by:

$$M_{\text{full}} = \begin{bmatrix} M_C & 0 \\ B & M_D \end{bmatrix},$$

where B encodes control-to-data coupling (which data operations activate when a control block is active).

Step 5: Define the Phasor or Halting Condition
Loops and termination conditions are expressed as a phasor constraint:

$$C_p\mathbf{V} + c_p = 0,$$

where C_p selects the variable or control flag that signals convergence (e.g., `x < 5`, `swapped = false`, etc.).

The iteration proceeds until this condition is met or a steady-state is reached in the phasor subspace.

Step 6: Simulation or Execution
At each discrete timestep t:

$$\mathbf{V}_{t+1} = M_{\text{full}}\mathbf{V}_t + \mathbf{c}_{\text{full}}.$$

Execution continues until the phasor convergence condition holds. For analog or quantum mapping, each iteration corresponds to a physical transformation or operator application.

Step 7: Expansion (optional)

For detailed analysis, the matrices can be expanded to an explicit or block-wise form:

$$M_{\text{expanded}} = \begin{bmatrix} M_C & 0 & 0 \\ B_1 & M_D & 0 \\ 0 & B_2 & M'_D \end{bmatrix},$$

allowing representation of nested control structures or multi-threaded flows.

10.3 Algorithmic summary

Algorithm 1 Generic Procedure for VPL Encoding

Require: Classical algorithm $\mathcal{A}$
Ensure: VPL matrices $(M_{\text{full}}, \mathbf{c}_{\text{full}}, C_p, c_p)$

```
Parse A into control and data components
Construct control matrix M_C
Construct data matrix M_D and offset c_D
Link control to data via coupling matrix B
Assemble global matrix M_full
Define halting condition (C_p, c_p)
Initialize state vector V_0
while not converged do
    V_{t+1} = M_full V_t + c_full
end while
return Final state V_{t*}
```

10.4 Remarks

- This procedure is agnostic to the nature of the algorithm: sequential, parallel, or recursive.
- All program constructs (loops, conditionals, function calls) are reducible to matrix transformations over finite-dimensional vector spaces.
- The same pipeline enables mappings to digital hardware, analog dynamics, or even quantum gates, depending on the algebraic field used ($\mathbb{R}$, $\mathbb{C}$, or Hilbert space).

This formal algorithmic pathway constitutes the *VPL compilation pipeline*, transforming abstract computation into vector space evolution under linear-algebraic dynamics.

11 General Problem Formulation in VPL

While the previous section outlined the transformation of a classical algorithm into the Vector Processing Language (VPL), in this section we extend the methodology to formulate problems directly in the VPL framework. This eliminates the need for intermediary code, allowing problems to be defined natively in terms of vector spaces, operators, and constraints.

11.1 Conceptual Overview

A computational problem can be represented as a transformation or evolution of a system's state vector

$$\mathbf{V} \in \mathbb{R}^n$$

subject to algebraic, logical, or dynamical constraints. The objective of VPL formulation is to express this transformation as:

$$\mathbf{V}_{t+1} = M_{\text{full}}\mathbf{V}_t + \mathbf{c}_{\text{full}},$$

with a halting or equilibrium condition:

$$C_p\mathbf{V} + c_p = 0.$$

Thus, a problem becomes the search for $\mathbf{V}^*$ satisfying both the dynamical law and the stopping constraint.

11.2 General Procedure

Step 1: Identify Problem Domain and State Variables

1. Determine the fundamental quantities of the problem: data, parameters, and control flags.
2. Assign each such quantity to an entry in the global state vector $\mathbf{V}$.
3. Define dimensions for all entities (scalars, arrays, indices, counters).

Example: For a shortest-path problem, $\mathbf{V}$ may include vertex distances, adjacency weights, and an active node index.

Step 2: Define Transformation Rules

1. Express how the state evolves between steps (e.g., relaxation, update, propagation, or optimization rules).
2. Encode each rule as a matrix or block within M_{full}.
3. If nonlinear relationships appear, approximate them via piecewise-linear or iterative operators.

$$\mathbf{V}_{t+1} = M_i \mathbf{V}_t + \mathbf{c}_i, \quad \text{for active rule } i.$$

Step 3: Establish Control Flow or Phase Activation

1. Introduce control components that indicate which rule or phase is currently active.
2. Create a control transition matrix M_C such that $(M_C)_{ij} = 1$ if the system moves from phase i to phase j.
3. The activation of a rule is determined by the control subvector $\mathbf{v}_C$.

This step defines how the system self-regulates: which operator applies at which stage.

Step 4: Formulate the Objective or Halting Condition

1. Define an algebraic constraint representing completion or convergence:

$$C_p \mathbf{V} + c_p = 0.$$

2. This condition generalizes stopping criteria such as:
   - Reaching steady-state ($\mathbf{V}_{t+1} = \mathbf{V}_t$)
   - Logical completion (e.g., all nodes visited, condition false)
   - Optimization convergence ($\|\nabla f(\mathbf{V})\| < \epsilon$)

Step 5: Construct the Global Operator System
Combine all data and control transformations into a unified block matrix:

$$M_{\text{full}} = \begin{bmatrix} M_C & 0 \\ B & M_D \end{bmatrix}, \quad \mathbf{c}_{\text{full}} = \begin{bmatrix} 0 \\ \mathbf{c}_D \end{bmatrix}.$$

Here:

- M_C: control dynamics (phase progression)
- M_D: data transformations (numeric updates)
- B: coupling between control and data

Step 6: Define Initial State
Specify:

$$\mathbf{V}_0 = \begin{bmatrix} \mathbf{v}_C(0) \\ \mathbf{x}(0) \end{bmatrix},$$

where $\mathbf{v}_C(0)$ activates the initial condition or phase and $\mathbf{x}(0)$ contains the problem's initial data (e.g., array, matrix, graph).

Step 7: Iterate or Solve
The evolution proceeds as:

$$\mathbf{V}_{t+1} = M_{\text{full}}\mathbf{V}_t + \mathbf{c}_{\text{full}},$$

until $C_p\mathbf{V} + c_p = 0$ is satisfied.

Depending on context, this can be:

- Discrete simulation (classical iteration)
- Continuous integration (analog circuit mapping)
- Operator evolution (quantum analog)

11.3 Problem Encoding Summary

Algorithm 2 General Procedure for Direct VPL Problem Encoding

Require: Problem specification P

Ensure: VPL matrices $(M_{\text{full}}, \mathbf{c}_{\text{full}}, C_p, c_p, \mathbf{V}_0)$

1: Identify variables and parameters to form $\mathbf{V}$
2: Define state transition rules and encode as matrices M_i
3: Construct control flow matrix M_C
4: Build data operator matrix M_D and offset $\mathbf{c}_D$
5: Combine them into M_{full}
6: Define halting condition (C_p, c_p)
7: Initialize $\mathbf{V}_0$
8: **while** not converged **do**
9: Update $\mathbf{V}_{t+1} = M_{\text{full}}\mathbf{V}_t + \mathbf{c}_{\text{full}}$
10: **end while**
11: **return** solution state $\mathbf{V}^*$

11.4 Remarks

- This approach treats *problems as dynamical systems*, where solving corresponds to evolving toward a fixed point.
- It naturally supports analog, neural, or quantum realizations.
- It generalizes optimization, simulation, and computation into a unified vector space formalism.

Hence, any problem P can be directly encoded into VPL by defining the structure of its vector space, the operators that evolve it, and the conditions that determine its solution.

11.5 Example: Direct Problem Formulation for a While Loop

To illustrate the general problem formulation procedure, consider the simple iterative process:

$$\texttt{while (x < 5)} \quad \{x = x + 2;\}$$

This problem describes a system that begins with $x = 0$ and evolves by repeatedly adding 2 until the condition $x < 5$ becomes false.

11.5.1 Step 1: Identify Problem Domain and Variables

We have a single scalar variable $x \in \mathbb{R}$, representing the system state. We define the initial state vector:

$$\mathbf{V}_0 = [x_0] = [0].$$

11.5.2 Step 2: Define Transformation Rule

The update rule is purely arithmetic:

$$x_{t+1} = x_t + 2.$$

This can be represented as a linear affine operator:

$$M_D = [1], \qquad \mathbf{c}_D = [2].$$

11.5.3 Step 3: Define Control and Halting Condition

The evolution continues while the predicate $x < 5$ holds. In vector form, this becomes a *phasor constraint*:

$$C_p = [1], \quad c_p = [-5],$$

where the evolution halts when $C_p\mathbf{V} + c_p = 0 \Rightarrow x = 5$.

Optionally, a binary control variable u may be introduced to represent the "active" phase of the loop:

$$u_{t+1} = u_t \cdot H(5 - x_t),$$

where H is the Heaviside step function (1 if the condition holds, 0 otherwise). When $u_t = 0$, the system halts automatically.

11.5.4 Step 4: Construct the Global Operator System

For the single-variable case, the global system can be represented as:

$$M_{\text{full}} = \begin{bmatrix} 1 \end{bmatrix}, \quad \mathbf{c}_{\text{full}} = \begin{bmatrix} 2 \end{bmatrix}.$$

Thus, the general dynamical form is:

$$\mathbf{V}_{t+1} = M_{\text{full}}\mathbf{V}_t + \mathbf{c}_{\text{full}},$$

with halting condition $C_p\mathbf{V} + c_p = 0$

11.5.5 Step 5: Solve the System Evolution

The closed-form solution is obtained by recursive expansion:

$$x_t = M_D^t x_0 + \sum_{k=0}^{t-1} M_D^k \mathbf{c}_D.$$

Since $M_D = [1]$,

$$x_t = 0 + 2t.$$

The halting condition $x_t = 5$ yields:

$$2t = 5 \Rightarrow t = 2.5.$$

In discrete iteration, the loop terminates after the smallest integer t where $x_t \geq 5$, i.e. after $t = 3$ iterations, yielding $x = 6$.

11.5.6 Step 6: Interpretation in VPL Terms

In VPL representation:

$$\mathbf{V}_{t+1} = \begin{bmatrix}1\end{bmatrix}\mathbf{V}_t + \begin{bmatrix}2\end{bmatrix}, \quad C_p = [1], \quad c_p = [-5].$$

The system halts when $C_p\mathbf{V} + c_p = 0$. Each iteration corresponds to a linear transformation in a 1-dimensional vector space, and the phasor condition acts as a boundary operator defining the stopping manifold.

11.5.7 Step 7: Optional Control-Data Coupled Representation

If control is explicitly represented (as in a general VPL implementation), the full system becomes two-dimensional:

$$\mathbf{V} = \begin{bmatrix}u\\x\end{bmatrix}, \quad M_{\text{full}} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}, \quad \mathbf{c}_{\text{full}} = \begin{bmatrix}0\\2u\end{bmatrix}.$$

The update rule is now:

$$\mathbf{V}_{t+1} = M_{\text{full}}\mathbf{V}_t + \mathbf{c}_{\text{full}},$$

with

$$u_{t+1} = H(5 - x_t), \quad x_{t+1} = x_t + 2u_t.$$

This formulation reveals the **decoupled yet coupled** nature of control and data within the VPL framework — the control (phasor) acts multiplicatively on the data update, and the condition itself evolves as a vector component.

11.5.8 Step 8: Result

The evolution of the state vector is summarized as:

t	x_t	u_t
0	0	1
1	2	1
2	4	1
3	6	0

The system halts when $u_t = 0$, i.e. when $x_t \geq 5$.

This compactly illustrates how a simple iterative loop can be formulated, simulated, and analyzed within the VPL mathematical framework as a **self-regulated linear dynamical system with a phasorial halting condition**.

11.6 Example: Direct Problem Formulation for Graph Traversal

To illustrate how graph algorithms can be represented and solved using the Vector Programming Language (VPL) framework, consider the problem of traversing a small directed graph $G = (V, E)$, defined as:

$$V = \{A, B, C, D\}, \quad E = \{(A, B), (A, C), (B, D), (C, D)\}.$$

We wish to visit all nodes reachable from the starting node A.

11.6.1 Step 1: Identify Problem Domain and Variables

We define a state vector representing the activation state of each node:

$$\mathbf{V}_t = \begin{bmatrix} v_A \\ v_B \\ v_C \\ v_D \end{bmatrix}, \quad v_i \in \{0, 1\},$$

where $v_i = 1$ indicates that node i is active (visited or currently in the traversal frontier).

The initial condition corresponds to activating only node A:

$$\mathbf{V}_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

11.6.2 Step 2: Constructing the Connectivity (Adjacency) Matrix

The structure of the graph is captured by its **adjacency matrix**, which represents how nodes are connected. Formally, we define:

$$M_G[i, j] = \begin{cases} 1, & \text{if there is a directed edge from node } i \text{ to node } j, \\ 0, & \text{otherwise.} \end{cases}$$

Given $E = \{(A, B), (A, C), (B, D), (C, D)\}$, we can write the connectivity explicitly:

$$M_G = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Each row i represents the outgoing edges of node i. For example:

- Row 1 corresponds to node A: it has outgoing edges to B and C, hence entries $M_G[1, 2] = 1$ and $M_G[1, 3] = 1$. - Row 2 corresponds to node B: it connects to D, so $M_G[2, 4] = 1$. - Row 3 corresponds to node C: it also connects to D, so $M_G[3, 4] = 1$. - Row 4 corresponds to node D: it has no outgoing edges, so all zeros.

This matrix representation provides a linear-algebraic description of connectivity — every traversal step corresponds to multiplying the current state vector $\mathbf{V}_t$ by $M_G^\top$.

11.6.3 Step 3: Define the Traversal Transformation

At each iteration, the traversal activates the next frontier of nodes reachable from the currently active ones:

$$\mathbf{V}_{t+1} = H(M_G^\top \mathbf{V}_t),$$

where $H(\cdot)$ is the Heaviside (threshold) function, converting nonzero entries to 1. In other words, if a node receives any connection from an active node, it becomes active at the next step.

This rule captures the essential traversal logic: "activate all neighbors of currently active nodes."

11.6.4 Step 4: Linearized VPL Formulation

In the VPL formalism, we model this transformation using a **data operator matrix** and **control condition**. Ignoring the nonlinearity (for analysis or simulation), we can write the update as a linear recurrence:

$$\mathbf{V}_{t+1} = (M_G + I)\,\mathbf{V}_t.$$

Here, the identity matrix I ensures that active nodes remain active (self-retaining state). This form makes traversal directly compatible with the general VPL evolution model:

$$\mathbf{S}_{t+1} = M_{\text{full}}\,\mathbf{S}_t + \mathbf{c}_{\text{full}}.$$

11.6.5 Step 5: Control Vector and Halting Condition

We introduce a control vector $\mathbf{u}_t$ that tracks which nodes have already been processed:

$$\mathbf{u}_{t+1} = \mathbf{V}_t - \mathbf{u}_t.$$

The complete system can be expressed as a combined state:

$$\mathbf{S}_t = \begin{bmatrix} \mathbf{V}_t \\ \mathbf{u}_t \end{bmatrix}, \quad \mathbf{S}_{t+1} = \begin{bmatrix} M_G + I & 0 \\ I & -I \end{bmatrix} \mathbf{S}_t.$$

Traversal halts when no new nodes are activated:

$$C_p \mathbf{S}_t + c_p = 0,$$

where $C_p = [I, -I]$ selects the difference between consecutive activation states.

11.6.6 Step 6: Numerical Example of Evolution

Starting from $\mathbf{V}_0 = [1, 0, 0, 0]^T$:

$$\begin{aligned} \mathbf{V}_1 &= H(M_G^\top \mathbf{V}_0) = [0, 1, 1, 0]^T, \\ \mathbf{V}_2 &= H(M_G^\top \mathbf{V}_1) = [0, 0, 0, 1]^T, \\ \mathbf{V}_3 &= H(M_G^\top \mathbf{V}_2) = [0, 0, 0, 0]^T. \end{aligned}$$

The cumulative activation vector gives:

$$\mathbf{V}_{\text{visited}} = \mathbf{V}_0 + \mathbf{V}_1 + \mathbf{V}_2 = [1, 1, 1, 1]^T,$$

confirming that all nodes are reachable from A.

11.6.7 Step 7: Phasor Interpretation

In the extended phasor form, each active node can be represented by a complex-valued control phasor:

$$\mathbf{\Phi}_t = e^{j\omega t} \mathbf{V}_t,$$

linking discrete traversal to continuous dynamical evolution. The traversal halts when the phasor field becomes stationary:

$$\Delta \mathbf{\Phi}_t = 0.$$

This representation naturally bridges **graph-based propagation**, **wave dynamics**, and **quantum analogs**, showing the deep unification potential of the VPL formalism.

11.6.8 Step 8: Summary of Vector Space Representation

$$\begin{aligned} M_{\text{data}} &= M_G + I, \\ \mathbf{c}_{\text{data}} &= \mathbf{0}, \\ C_p &= [I, -I], \\ c_p &= \mathbf{0}. \end{aligned}$$

Thus, the traversal process becomes a discrete-time vector evolution:

$$\mathbf{S}_{t+1} = M_{\text{full}}\mathbf{S}_t + \mathbf{c}_{\text{full}}, \quad \text{halt when } C_p\mathbf{S}_t + c_p = 0.$$

11.7 Direct Problem formulation for Parallel Graph Traversal

Given a directed graph

$$G = (V, E), \qquad V = \{A, B, C, D\},$$

with edges

$$E = \{(A, B),\ (A, C),\ (B, D),\ (C, D)\},$$

perform breadth-first-style traversal starting simultaneously from the two seeds

$$S^{(1)} = \{A\}, \qquad S^{(2)} = \{B\}.$$

We wish to activate (visit) all nodes reachable from either seed.

11.7.1 Step 1: State variables and vector space

Define the discrete frontier / activation vector

$$\mathbf{f}_t = [f_A(t),\ f_B(t),\ f_C(t),\ f_D(t)]^\top \in \{0, 1\}^4,$$

where $f_v(t) = 1$ indicates node v is active (in the frontier) at step t.

For bookkeeping of visited nodes we may define a cumulative visited vector

$$\mathbf{g}_t = [g_A(t),\ g_B(t),\ g_C(t),\ g_D(t)]^\top,$$

where $g_v(t) = 1$ if v has been visited at or before step t.

We combine control (phase) and data into the global state

$$\mathbf{V}_t = \begin{bmatrix} \mathbf{c}_t \\ \mathbf{f}_t \\ \mathbf{g}_t \end{bmatrix},$$

where $\mathbf{c}_t$ is a small control vector (e.g. $\{c_{\text{init}}, c_{\text{step}}, c_{\text{done}}\}$) used to gate application of the traversal operator. For this example we will use a simple single-phase control with two logical phases (step / done), but the formalism generalizes.

11.7.2 Step 2: Connectivity (adjacency) operator

Construct the adjacency matrix A_G with the convention $A_G[i, j] = 1$ iff there is an edge $i \to j$. For our graph (ordering nodes as A, B, C, D):

$$A_G = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Acting on a frontier vector the next-candidates (pre-deduplication) are given by

$$\mathbf{h}_{t+1} = A_G^\top \mathbf{f}_t,$$

because column j of $A_G^\top$ collects incoming edges to node j.

11.7.3 Step 3: Linearized traversal operator and saturation

To remain in linear algebra, we adopt the standard linearized-with-saturation model:

$$\mathbf{f}_{t+1} = \sigma\big(A_G^\top \mathbf{f}_t\big),$$

where $\sigma(\cdot)$ is an element-wise saturation function (e.g. thresholding to $\{0, 1\}$). For algebraic analysis we can work with the pre-threshold linear quantity

$$\tilde{\mathbf{f}}_{t+1} = A_G^\top \mathbf{f}_t,$$

and view the threshold as an external projection step when mapping to discrete software or hardware.

To ensure already-active nodes persist in the cumulative visited set we define

$$\mathbf{g}_{t+1} = \mathbf{g}_t + \sigma(\tilde{\mathbf{f}}_{t+1}),$$

with implicit saturation of $\mathbf{g}$ entries to 1.

11.7.4 Step 4: Parallel (superposed) initial condition

Parallel traversal from two seeds is expressed as the superposition (sum) of the two frontier vectors:

$$\mathbf{f}_0 = \mathbf{f}_0^{(1)} + \mathbf{f}_0^{(2)} = [1,0,0,0]^\top + [0,1,0,0]^\top = [1,1,0,0]^\top.$$

This single initial vector encodes both traversals simultaneously; linearity of $A_G^\top$ guarantees that propagation from both seeds happens in the same evolution step.

11.7.5 Step 5: Control and global operator form

Introduce a compact control vector $\mathbf{c}_t \in \mathbb{R}^2$ with basis $\{\text{step}, \text{done}\}$. For clarity we put the control gate in front of data updates. The full state is

$$\mathbf{V}_t = \begin{bmatrix} \mathbf{c}_t \\ \mathbf{f}_t \\ \mathbf{g}_t \end{bmatrix} \in \mathbb{R}^{2+4+4} = \mathbb{R}^{10}.$$

Define the block transition as

$$\mathbf{V}_{t+1} = M_{\text{full}}\, \mathbf{V}_t + \mathbf{c}_{\text{full}},$$

with the block structure

$$M_{\text{full}} = \begin{bmatrix} M_{CC} & 0 & 0 \\ B_{FC} & 0_{4\times 4} & 0_{4\times 4} \\ 0 & I_4 & I_4 \end{bmatrix}, \qquad \mathbf{c}_{\text{full}} = \mathbf{0}.$$

Here:

- M_{CC} advances control from init to step and eventually to done,
- B_{FC} implements the gated application of the adjacency operator: when the control is in step, we set $\tilde{\mathbf{f}}_{t+1} = A_G^\top \mathbf{f}_t$.

Concretely, if we write the control basis as $[c_{\text{step}}, c_{\text{done}}]$, a simple B_{FC} that applies $A_G^\top$ when $c_{\text{step}} = 1$ can be written conceptually as

$$B_{FC} = \begin{bmatrix} c_{\text{step}} \cdot A_G^\top & \mathbf{0} \end{bmatrix},$$

which in matrix algebra corresponds to embedding $A_G^\top$ into appropriate sub-block columns gated by the control component. (In discrete implementations the gating is implemented by zeroing or selecting rows/columns conditioned on control; in analog/continuous mapping the gating can be multiplicative.)

The cumulative visit update $\mathbf{g}_{t+1} = \mathbf{g}_t + \sigma(\tilde{\mathbf{f}}_{t+1})$ is represented by the lower-right block I_4 (accumulation) plus the injection of $\tilde{\mathbf{f}}_{t+1}$ into $\mathbf{g}$.

11.7.6 Step 6: Phasor halting condition

Traversal halts when the frontier becomes empty (no newly active nodes remain). Equivalently, when the linearized pre-threshold vector is zero after projection:

$$C_p \mathbf{V}_t + c_p = 0, \quad C_p = [\, 0_{1\times 2} \mid \mathbf{1}_{1\times 4} \mid \mathbf{0}_{1\times 4} \,], \quad c_p = 0,$$

so that $C_p \mathbf{V}_t = \sum_j f_j(t)$. The halting criterion is $\sum_j f_j(t) = 0$.

(If using thresholded updates, one tests the saturated frontier; in linear analysis one can test the norm $\|A_G^\top \mathbf{f}_t\|$ or the projection above.)

11.7.7 Step 7: Solve the evolution (numeric trace)

With

$$\mathbf{f}_0 = [1,1,0,0]^\top, \qquad \mathbf{g}_0 = \mathbf{f}_0,$$

compute successive (thresholded) updates:

$$\tilde{\mathbf{f}}_1 = A_G^\top \mathbf{f}_0 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \qquad \Rightarrow \quad \mathbf{f}_1 = \sigma(\tilde{\mathbf{f}}_1) = [0,1,1,0]^\top,$$

$$\mathbf{g}_1 = \mathbf{g}_0 + \mathbf{f}_1 = [1,1,1,0]^\top,$$

$$\tilde{\mathbf{f}}_2 = A_G^\top \mathbf{f}_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \Rightarrow \quad \mathbf{f}_2 = [0,0,0,1]^\top,$$

$$\mathbf{g}_2 = \mathbf{g}_1 + \mathbf{f}_2 = [1,1,1,1]^\top,$$

$$\tilde{\mathbf{f}}_3 = A_G^\top \mathbf{f}_2 = \mathbf{0}, \qquad \Rightarrow \quad \mathbf{f}_3 = \mathbf{0},$$

$$\mathbf{g}_3 = \mathbf{g}_2.$$

At $t = 3$ the phasor halting condition $\sum_j f_j(3) = 0$ is satisfied and the traversal terminates with $\mathbf{g}_3 = [1, 1, 1, 1]^\top$ (all nodes reached).

11.7.8 Step 8: Parallelism interpretation

Parallelism here is entirely algebraic:

$$A_G^\top(\mathbf{f}_t^{(1)} + \mathbf{f}_t^{(2)}) = A_G^\top \mathbf{f}_t^{(1)} + A_G^\top \mathbf{f}_t^{(2)},$$

so a single application of $A_G^\top$ propagates both traversals simultaneously. No separate control threads are required — the superposed initial frontier vector embeds multi-source exploration in the same linear evolution.

11.7.9 Step 9: Hardware mapping remarks

- **Digital:** implement the frontier and visited registers, compute $A_G^\top \mathbf{f}$ via bitwise combinational logic (or sparse multiply), then threshold and accumulate.
- **Analog:** represent $\mathbf{f}$ as voltages/currents; connections in A_G are implemented by weighted interconnects; summing nodes produce the next frontier; a comparator array performs thresholding.
- **Quantum (conceptual):** a superposed initial state over nodes evolves under an operator derived from A_G (or its normalized form), analogous to discrete-time quantum walk; measurement yields reachable amplitude distribution.

11.7.10 Summary

This formulation demonstrates how a parallel graph traversal is written directly as a VPL problem:

- express connectivity as a linear operator A_G;
- encode frontier and visited sets as data subspaces;
- initialize the frontier with a superposition of seeds;
- iterate the linear operator (with thresholding) until the phasor halting condition (empty frontier) is met.

All steps are native to the VPL/VEM formalism and readily map to numerical simulation or hardware realizations.

12 Transformation Rules: From Algorithmic Constructs to VPL

In order to systematically encode classical algorithmic structures into the Vector Programming Language (VPL) framework, we define a set of canonical *transformation rules*. Each rule maps a high-level construct (control structure or operation) to its equivalent VPL formulation in terms of:

- **Control vectors c** (activating or gating operations),
- **Data vectors x** or state subspaces,
- **Linear transformations** M,
- **Activation or halting conditions** $C\mathbf{v} + c$.

Construct	Description / Classical Semantics	VPL Transformation Rule
`Initialization`	Assign an initial value to a variable or state.	Set initial vector $\mathbf{x}_0$. Control $\mathbf{c}_0$ activates the *init* phase. M may be identity or direct assignment operator.
`Assignment`	$x := f(x)$ or $x := c$.	Introduce transformation M_{assign} such that $\mathbf{x}_{t+1} = M_{assign}\mathbf{x}_t + \mathbf{c}_{assign}$.
`If / Condition`	Branching depending on a Boolean predicate.	Implement with a *phasor or control gating vector* $\mathbf{c}$. Define $C_p\mathbf{v} + c_p$ as a halting/activation condition; branch is realized by selecting one transformation block or another based on control state.
`While loop`	Execute a body while $cond(x)$ is true.	Encode condition as phasor halting rule $C_p\mathbf{v} + c_p \neq 0$. Control cycle: $\mathbf{c}_t \to$ loop body $\to$ check condition $\to$ either repeat or transition to `done`.
`For loop`	Fixed iteration loop over $0, \ldots, N-1$.	Expand as linear iteration with step counter $\mathbf{k}_t$ and decrement operator. Halting: $k = 0$.
`Function call`	Call procedure f.	Map to sub-block operator M_f acting on a local subspace. Control vectors $\mathbf{c}_{call}, \mathbf{c}_{return}$ manage entry/exit.
`Parallel execution`	Multiple threads/branches active at once.	Superpose initial state vectors. Single operator application evolves all in parallel: $M(\mathbf{v}^{(1)} + \mathbf{v}^{(2)} + \cdots) = \sum_i M\mathbf{v}^{(i)}$. No explicit scheduling needed.
`Break / Halt`	Exit loop or program early.	Trigger halting phasor $C_h\mathbf{v}+c_h = 0$ and transition control vector to `done` state.
`Graph traversal / propagation`	Visit neighbor nodes.	Define adjacency matrix A_G. Next frontier $\mathbf{f}_{t+1} = A_G^\top \mathbf{f}_t$. Add thresholding or gating by control.
`Recursion`	Function calls itself on smaller data.	Model using stack/state replication in the control vector $\mathbf{c}$ and hierarchical block structure of M. Unfold or embed as fixed-depth or lazy operator expansion.
`I/O Interaction`	Read / write external signals.	Represent as source or sink subspaces: external inputs are appended to $\mathbf{v}_t$ before applying M, outputs are projections $y_t = P\mathbf{v}_t$.
`Halting`	Program ends.	Defined by halting phasor condition $C_h\mathbf{v}+c_h = 0$. The `done` state of $\mathbf{c}$ disables all transformations.

Table 2: Canonical transformation rules mapping classical algorithmic constructs to VPL formulation.

This conversion table allows the systematic transformation of an arbitrary algorithm into a set of VPL components. The key idea is that *every control construct* can be expressed as:

$\mathbf{v}_{t+1} = M\mathbf{v}_t + \mathbf{c}$, with halting and activation encoded in control vectors and phasor conditions.

This formulation not only provides a path to **parallel execution** by default, but also makes direct mapping to **hardware** (digital, analog, or quantum) possible.

13 Problem Class to VPL Transformation Rules

While the previous section presented transformation rules for algorithmic constructs, it is also possible to formulate entire *problem classes* directly in VPL without first specifying their algorithmic structure.

Each problem class can be characterized by:

- The **state space decomposition** into data and control subspaces,
- The **linear operator structure** M or derived matrices (e.g., adjacency, propagation, or cost operators),
- The **control flow and phasor halting conditions**,
- The **output projection** or solution representation.

Problem Class	Conceptual Formulation	VPL Transformation Pattern
Looping / Iteration	Repeated update of a variable or state until condition is met (e.g., convergence, threshold).	State space: $\mathbf{x}$ (data), $\mathbf{c}$ (control). Operator: iteration matrix M. Halting: phasor condition $C_p\mathbf{v} + c_p = 0$.
Search / Exploration	Traverse a structure (e.g., graph, string, tree) until target is found.	Data space: position or frontier. Operator: adjacency or shift matrix A. Halting: match or visited condition encoded in control phasor.
Graph Traversal	Explore nodes and edges in parallel (e.g., BFS, DFS).	$M = A_G^\top$ (adjacency propagation), $\mathbf{c}$ defines visited/unvisited states. Parallelism is implicit. Halting: empty frontier or condition met.
String Matching	Compare sequences symbol by symbol.	Data = encoded symbols. M = shift/compare operator. Halting: mismatch or end of string. Output: match flag.
Sorting / Ordering	Reorder a sequence based on comparisons.	M encodes comparison-exchange network. Control = pass structure. Halting = stable order.
Optimization (Iterative)	Improve solution iteratively (gradient, heuristic).	M implements update step. Control tracks convergence. Halting when objective stable or below tolerance.
Linear Algebra Problems	Solve systems, eigensolvers, transformations.	M = linear system operator. Control = iteration or exact solver. Halting when norm residual = 0.
Decision Problems	True/False outcome, possibly complex evaluation tree.	Encode decision tree as control vector graph. Halting when terminal node reached.
Pattern Recognition / Matching	Detect occurrence of structure in data.	M encodes propagation over data space. Control vector gates pattern comparison. Halting when pattern found.
Quantum / Waveform Problems	Problems expressible through unitary evolution or phasors.	M can be made unitary or complex-valued. Control and data in Hilbert subspace. Halting via phasor amplitude criteria.
Simulation Problems	Iteratively simulate system dynamics.	M encodes dynamics (discrete step). Halting at time horizon or steady state.

Table 3: Canonical problem classes and their direct VPL transformation patterns.

This table allows us to bypass the algorithmic intermediate step: once a problem is categorized, its VPL formulation can be generated directly as:

$$\mathbf{v}_{t+1} = M\mathbf{v}_t + \mathbf{c},$$

with appropriate control and halting conditions. This is particularly powerful for large classes of problems (e.g., search, graph, optimization) that naturally map onto linear or affine transformations.

14 Canonical Problem Classes and Transformation Patterns

A central advantage of the Vector Programming Language (VPL) formalism is that any computable problem can be represented as a state evolution system of the form:

$$\mathbf{v}_{t+1} = M\mathbf{v}_t + \mathbf{c}, \tag{2}$$

with a halting or terminal condition

$$C_p\mathbf{v} + c_p = 0. \tag{3}$$

Here, M encodes the evolution operator (control, data flow, or both), while $\mathbf{v}_t$ represents the state vector at time t.

Although the number of problems in computation is theoretically unbounded, their mappings to VPL representations can be structured into a *finite set of canonical problem classes*. This is analogous to the way any logical circuit can be constructed from a small set of basic gates, or how any unitary quantum circuit can be constructed from a universal gate set.

14.1 Canonical VPL Problem Classes

#	Problem Class	Examples	Typical Matrix Structure
1	Iteration / Looping	Counter loops, recurrence relations	Shift, identity
2	Search / Exploration	DFS, BFS, pattern search	Adjacency, permutation
3	Graph Traversal	BFS, DFS, spanning trees, parallel search	Sparse adjacency
4	String / Sequence Matching	Regex matching, sequence comparison	Shift, compare matrices
5	Sorting / Ordering	Bubble sort, merge sort, selection	Comparison-exchange networks
6	Optimization (Iterative)	Gradient descent, local search	Affine operators, gradient steps
7	Linear Algebraic	Solvers, transformations, PCA	Dense or structured linear operators
8	Decision Problems	SAT, branching logic, control trees	Control graph as matrix
9	Pattern Recognition / Signal	Filters, neural layers	Convolutional / structured linear maps
10	Quantum / Waveform	Unitary evolution, interference	Unitary matrices
11	Simulation / Dynamics	PDE discretization, cellular automata	Evolution matrices
12	Combinatorial Enumeration	Backtracking, generators	State-space unfolding
13	Automata / Language Processing	DFA, NFA, regex engines	Transition matrices
14	Probabilistic / Stochastic	Markov chains, random walks	Stochastic matrices
15	Control and Scheduling	Timers, clocks, dispatching	Time-driven or event-triggered matrices
16	Hybrid / Mixed Systems	Multi-domain (e.g., graph + optimization)	Block matrix composition

Table 4: Canonical problem classes and their typical VPL matrix structure.

14.2 Transformation Rationale

Each problem class corresponds to a *transformation rule* that maps:

1. The abstract problem definition into a set of state variables $\mathbf{v}$.
2. Control or data dependencies into an evolution operator M.
3. Halting or boundary conditions into a phasor or projector C_p.

The practical outcome is that any algorithm—be it imperative, functional, or quantum—can be mapped into a canonical VPL class or a combination thereof.

14.3 Compositionality

Many real-world problems belong to multiple classes simultaneously. For example:

- A parallel graph traversal with optimization can be represented as a **block matrix** combining Classes 3 and 6.
- A quantum-enhanced search can be seen as a combination of Classes 2 and 10.
- A hybrid control system with stochastic effects may use Classes 14 and 15.

This compositionality makes VPL a *universal algorithmic substrate*, capable of describing classical, quantum, stochastic, and hybrid computations in a single linear algebraic language.

14.4 Relation to Classical Complexity Classes

While VPL is not defined in terms of Turing machine tape operations, its expressive power covers at least the Church–Turing domain and provides a pathway for modeling computational processes that can leverage parallelism, analog computation, and even quantum evolution. Each canonical class can be mapped to known complexity classes (P, NP, BQP, PSPACE, etc.) depending on the structure of M and the allowed parallelism.

14.5 Implications

- **Finite Basis for Infinite Problems:** These canonical classes act as a basis set for representing any computable problem.
- **Automated Compilation:** Transformation rules can be codified, allowing automatic translation of abstract problem specifications into VPL programs.
- **Hardware Mapping:** Once represented in VPL, problems can be directly mapped into digital circuits, analog systems, or even quantum circuits.

This section provides the theoretical foundation for the systematic conversion from arbitrary problems to VPL representations, which underpins the transformation tables and compilation procedures described in later sections.

15 Transformation Rules Table

One of the central contributions of the Vector Programming Language (VPL) framework is the existence of *systematic transformation rules* that allow the formulation of any algorithm or problem as a linear state evolution process:

$$\mathbf{v}_{t+1} = M\mathbf{v}_t + \mathbf{c}, \tag{4}$$

with termination or boundary conditions:

$$C_p\mathbf{v} + c_p = 0. \tag{5}$$

Each canonical problem class (Table **??**) can be mapped through a structured sequence of transformation steps, which translate high-level algorithmic concepts (loops, conditionals, traversal, optimization) into VPL constructs.

#	Problem Class	Variable Mapping	Matrix Construction	Termination / Control Rule
1	Iteration / Looping	Loop counters, state flags	Shift or increment matrices	Stop when counter reaches bound
2	Search / Exploration	Node states, visited flags	Permutation / adjacency	All goals visited or target found
3	Graph Traversal	Node set, active frontier	Adjacency matrix propagation	Frontier empty or visited all
4	String / Sequence Matching	Character indices, match flags	Shift and comparison matrices	String length reached or mismatch
5	Sorting / Ordering	Array elements, index pointers	Comparison-swap network matrix	Sorted flag or fixed point reached
6	Optimization (Iterative)	State vector, gradient	Gradient update matrix	Convergence or max iterations
7	Linear Algebraic	Vector/matrix elements	Direct linear operators	Residual $\|Ax - b\| < \epsilon$
8	Decision Problems	Boolean flags, branch states	Control matrix with guards	Accept / reject condition
9	Pattern Recognition	Feature vectors	Convolutional or structured M	Classification or fixed threshold
10	Quantum / Waveform	Amplitude states	Unitary U as M	Measurement or max steps
11	Simulation / Dynamics	Physical state variables	Discretized evolution matrix	Time step or stability limit
12	Combinatorial Enumeration	Search space states	Expansion / unfolding matrix	Exhausted search space
13	Automata / Language	DFA states	Transition matrix	Final state or dead state
14	Probabilistic / Stochastic	Probability distribution	Stochastic transition matrix	Steady state or threshold
15	Control and Scheduling	Timers, event flags	Time-step or priority matrix	End of schedule
16	Hybrid / Mixed Systems	Block-structured variables	Block-composed operators	Mixed stopping criteria

Table 5: Transformation rules from abstract problem structure to VPL representation.

15.1 General Transformation Procedure

For any problem class, the following steps define the canonical mapping to VPL:

1. **Identify State Variables:** Determine the minimal set of variables required to describe the state of the system. These become components of the vector $\mathbf{v}$.

2. **Define Evolution Logic:** Identify how each state variable changes with each computational step. Encode this logic in the matrix M and vector $\mathbf{c}$.

3. **Establish Control / Halting Conditions:** Define conditions that correspond to successful termination, fixed points, or steady states. These are represented through C_p and c_p.
4. **Construct Initial State:** Encode the problem input in $\mathbf{v}_0$.
5. **Simulate or Compile:** Evolve $\mathbf{v}$ via M until termination is reached.

15.2 Compositional Transformation

More complex problems are transformed by combining elementary transformation rules:

- **Sequential Composition:** Block matrix concatenation.
- **Parallel Composition:** Direct sum of matrices or block-diagonal structures.
- **Conditional Composition:** Controlled submatrices activated by guard variables.

15.3 Transformation Examples

While Loop (Class 1):

- State variable: x
- Evolution: $x \mapsto x + 2$
- Condition: $x < 5$
- $M = [1]$, $c = [2]$, $C_p = [1]$, $c_p = -5$.

Graph Traversal (Class 3):

- State variables: active frontier, visited nodes
- Evolution: propagate frontier via adjacency matrix
- Condition: no new active nodes
- M = adjacency matrix (possibly block-composed), C_p = projection on empty frontier.

Quantum Evolution (Class 10):

- State variables: complex amplitudes
- Evolution: $\mathbf{v}_{t+1} = U\mathbf{v}_t$
- Condition: measurement or max steps

15.4 Automation and Compilation

These transformation rules enable:

- Automated compilation of problem specifications into VPL.
- Systematic mapping to digital, analog, and quantum architectures.
- A uniform interface for simulation, verification, and hardware generation.

Remark: This table plays the same role in VPL as logic gate sets do in digital design: it provides a finite set of construction rules for an unbounded class of algorithms.

15.5 Representative Matrix Structures by Problem Class

Table 6 illustrates how various problem classes are encoded in the VPL formalism. For each class, we show: (1) a representative example, (2) the typical matrix structure of M, and (3) the general form of M itself, emphasizing its semantics.

#	Problem Class	Example	Typical Matrix Structure	Representative M Form
1	**Iteration / Looping**	`while (x<5) x=x+2;`	1×1 scalar matrix (identity)	$M = [1]$, $c = [2]$; simple additive iteration
2	**Linear Recurrence**	Fibonacci sequence	2×2 constant recurrence matrix	$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
3	**Graph Traversal**	BFS or DFS propagation	Adjacency matrix of graph	$M = A_{graph}$, where $A_{ij} = 1$ if edge (i, j) exists
4	**String / Sequence Matching**	"Hello" == "Hello"	Block-diagonal or shift structure	$M = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ controlling position shifts
5	**Sorting / Swapping**	Bubble Sort	Permutation/swap matrix	M_{ij} swaps positions (i, j) based on comparison mask
6	**Optimization (Iterative)**	Gradient Descent	Diagonal with decay or learning rate	$M = I - \eta\nabla^2 f(x)$, $c = \eta\nabla f(x)$
7	**Decision / Branching**	If–Else logic	Block matrix with guards	$M = \begin{bmatrix} M_{true} & 0 \\ 0 & M_{false} \end{bmatrix}$, activated by condition vector
8	**Graph Search**	Dijkstra or A*	Weighted adjacency matrix	$M_{ij} = e^{-w_{ij}}$ or normalized edge weights
9	**Probabilistic / Markov**	Markov chain evolution	Stochastic matrix (rows sum to 1)	$M_{ij} = P(X_{t+1} = j \mid X_t = i)$
10	**Quantum Evolution**	1-qubit rotation	Unitary matrix ($U^\dagger U = I$)	$M = e^{-iHt}$ or $R_y(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$
11	**Differential Simulation**	$x' = Ax$ dynamics	Continuous-time exponential form	Discrete step $M = e^{A\Delta t}$, $c = 0$
12	**Pattern Recognition**	Neural feature propagation	Sparse / convolutional matrix	M = Toeplitz or convolution kernel operator
13	**Automata / DFA**	State machine transitions	Binary transition matrix	$M_{ij} = 1$ if $\delta(q_i, a) = q_j$, else 0
14	**Combinatorial Enumeration**	Counting subsets	Upper-triangular matrix	$M_{ij} = 1$ if subset i extends j by one element
15	**Control Systems**	PID feedback loop	Block structure coupling plant and controller	$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ with feedback coupling
16	**Hybrid Systems**	Mixed discrete-continuous model	Block-diagonal with cross terms	$M = \begin{bmatrix} M_{disc} & 0 \\ B & M_{cont} \end{bmatrix}$

Table 6: Representative VPL matrix structures across canonical problem classes.

Each of these M matrices captures the *complete logic* of the correspond-

ing algorithmic step. The choice of M (and c) determines whether the system behaves deterministically, probabilistically, or continuously, making the VPL model general enough to represent classical, analog, and quantum computations under a single algebraic framework.

15.6 Foundational Construction of the Control Matrix M

In the Vector Processing Language (VPL), the matrix M governs the evolution of the program's state vector through a linear or quasi-linear transformation of the form:

$$X_{t+1} = MX_t + C,$$

where M encodes control flow and data dependencies, and C encodes constant or affine shifts. To ensure generality and a formal foundation, M can be systematically constructed from a small set of primitive operators and algebraic composition rules. This construction provides the same universality as logic gates in digital circuits or universal gates in quantum computation.

15.6.1 Primitive Matrices (Atomic Constructors)

The foundational building blocks of M are defined as a minimal algebraic basis:

Symbol	Name	Meaning / Example	Form
I	Identity	Pass-through / no change	$M = I$
S_{ij}	Swap	Exchanges vector components	$M_{pq} = 1$ if $(p, q) = (i, j)$ or (j, i)
A_k	Additive Shift	Adds constant or variable	$X' = X + k$
D_α	Damping / Scaling	Multiplies by scalar α	$M = \alpha I$
T_f	Functional Map	Applies unary transformation $f(x)$	$X' = f(X)$
C_ϕ	Conditional Mask	Activates sub-block under condition ϕ	$M = I \cdot \chi(\phi)$
P_θ	Phasor Operator	Rotational or oscillatory control	$M = e^{i\theta} I$
Π	Projection	Selects subspace / subset of variables	$M = \text{diag}(1, 0, ...)$

Each of these primitive matrices corresponds to a basic computational element. Together, they can represent data transfer, arithmetic operations, logical conditions, and cyclic control behavior.

15.6.2 Composition Rules

Higher-order structures such as loops, conditionals, and concurrent blocks are built by composing the primitive matrices through algebraic operators. The following composition rules define the formal grammar of VPL transformations:

- **Sequential Composition:**

$$M = M_2 M_1$$

  Represents consecutive execution, where M_1 is followed by M_2.

- **Parallel Composition:**

$$M = M_1 \oplus M_2$$

  Represents concurrent execution through block-diagonal composition.

- **Conditional Composition:**

$$M = C_\phi M_1 + (I - C_\phi) M_2$$

  Implements *if-else* behavior, controlled by mask C_ϕ.

- **Loop Closure (Recurrent Composition):**

$$M = (I - M_L)^{-1}$$

  Encodes self-recurrent transitions in a while or for loop, analogous to fixed-point iteration.

- **Tensor Composition:**

$$M = M_1 \otimes M_2$$

  Represents Cartesian products of subsystems, e.g., multi-agent or coupled-state systems.

These operators form a closed algebraic system, ensuring that any computable control structure can be expressed as a composition of primitive transformations.

15.6.3 Example: While Loop Construction

Consider the simple loop:

```
while (x < 5) { x = x + 2; }
```

The loop can be represented algebraically as:

$$\begin{aligned} M_{\text{inc}} &= I, \quad C_{\text{inc}} = [2], \\ C_{\text{while}} &= \chi(x < 5), \\ M_{\text{loop}} &= C_{\text{while}} \cdot M_{\text{inc}}, \\ M_{\text{total}} &= (I - M_{\text{loop}})^{-1}. \end{aligned}$$

Here, the mask C_{while} governs the activation of the loop body, and the recurrent form $(I - M_{\text{loop}})^{-1}$ encodes the repeated execution until the condition becomes false.

15.6.4 Hierarchy of Construction

The generative hierarchy of matrix constructions is summarized as:

Level	Construct	Description
0	Scalars and Identity	Base numerical transformations
1	Atomic Operators	Simple variable updates
2	Control Operators	Conditionals and masks
3	Composite Blocks	Loops and subroutines
4	Parallel / Tensor Networks	Multi-agent and concurrent systems
5	Adaptive / Dynamic Systems	Continuous-time or time-varying $M(t)$

This hierarchy provides a scalable pathway from primitive computational steps to complex algorithmic systems, including analog, digital, and quantum realizations.

15.6.5 Universality of the Basis

Analogous to the universality of NAND gates in digital logic or the $\{H, S, \text{CNOT}\}$ gate set in quantum computing, the VPL computational model admits a *universal generating set*:

$$\mathcal{B}_{VPL} = \{I, S, A, D, C_\phi, P_\theta\},$$

from which any computable transformation can be constructed through successive compositions.

This establishes the VPL matrix formalism as a general computational substrate, bridging digital, analog, and quantum paradigms under a unified linear algebraic representation.

16 Constructive Algorithm Definition in VPL

The Vector Programming Language (VPL) provides a mathematical substrate in which both data and control structures are expressed as transformations in a common vector space. This section introduces a constructive procedure for defining algorithms directly in VPL, starting from its primitive operations and progressing to composite algorithmic building blocks.

16.1 From Primitives to Algorithmic Constructs

The VPL core primitives, denoted

$$\{I, S_{ij}, A_k, D_\alpha, T_f, C_\phi, P_\theta, \Pi\},$$

represent the atomic transformations of the vector state $X \in \mathbb{R}^n$:

- I — Identity transformation.
- S_{ij} — Swap operator between components i and j.
- A_k — Additive or assignment operator.
- D_α — Scaling or derivative operator.
- T_f — Functional transform corresponding to a user-defined mapping f.
- C_ϕ — Conditional projection defined by predicate ϕ.
- P_θ — Phasor or rotation operator, used in oscillatory and quantum mappings.
- Π — Projection or output extraction operator.

Each primitive acts linearly or affinely on X:

$$X' = M_{\text{primitive}} X + C_{\text{primitive}}.$$

Compositions of these primitives define higher-level algorithmic patterns.

16.2 Macro-Blocks and Building Blocks

By combining primitives, VPL defines a set of *macro-blocks*—canonical substructures corresponding to classical control constructs. Table 7 summarizes their definitions.

Table 7: VPL Building Blocks as Compositions of Primitives

Building Block	**Description**	**Constructed From**	**Mathematical Form**
Assignment	Updates variable	A_k, D_α	$X' = D_\alpha X + A_k$
Comparison	Logical condition	C_ϕ	$\chi(\phi(X))$
Conditional	Branching	C_ϕ, I	$M = C_\phi M_1 + (I - C_\phi) M_2$
Loop	Iteration	C_ϕ, M	$(I - C_\phi M)^{-1}$
Function Call	Subroutine	Π, T_f	$X' = \Pi_f T_f(X)$
Parallel Region	Concurrent paths	$M_1 \oplus M_2$	Block-diagonal merge
Communication	Data sharing	S_{ij}, A_k	Swap + Addition
Phasor Update	Oscillatory control	P_θ	$e^{i\theta} X$

These macro-blocks form the basis for all higher-order algorithmic constructs expressible within VPL. Each block is parameterizable, allowing for compositional and recursive algorithm design.

16.3 Constructive Procedure for Algorithm Definition

The general procedure to construct an algorithm in VPL is as follows:

1. **Define the Computational Space:** Identify all variables and control states. These become the basis vectors of the composite state space X.

2. **Select Primitives:** For each atomic operation, choose the appropriate VPL primitive or macro-block.

3. **Define Connectivity:** Establish the transitions between control blocks using the adjacency or control matrix M.

4. **Compose Transformations:** Combine primitives according to algebraic rules:
$$\{M_{\text{seq}}, M_{\text{par}}, M_{\text{cond}}, M_{\text{loop}}\}.$$

5. **Define Constants:** Specify the affine offset vector C for additive updates or inputs.

6. **Assemble the Total Model:**
$$X_{t+1} = M_{\text{global}} X_t + C,$$
where M_{global} encodes the control and data flow of the entire program.

16.4 Illustrative Example: While Loop

Consider a simple algorithm: increment x by 2 while $x < 5$, then print x.

$$\begin{gathered} x = 0; \\ \text{while}(x < 5) : x = x + 2; \\ \text{print}(x); \end{gathered}$$

In VPL form:

$$\begin{aligned} X_0 &= [x = 0], \\ C_{x<5} &: \text{conditional mask operator}, \\ A_2 &: \text{addition operator}. \end{aligned}$$

The composite transformation is

$$M = (I - C_{x<5} A_2)^{-1}, \quad C = [0].$$

The output projection is given by Π_x, corresponding to the print operation.

16.5 Universality and Library Construction

This constructive framework defines a one-to-one mapping between algorithmic structures and algebraic operators. The result is a universal formalism suitable for automatic translation into:

- digital execution models (VEM simulators),

- analog and mixed-signal circuits,
- and unitary (quantum) systems through P_θ-based mappings.

A standard VPL library may thus be organized as:

- `vpl.core` — atomic primitives (I, S, A, D, C, P);
- `vpl.control` — control flow blocks (if, while, for);
- `vpl.math` — arithmetic and algebraic operators;
- `vpl.parallel` — block-diagonal and tensor constructions;
- `vpl.quantum` — unitary and phasorial mappings;
- `vpl.io` — input/output vector projections.

Together, these define a complete methodology for constructing algorithms directly in vector form, enabling seamless transitions between symbolic computation, simulation, and physical realization.

17 Examples

17.1 Numerical Example: Fully Explicit Matrix Computation

This subsection illustrates how a simple imperative program can be represented as both a minimal affine matrix transformation and as a fully expanded control–data system in the Vector Programming Language (VPL). This example demonstrates the expressive range of the Vector Evolution Machine (VEM) formalism.

Consider the simple loop:

```
x = 0;
while (x < 5) {
    x = x + 1;
}
```

We will describe two levels of matrix representation:

- A *minimal affine matrix form*, which captures only the evolution of data variables.
- A *VPL expanded matrix form*, which captures control and data in a single linear system.

17.1.1 Minimal Affine Matrix Form

In the minimal representation, the state vector is

$$\mathbf{v}_t = \begin{bmatrix} x_t \\ 1 \end{bmatrix}.$$

The loop body corresponds to the affine update

$$x_{t+1} = x_t + 1.$$

This is represented by the 2×2 affine transformation matrix

$$\mathbf{M}_{\text{min}} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

and the system evolves according to

$$\mathbf{v}_{t+1} = \mathbf{M}_{\text{min}} \mathbf{v}_t.$$

Starting from

$$\mathbf{v}_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

the iterations produce

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{v}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \mathbf{v}_3 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}, \mathbf{v}_4 = \begin{bmatrix} 4 \\ 1 \end{bmatrix}, \mathbf{v}_5 = \begin{bmatrix} 5 \\ 1 \end{bmatrix}.$$

The halting condition $x \geq 5$ is checked externally. This representation is compact and mathematically elegant, but it does not encode control flow.

17.1.2 VPL Expanded Control–Data Matrix Form

In the VPL model, both control and data are represented in the same linear space. We define the state vector as

$$\mathbf{V}_t = \begin{bmatrix} c_{\text{init}} \\ c_{\text{while}} \\ c_{\text{assign}} \\ c_{\text{print}} \\ c_{\text{exit}} \\ x \end{bmatrix},$$

where each c_i is a control activation variable for a control block, and the last element encodes the data variable x.

The system evolution is represented by

$$\mathbf{V}_{t+1} = \mathbf{M}_{\text{full}} \mathbf{V}_t + \mathbf{c}_{\text{full}},$$

with

$$\mathbf{M}_{\text{full}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \mathbf{c}_{\text{full}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

The halting condition is encoded phasorially as

$$\mathbf{C}_p = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \mathbf{c}_p = \begin{bmatrix} -5 \end{bmatrix},$$

corresponding to

$$\mathbf{C}_p \mathbf{V}_t + \mathbf{c}_p \geq 0,$$

which is equivalent to $x \geq 5$.

17.1.3 Step-by-Step Evolution in Expanded Form

Initial condition:

$$\mathbf{V}_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

t	c_{init}	c_{while}	c_{assign}	c_{print}	c_{exit}	x	Halt?
0	1	0	0	0	0	0	No
1	0	1	0	0	0	0	No
2	0	0	1	0	0	1	No
3	0	1	0	0	0	1	No
4	0	0	1	0	0	2	No
5	0	1	0	0	0	2	No
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
10	0	1	0	0	0	5	Yes

Table 8: Step-by-step state evolution in VPL expanded form.

Once the halting condition is satisfied, control moves to c_{exit} and the computation terminates.

17.1.4 Comparison of the Two Forms

Aspect	Minimal Matrix	VPL Expanded Matrix
Dimensionality	2×2	6×6
Encodes	Data update only	Control flow + data evolution + halting
Halting	External check	Internal phasorial condition
Suitability	Pure mathematical abstraction	Execution model and hardware mapping
Implementation	Simple	Moderate (with control wiring)
Expressive Power	Arithmetic evolution	Full program semantics

Table 9: Comparison between minimal and expanded matrix representations.

The minimal matrix provides a compact algebraic core of the computation, while the expanded matrix provides a fully executable representation suitable for VEM-based interpretation or compilation to hardware.

17.2 Toy Example: String Equality — `"Hello" == "Hello"`

To illustrate how VPL programs can be represented as vector evolution systems, we present a minimal but instructive example: testing the equality of

two strings,

$$\texttt{"Hello"} == \texttt{"Hello"}.$$

This example captures several essential elements of the VEM model: string encoding as data vectors, character-wise iteration through control evolution, and the use of a phasorial halting condition.

1. String Representation

We represent each character of the string `"Hello"` as a 7-bit ASCII binary vector. For simplicity, we concatenate the characters in sequence:

$$H\ (72),\ E\ (69),\ L\ (76),\ L\ (76),\ O\ (79).$$

Let each character be mapped to a 7-dimensional one-hot or binary vector. Concatenating for 5 characters gives us a $5 \times 7 = 35$ dimensional *data vector* for each string. Thus:

$$\mathbf{d}_1, \mathbf{d}_2 \in \mathbb{R}^{35},$$

correspond to the first and second string respectively. Since the two strings are identical, we have $\mathbf{d}_1 = \mathbf{d}_2$.

2. Control Flow Encoding

The equality check can be represented by a loop over character positions. In imperative pseudocode:

```
i = 0
while (i < 5 && s1[i] == s2[i]) {
    i = i + 1
}
equal = (i == 5)
```

In the VPL/VEM model, this is captured as a set of control states:

- C_0 — Initialization
- C_1 — Check condition ($i < 5 \wedge s1[i] == s2[i]$)
- C_2 — Increment index i
- C_3 — Success (equal)

- C_4 — Failure (not equal)

The *control vector* $\mathbf{c}(t)$ evolves through these states. Initially:

$$\mathbf{c}(0) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}^\top.$$

3. Transition Matrix for Control

The control transitions are encoded in the global control matrix $M_c \in \mathbb{R}^{5\times 5}$:

$$M_c = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

- $C_0 \to C_1$: initialization to first comparison. - $C_1 \to C_2$: if characters match and index $i < 5$. - $C_1 \to C_4$: if mismatch or end-of-string not reached properly. - $C_2 \to C_1$: loop back to check next character. - C_3 and C_4: terminal states.

4. Data Update and Comparison Phasor

The *phasor comparison* step is represented by

$$\mathbf{p}(i) = \mathbf{d}_1^{(i)} - \mathbf{d}_2^{(i)},$$

where $\mathbf{d}_1^{(i)}$ and $\mathbf{d}_2^{(i)}$ are the character vectors at position i.

The equality check uses the halting condition

$$\|\mathbf{p}(i)\| = 0 \quad \forall i \in \{0, \ldots, 4\},$$

meaning that all corresponding character vectors match. A mismatch triggers the $C_1 \to C_4$ transition.

We model this halting condition in matrix form with

$$C_p \cdot \mathbf{x} + c_p = 0,$$

where $\mathbf{x}$ is the full system vector and C_p selects the relevant positions for the phasor comparison.

5. Full System State

The complete system vector is the concatenation of control and data:

$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{c}(t) \\ \mathbf{d}_1 \\ \mathbf{d}_2 \\ i(t) \end{bmatrix}.$$

Here $i(t)$ represents the loop index, evolving linearly with each iteration.

6. Evolution Equation

The program execution corresponds to repeated application of a global linear operator M:

$$\mathbf{x}(t+1) = M\mathbf{x}(t) + \mathbf{c}.$$

Control flow evolves according to M_c, while $i(t)$ increments through a linear submatrix M_i:

$$M_i = [\ldots 1 \ldots],$$

encoding $i(t+1) = i(t) + 1$ when in C_2.

7. Halting Condition

The program halts when:

$$i(t) = 5 \quad \text{or} \quad \|\mathbf{p}(i)\| \neq 0.$$

If $i(t) = 5$, control enters C_3 and the strings are equal. If a mismatch occurs earlier, the system transitions to C_4.

8. Example Evolution

Starting at $t = 0$ in C_0, the evolution proceeds as follows:

$$\begin{aligned} \mathbf{c}(0) &= [1, 0, 0, 0, 0]^\top \\ \mathbf{c}(1) = M_c\mathbf{c}(0) &= [0, 1, 0, 0, 0]^\top \\ \mathbf{c}(2) = M_c\mathbf{c}(1) &= [0, 0, 1, 0, 0]^\top \\ \mathbf{c}(3) = M_c\mathbf{c}(2) &= [0, 1, 0, 0, 0]^\top \end{aligned}$$

$$\vdots$$

$$\mathbf{c}(6) = [0, 0, 0, 1, 0]^\top$$

At this step, since all characters matched and $i = 5$, the control enters C_3 (success state).

9. Remarks

This simple example shows how a traditional symbolic operation (string equality) can be encoded as vector evolution. Notably:

- The loop is represented by a *phasor recurrence* rather than imperative iteration.
- Control and data coexist in a single structured vector.
- Branching (success vs. failure) is represented as linear control transitions.
- The halting condition is encoded algebraically.

This structure generalizes naturally to more complex string operations (e.g., prefix matching, regular expression matching) and parallel string comparisons through superposition of control vectors.

17.3 Example: Two Parallel While Loops

Consider two independent counters that run in parallel:

```
x = 0;
y = 0;
while (x < 5) { x = x + 1; }
while (y < 5) { y = y + 1; }
```

This example illustrates two complementary views in VPL/VEM: a minimal data-centric affine model and a full expanded control–data model that encodes parallel control explicitly.

17.3.1 Minimal (data-only) affine model

If both loops are executed *in lock-step* (i.e., each logical "step" increments both counters simultaneously), we may capture the data evolution by the affine state vector

$$\mathbf{v}_t = \begin{bmatrix} x_t \\ y_t \\ 1 \end{bmatrix}.$$

The per-step affine update that increments both counters is

$$\mathbf{v}_{t+1} = \mathbf{M}_{\min}\,\mathbf{v}_t, \qquad \mathbf{M}_{\min} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

Starting from $\mathbf{v}_0 = [0, 0, 1]^T$ the iterations yield

$$\mathbf{v}_t = \begin{bmatrix} t \\ t \\ 1 \end{bmatrix},$$

and each counter reaches 5 at $t = 5$.

The minimal model is compact and useful for purely numeric reasoning. It does *not* encode which control blocks are active or the per-loop halting conditions internally; those checks are applied externally.

17.3.2 Expanded VPL Control–Data model

VPL requires an explicit encoding of control blocks so that independent control flows (and their possible interactions) are represented inside the same vector space. We choose a control decomposition with separate control blocks for the x loop and the y loop:

$$\mathbf{C} = \{c_{\text{init}},\ c_{x\_while},\ c_{x\_assign},\ c_{x\_exit},\ c_{y\_while},\ c_{y\_assign},\ c_{y\_exit}\}.$$

Form the global state

$$\mathbf{V}_t = \begin{bmatrix} \mathbf{c}_t \\ x_t \\ y_t \end{bmatrix} \in \mathbb{R}^9 \quad \text{with} \quad \mathbf{c}_t \in \mathbb{R}^7.$$

A natural block-structured transition matrix is

$$M_{\text{full}} = \begin{bmatrix} M_{CC} & 0_{7\times 2} \\ B_{2\times 7} & M_{DD} \end{bmatrix},$$

with the blocks described below.

Control–control block M_{CC} The control submatrix encodes local loop graphs for the two loops (here written conceptually; explicit numeric entries follow the same pattern as earlier single-loop examples):

$$M_{CC} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

(Interpretation: from `init` we activate both x and y while-blocks in parallel; x_while and y_while each drive their assign states and loop back.)

Data–data block M_{DD} We keep the data evolution affine by including a constant offset; the homogeneous data-only block is identity:

$$M_{DD} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Control-to-data coupling B **and offset** c_{full} The matrix B (size 2×7) encodes updates of x and y when the corresponding assign control is active. A simple choice is:

$$B = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

We then use a global affine offset vector to implement the unit increment (alternatively, the increments can be encoded in B with the constant column included in an augmented state). For clarity we place increments in a constant vector:

$$\mathbf{c}_{\text{full}} = \begin{bmatrix} \mathbf{0}_7 \\ 1 \\ 1 \end{bmatrix},$$

but this offset must be gated by the assign control activation (so in a full numeric realization the increment appears only when the appropriate assign control is active; in the matrix formalism the gating is represented by selecting rows/columns that combine control activation with the offset — the block form above is conceptual and straightforward to implement as sparse blocks).

Phasorial halting conditions Each loop has its own phasor condition selecting its data variable:

$$C_{p,x} = \begin{bmatrix} 0_{1\times 7} & 1 & 0 \end{bmatrix}, \quad c_{p,x} = -5,$$
$$C_{p,y} = \begin{bmatrix} 0_{1\times 7} & 0 & 1 \end{bmatrix}, \quad c_{p,y} = -5.$$

Loop x halts when $C_{p,x}\mathbf{V}_t + c_{p,x} \geq 0$ (i.e. $x \geq 5$), likewise for y.

17.3.3 Parallel activation and control superposition

Two alternative but equivalent interpretations of "parallel" execution in VPL:

1. **Structural parallelism (block-diagonal):** the global matrix naturally decomposes into two largely independent subgraphs for x and y, so they evolve simultaneously because M_{full} applies both updates at each step.

2. **Control superposition:** the control vector can be a linear combination that activates multiple control basis vectors at once, e.g.

   $$\mathbf{c}_t = \alpha\, \mathbf{e}_{x\_while} + \beta\, \mathbf{e}_{y\_while},$$

   with $\alpha = \beta = 1$ in the deterministic, synchronous case. The linearity of M then propagates both activations in parallel.

Both views are isomorphic in the linear-algebraic semantics: either the structure of M_{full} applies independent transitions to separate subspaces, or a superposed control vector triggers multiple transitions simultaneously.

17.3.4 Numeric step-by-step example (synchronous increments)

Assume synchronous operation where each step runs both assign updates when the assign states are active. Starting state:

$$\mathbf{V}_0 = [\,1, 0, 0, 0, 0, 0, 0, 0,\ 0,\ 0\,]^\top$$

(control at `init`, $x = 0, y = 0$).

If the pipeline activates both while-blocks after init, at each cycle both counters increment once (via assign states). The observable data evolution is:

t	c (active control blocks)	x_t	y_t
0	init	0	0
1	x_while, y_while	0	0
2	x_assign, y_assign	1	1
3	x_while, y_while	1	1
4	x_assign, y_assign	2	2
⋮	⋮	⋮	⋮
10	x_while (or exit)	5	5

Both counters reach 5 at the same logical time $t = 5$ (if they are perfectly synchronous). When x and y individually satisfy their phasor conditions, their control flows transition to their respective exit blocks.

17.3.5 Asynchronous or unbalanced loops

If bounds differ (e.g. $x < 7$ but $y < 5$), the same machinery handles it naturally: the phasor for y will trigger first, moving y's control into y_exit while x continues. This is encoded within M_{full} and the distinct $C_{p,\cdot}$ constraints — no special scheduler is required.

17.3.6 Hardware mapping implications

The block structure of M_{full} makes the parallel implementation straightforward:

- **Digital mapping:** separate logic/data paths for the x and y subsystems, with simple control signals gating the increment. The compiler can synthesize two small finite-state controllers and two adders/registers.

- **Analog mapping:** two parallel integrator/adder circuits driven by phasor-controlled switches; phasor thresholds implement comparisons $x \geq 5$, $y \geq 5$.
- **Quantum mapping (conceptual):** the two data registers occupy distinct subspaces (or qubit registers); the same unitary blocks implementing increments (or arithmetic modulo a value) can be applied in parallel (tensor product) where appropriate.

17.3.7 Remarks

This example shows how VPL/VEM captures true parallelism in the algebraic semantics: parallel loops are either represented as block-structured submatrices or as superposed control vectors. The phasor halting mechanism cleanly separates per-loop termination without centralized scheduling, enabling both synchronous and asynchronous behavior to be expressed uniformly.

17.4 Example: Graph Traversal

Graph algorithms are a natural fit for the VPL/VEM formalism because they can be expressed as repeated applications of adjacency transformations. Consider a simple directed graph

$$G = (V, E), \qquad V = \{A, B, C, D\},$$

with edges

$$E = \{(A, B),\ (A, C),\ (B, D),\ (C, D)\}.$$

17.4.1 Classical algorithmic description

A standard breadth-first traversal (BFS-like) starting from node A can be described as:

```
frontier = {A};
visited = {A};
while (frontier is not empty) {
    next = {};
    for each v in frontier:
```

```
        for each (v,u) in edges:
            if u not in visited:
                visited.add(u);
                next.add(u);
    frontier = next;
}
```

17.4.2 Graph as an adjacency matrix

The graph can be represented by the adjacency matrix

$$\mathbf{Adj} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

where rows and columns correspond to $[A, B, C, D]$.

17.4.3 State vector representation

Let the traversal state at time t be represented by the vector

$$\mathbf{f}_t \in \mathbb{R}^4,$$

where $\mathbf{f}_t[i] = 1$ if node i is currently in the frontier and 0 otherwise.

Start with

$$\mathbf{f}_0 = [\,1, 0, 0, 0\,]^T$$

(corresponding to node A).

At each iteration, the frontier is updated by

$$\mathbf{f}_{t+1} = \mathrm{clip}\big(\mathbf{Adj}^T \mathbf{f}_t\big),$$

where clip ensures binary occupancy (or, more generally, can be linear if multiplicities are allowed).

17.4.4 Full VPL encoding

To incorporate this traversal into the VPL formal model, we add explicit control blocks:

$$\mathcal{C} = \{c_{\text{init}},\ c_{\text{frontier\_step}},\ c_{\text{done}}\}.$$

The full state vector is

$$\mathbf{V}_t = \begin{bmatrix} \mathbf{c}_t \\ \mathbf{f}_t \end{bmatrix} \in \mathbb{R}^7,$$

where $\mathbf{c}_t \in \mathbb{R}^3$ (control) and $\mathbf{f}_t \in \mathbb{R}^4$ (frontier).

The global transition matrix is block-structured:

$$M_{\text{full}} = \begin{bmatrix} M_{CC} & 0_{3\times 4} \\ B_{4\times 3} & M_{DD} \end{bmatrix}.$$

- M_{CC} encodes the progression $c_{\text{init}} \to c_{\text{frontier\_step}} \to c_{\text{frontier\_step}} \to c_{\text{done}}$. - M_{DD} corresponds to the adjacency-based propagation (i.e. $M_{DD} = \mathbf{Adj}^T$). - B couples control state $c_{\text{frontier\_step}}$ to the application of M_{DD} (gated update). - $\mathbf{c}_{\text{full}} = 0$ (no offset needed here).

17.4.5 Phasor halting condition

Traversal halts when the frontier vector becomes empty. We can express this with a phasor halting condition

$$C_p = [\, 0_{1\times 3} \mid 1\ 1\ 1\ 1\,], \qquad c_p = 0,$$

and halting is triggered when $C_p\mathbf{V}_t + c_p = 0$.

17.4.6 Numeric execution trace

Starting from

$$\mathbf{V}_0 = [\,1,0,0,\ 1,0,0,0\,]^T$$

(init control, frontier on A):

t	Control block	Frontier vector $\mathbf{f}_t$	Active nodes
0	init	[1,0,0,0]	{A}
1	step	[0,1,1,0]	{B,C}
2	step	[0,0,0,1]	{D}
3	step	[0,0,0,0]	$\emptyset$
4	done	[0,0,0,0]	$\emptyset$

17.4.7 Parallelism and superposition

Multiple frontier nodes are naturally handled in parallel because the multiplication by $\mathbf{Adj}^T$ activates all successors simultaneously. No explicit loop over edges is needed — the matrix multiplication is the traversal.

This reflects the *parallel superposition principle* of VPL:

$$\mathbf{Adj}^T(\mathbf{f}_t^{(1)} + \mathbf{f}_t^{(2)}) = \mathbf{Adj}^T\mathbf{f}_t^{(1)} + \mathbf{Adj}^T\mathbf{f}_t^{(2)}.$$

17.4.8 Hardware mapping remarks

- **Digital**: The adjacency matrix can be implemented as a combinational logic block or lookup table that updates the frontier register each cycle.
- **Analog**: Each node can be represented by a voltage or current level; edges correspond to coupling elements (e.g., transconductance or switches), producing the next frontier in parallel.
- **Quantum**: A superposition of frontier states corresponds to a superposition of graph positions; controlled unitaries can implement the adjacency action, making this example close to continuous-time quantum walk models.

17.4.9 Summary

Graph traversal illustrates how:

1. classical iterative control can be collapsed into a single matrix iteration over a vector,
2. frontier propagation is inherently parallel in VPL,
3. phasor conditions provide an elegant and clean halting criterion,
4. and the resulting model maps cleanly to digital, analog, and even quantum hardware.

17.5 Example: Parallel Graph Traversal

Graph traversal is inherently well suited for VPL because traversal of multiple starting points can be expressed as linear superposition of state vectors. Consider again the directed graph

$$G = (V, E), \qquad V = \{A, B, C, D\},$$

with edges

$$E = \{(A, B),\ (A, C),\ (B, D),\ (C, D)\}.$$

17.5.1 Multiple traversal tasks

Suppose we want to traverse the graph from two different starting nodes simultaneously:

$$S^{(1)} = \{A\}, \qquad S^{(2)} = \{B\}.$$

In a classical implementation, this would typically require two separate traversals or explicit parallel threads. In VPL, we represent both traversals in the same **vector space**.

17.5.2 Adjacency matrix representation

The graph is encoded as

$$\mathbf{Adj} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

This matrix acts linearly on the frontier vectors.

17.5.3 Superposed initial state

We now initialize the frontier state vector as

$$\mathbf{f}_0 = \mathbf{f}_0^{(1)} + \mathbf{f}_0^{(2)} = [\,1, 1, 0, 0\,]^T,$$

where $\mathbf{f}_0^{(1)} = [1, 0, 0, 0]^T$ corresponds to A and $\mathbf{f}_0^{(2)} = [0, 1, 0, 0]^T$ corresponds to B.

17.5.4 Control structure and total state

The control blocks remain identical to the single traversal case:

$$\mathcal{C} = \{c_{\text{init}},\ c_{\text{frontier\_step}},\ c_{\text{done}}\}.$$

The total vector is

$$\mathbf{V}_t = \begin{bmatrix} \mathbf{c}_t \\ \mathbf{f}_t \end{bmatrix} \in \mathbb{R}^7,$$

and the transition matrix is

$$M_{\text{full}} = \begin{bmatrix} M_{CC} & 0 \\ B & \mathbf{Adj}^T \end{bmatrix}.$$

No change in control is needed to support multiple traversals — only the data subspace differs.

17.5.5 Execution trace

t	Control block	Frontier vector $\mathbf{f}_t$	Active nodes
0	init	[1, 1, 0, 0]	{A,B}
1	step	[0, 1, 1, 1]	{B,C,D}
2	step	[0, 0, 0, 1]	{D}
3	step	[0, 0, 0, 0]	$\emptyset$
4	done	[0, 0, 0, 0]	$\emptyset$

The action of $\mathbf{Adj}^T$ on a superposition of frontiers is equivalent to applying it to each frontier individually and summing the results:

$$\mathbf{Adj}^T\left(\mathbf{f}_t^{(1)} + \mathbf{f}_t^{(2)}\right) = \mathbf{Adj}^T\mathbf{f}_t^{(1)} + \mathbf{Adj}^T\mathbf{f}_t^{(2)}.$$

This is the algebraic essence of **parallelism in VPL**.

17.5.6 Phasor halting condition

The halting condition remains unchanged:

$$C_p = [\,0_{1\times 3} \mid 1\ 1\ 1\ 1\,], \qquad c_p = 0.$$

Traversal halts when no frontier node remains active.

17.5.7 Discussion: Parallelism without control overhead

- **Traditional view:** Multiple traversals require explicit thread scheduling, stack management, or separate data structures.
- **VPL view:** Multiple traversals correspond to the superposition of initial frontier vectors in the same data space. The same transition matrix handles them simultaneously.
- **Halting:** A single phasor condition cleanly handles the global stopping event.

17.5.8 Hardware mapping remarks

- **Digital**: Parallel traversal requires no extra control logic — multiple active nodes propagate simultaneously through the adjacency logic.
- **Analog**: Multiple initial currents or voltages correspond to multiple wavefronts propagating in parallel.
- **Quantum**: This superposition directly parallels a **quantum walk** starting in a superposed initial state.

17.5.9 Summary

This example shows that VPL parallelism naturally arises from linearity:

1. No replication of control structures is required.
2. Parallel traversal corresponds to a simple superposition of frontier vectors.
3. The computation unfolds in the same vector space, making the model highly scalable and efficient for graph algorithms.

17.6 Example: Bubble Sort in VPL

Sorting algorithms are ideal examples to demonstrate how iterative control structures and data updates are unified in the Vector Processing Language (VPL). In the case of the Bubble Sort, the algorithm repeatedly traverses a list, swapping adjacent elements that are out of order, until no swaps remain.

17.6.1 Classical pseudocode

```
int A[5] = {5, 3, 4, 1, 2};
int n = 5;
bool swapped = true;

while (swapped) {
    swapped = false;
    for (i = 0; i < n - 1; i++) {
        if (A[i] > A[i+1]) {
            temp = A[i];
            A[i] = A[i+1];
            A[i+1] = temp;
            swapped = true;
        }
    }
}
```

This structure combines:

- an outer `while` loop (conditional iteration),
- an inner `for` loop (deterministic iteration),
- conditional swapping (data-dependent control),
- and a global halting condition based on `swapped`.

17.6.2 VPL structural decomposition

The VPL compiler decomposes this into a set of control and data operators:

$$\mathcal{C} = \{\, c_{\text{init}},\ c_{\text{while}},\ c_{\text{for}},\ c_{\text{compare}},\ c_{\text{swap}},\ c_{\text{done}} \,\}.$$

Each block corresponds to a region of the control matrix M_C, while the data vector $\mathbf{x}$ contains both the array elements A_i and the flag variable `swapped`:

$$\mathbf{x} = [\, A_0, A_1, A_2, A_3, A_4, \texttt{swapped} \,]^T.$$

17.6.3 Control and data interaction

At each iteration step, the global VPL state evolves according to:

$$\mathbf{V}_{t+1} = M_{\text{full}}\,\mathbf{V}_t + \mathbf{c}_{\text{full}},$$

where

$$M_{\text{full}} = \begin{bmatrix} M_C & 0 \\ B & M_D \end{bmatrix},$$

and M_D encodes the conditional swap operations. For Bubble Sort, the local data update rule for two adjacent elements is:

$$M_{swap}(i, i+1) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

which is applied when $A_i > A_{i+1}$.

This can be represented as a piecewise affine transformation:

$$\mathbf{x}' = \begin{cases} S_i\mathbf{x} & \text{if } C_i(\mathbf{x}) = 1, \\ \mathbf{x} & \text{otherwise,} \end{cases}$$

where S_i is the swap operator and C_i is the comparison predicate. In matrix form, this corresponds to the application of a **conditional selection operator** $\Phi_i = C_i S_i + (1 - C_i)I$, which can be implemented via a block-matrix expression.

17.6.4 Phasor-based halting condition

The outer `while` loop is governed by the `swapped` flag, which is expressed as a phasor halting condition:

$$C_p = [\, 0_{1\times 5} \mid 1 \,], \quad c_p = 0.$$

The iteration stops when `swapped` $= 0$, i.e., no more exchanges are detected — equivalent to the phasor reaching steady phase.

17.6.5 Example of matrix-level execution

For a simple 3-element list $A = [3, 2, 1]$, the data subspace $\mathbf{x} \in \mathbb{R}^4$ evolves as follows:

$$\mathbf{x}_0 = [\,3, 2, 1, 1\,]^T, \qquad \mathbf{x}_{t+1} = M_D(\mathbf{x}_t)\mathbf{x}_t,$$

with

$$M_D(\mathbf{x}_t) = \begin{cases} \text{Swap}(0,1) & \text{if } A_0 > A_1, \\ \text{Swap}(1,2) & \text{if } A_1 > A_2. \end{cases}$$

After three iterations:

$$\mathbf{x}_3 = [\,1, 2, 3, 0\,]^T.$$

The halting phasor detects that `swapped` $= 0$ and terminates.

17.6.6 Expanded representation

The complete transition matrix for this system (for $n = 3$) expands to a 10×10 block matrix combining both control and data layers:

$$M_{\text{full}} = \left[\begin{array}{ccccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline B_{D1} & B_{D2} & B_{D3} & B_{D4} & B_{D5} & M_D & & & \end{array}\right].$$

Each B_{Di} block couples the control layer to the data comparison and swap logic.

17.6.7 Discussion

- The Bubble Sort in VPL becomes a **linear iterative system** with **conditional affine updates**.
- Nested loops are **unrolled into phasorial subspaces**; each inner loop iteration is represented as a repeated eigenvector transformation.
- The halting condition (no swap) is detected via **phasor cancellation**, equivalent to convergence in the vector subspace.
- The same formalism maps to hardware: in an analog form, each element comparison could correspond to a comparator circuit driving a multiplexer swap.

17.6.8 Conclusion

This example highlights how iterative, nested, and conditional control in classical algorithms are naturally expressed in VPL as block-linear transformations. Unlike procedural models, the entire system (control + data) evolves synchronously as a single vector dynamical system:

$$\mathbf{V}_{t+1} = M_{\text{full}}\mathbf{V}_t + \mathbf{c}_{\text{full}}.$$

Thus, even complex algorithms such as Bubble Sort become analyzable and executable using **linear algebra**, **phasorial convergence**, and **vector-space semantics**.

17.7 Example: Direct Problem Formulation — Parallel Graph Traversal

This worked example shows how to formulate a parallel graph traversal directly in the VPL formalism: state decomposition, operator construction, control (phasor) gating, explicit numeric evolution, and halting.

17.7.1 Problem statement

Let $G = (V, E)$ be the directed graph with vertex ordering $V = [A, B, C, D]$ and edge set

$$E = \{(A, B), (A, C), (B, D), (C, D)\}.$$

We want to perform a breadth-first-style traversal (parallel propagation) starting simultaneously from the two seeds $S^{(1)} = \{A\}$ and $S^{(2)} = \{B\}$. The goal is to compute the set of nodes reachable from either seed.

17.7.2 State vectors and decomposition

Define the frontier (active) vector and cumulative visited vector

$$\mathbf{f}_t = [f_A(t),\, f_B(t),\, f_C(t),\, f_D(t)]^\top \in \{0, 1\}^4,$$

$$\mathbf{g}_t = [g_A(t),\, g_B(t),\, g_C(t),\, g_D(t)]^\top \in \{0, 1\}^4,$$

with $f_v(t) = 1$ if node v is active at step t, and $g_v(t) = 1$ if v has ever been visited up to step t.

We also introduce a small control (phasor) vector $\mathbf{c}_t$ that gates the propagation step. For clarity we use a single-step control with two logical components ("step" and "done"):

$$\mathbf{c}_t = [c_{\text{step}}(t),\ c_{\text{done}}(t)]^\top.$$

The combined global state is

$$\mathbf{S}_t = \begin{bmatrix} \mathbf{c}_t \\ \mathbf{f}_t \\ \mathbf{g}_t \end{bmatrix} \in \mathbb{R}^{2+4+4} \ = \ \mathbb{R}^{10}.$$

17.7.3 Adjacency operator

The adjacency matrix A_G (rows = source, columns = target) for the chosen ordering $[A, B, C, D]$ is

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

Propagation from the frontier is effected by the transpose:

$$T \ = \ A_G^\top = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix},$$

so that the linear pre-threshold candidate frontier is $\tilde{\mathbf{f}}_{t+1} = T\mathbf{f}_t$.

17.7.4 VPL operator structure (conceptual)

We build a block operator M_{full} that applies propagation when the control phasor is in `step` and otherwise preserves state. A compact conceptual block structure is:

$$M_{\text{full}} = \begin{bmatrix} M_{cc} & 0 & 0 \\ B_{fc} & 0_{4\times4} & 0_{4\times4} \\ 0 & I_4 & I_4 \end{bmatrix}, \qquad \mathbf{C}_{\text{full}} = \mathbf{0},$$

where:

- M_{cc} advances control phases (e.g. from `step` to `done`); for our step-by-step simulation we will set $c_{\text{step}} = 1$ while propagation is active and $c_{\text{done}} = 0$ otherwise,
- B_{fc} injects the gated adjacency action: when the step phasor is active, B_{fc} applies T to the data frontier; concretely $B_{fc} = \begin{bmatrix} c_{\text{step}} \cdot T & 0 \end{bmatrix}$ (seen as column blocks),
- the lower-right blocks accumulate visited nodes: $\mathbf{g}_{t+1} = \mathbf{g}_t + \sigma(\tilde{\mathbf{f}}_{t+1})$ (where σ is element-wise saturation to $\{0, 1\}$).

For a transparent numeric trace we will apply the gated propagation manually per step (this avoids embedding the comparator/saturation nonlinearity into a single matrix).

17.7.5 Initial condition (parallel seeds)

Superpose the two seed frontiers into the initial frontier:

$$\mathbf{f}_0 = \mathbf{f}_0^{(1)} + \mathbf{f}_0^{(2)} = [1, 0, 0, 0]^\top + [0, 1, 0, 0]^\top = [1, 1, 0, 0]^\top.$$

Initialize visited equal to initial frontier:

$$\mathbf{g}_0 = \mathbf{f}_0 = [1, 1, 0, 0]^\top.$$

Initial control: set the step phasor active:

$$\mathbf{c}_0 = [1,\ 0]^\top \quad \text{(step active)}.$$

17.7.6 Numeric evolution (step-by-step)

We compute the evolution in discrete propagation steps. At each step:

$$\tilde{\mathbf{f}}_{t+1} = T\mathbf{f}_t, \qquad \mathbf{f}_{t+1} = \sigma(\tilde{\mathbf{f}}_{t+1}), \qquad \mathbf{g}_{t+1} = \mathbf{g}_t + \mathbf{f}_{t+1},$$

where $\sigma(\cdot)$ thresholds positive entries to 1 (here entries are integer counts so thresholding is trivial).

Step 0 → 1:

$$\mathbf{f}_0 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}.$$

Compute $\tilde{\mathbf{f}}_1 = T\mathbf{f}_0$:

$$T\mathbf{f}_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Threshold (saturation) gives

$$\mathbf{f}_1 = \sigma(\tilde{\mathbf{f}}_1) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Update visited:

$$\mathbf{g}_1 = \mathbf{g}_0 + \mathbf{f}_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 1 \end{bmatrix}.$$

(If we keep $\mathbf{g}$ binary we would saturate to $[1, 1, 1, 1]^\top$; numerical accumulation is useful for counting visits.)

Step 1 → 2: Take the saturated frontier as input (we use $\mathbf{f}_1 = [0, 1, 1, 1]^\top$):

$$\tilde{\mathbf{f}}_2 = T\mathbf{f}_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 2 \end{bmatrix}.$$

Threshold/saturate:

$$\mathbf{f}_2 = \sigma(\tilde{\mathbf{f}}_2) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Update visited (saturating to $\{0,1\}$):

$$\mathbf{g}_2 = \sigma(\mathbf{g}_1) + \mathbf{f}_2 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Step 2 → 3:

$$\tilde{\mathbf{f}}_3 = T\mathbf{f}_2 = T \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Threshold:

$$\mathbf{f}_3 = \mathbf{0}.$$

Visited remains $\mathbf{g}_3 = \mathbf{g}_2 = [1,1,1,1]^\top$.

Now the frontier is empty and propagation halts.

17.7.7 Phasor halting condition

We encode the halting test as a projection (phasor) over the frontier:

$$C_p \mathbf{S}_t + c_p = 0 \quad \Longleftrightarrow \quad \sum_{v \in V} f_v(t) = 0.$$

Concretely, using the global state ordering $\mathbf{S}_t = [c_{\text{step}}; c_{\text{done}}; f_A; f_B; f_C; f_D; g_A; g_B; g_C; g_D]$, a single-row C_p picks the frontier components:

$$C_p = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \qquad c_p = 0.$$

The halting criterion is $C_p \mathbf{S}_t = 0$ (no active frontier). When satisfied, the control phasor transitions to `done` and further propagation blocks are disabled.

17.7.8 Discussion: parallelism and superposition

Parallelism here is algebraic: the single initial vector $\mathbf{f}_0 = [1,1,0,0]^\top$ encodes two concurrent seeds. Because T is linear,

$$T(\mathbf{f}_0^{(1)} + \mathbf{f}_0^{(2)}) = T\mathbf{f}_0^{(1)} + T\mathbf{f}_0^{(2)},$$

so one matrix application propagates both traversals simultaneously. No explicit thread scheduling or interleaving is necessary; the vector-space representation does the concurrency for free.

17.7.9 Hardware mapping notes

- **Digital mapping:** Represent $\mathbf{f}$ and $\mathbf{g}$ as bit-vectors in registers; compute $T\mathbf{f}$ via sparse combinational logic (OR of incoming active bits per node), then threshold and update visited registers.
- **Analog mapping:** Represent node activations as voltages/currents; implement interconnects with weighted summing nodes that compute $\tilde{\mathbf{f}} = T\mathbf{f}$. Comparators implement thresholding; sample-and-hold elements implement discrete-step behavior.
- **Quantum mapping (conceptual):** Normalize T or derive a unitary analogue and perform unitary evolution (quantum walks). Measurement would project the amplitude vector and provide probabilistic visitation information.

17.7.10 Summary

The graph-traversal example demonstrates the VPL workflow:

1. encode the problem as linear operators (adjacency T) and data vectors $(\mathbf{f}, \mathbf{g})$,
2. initialize the frontier with a superposition of seeds,
3. iterate the gated linear propagation until the halting phasor (empty frontier) is reached,
4. read out the visited set via projection.

This direct problem-to-VPL formulation is compact, naturally parallel, and maps straightforwardly to digital, analog, or quantum hardware.

18 Applications

The Vector Programming Language (VPL) formalism is not limited to symbolic or software interpretation. Because of its algebraic nature, the model can be physically instantiated across diverse substrates that implement linear transformations of state vectors. This section explores three primary applications of the VPL model: (1) its mapping to electric and analog computational

systems, (2) its correspondence with quantum mechanical computation, and (3) its realization as a novel class of computational devices termed *Vector Evolution Machines* (VEMs).

18.1 Electric and Analog Mappings

VPL provides a direct bridge between abstract computation and circuit-level realization. Each control or data flow in a VPL program corresponds to a weighted connection between state variables, represented mathematically as the nonzero entries of the global transformation matrix M. This structure allows the construction of analog electrical circuits where voltages or currents represent vector components, and interconnections (resistors, capacitors, or operational amplifiers) embody the matrix coefficients.

Formally, a VPL state update,

$$\mathbf{S}_{t+1} = M\mathbf{S}_t + \mathbf{c},$$

can be realized as a dynamic network of weighted summing nodes, where:

- each element of $\mathbf{S}$ is represented by a circuit node potential (voltage),
- each nonzero coefficient M_{ij} corresponds to a conductance or gain linking node j to node i,
- and the bias term $\mathbf{c}$ corresponds to fixed voltage or current sources.

In continuous time, this becomes equivalent to a system of coupled differential equations,

$$\frac{d\mathbf{S}(t)}{dt} = A\mathbf{S}(t) + \mathbf{b},$$

where A and $\mathbf{b}$ are analog counterparts of M and $\mathbf{c}$. By proper scaling, feedback control, and saturation mechanisms, the analog network can execute iterative or conditional updates identical to discrete VPL semantics.

This analogy opens a pathway toward **analog computing architectures** that execute VPL programs natively through their inherent electrical dynamics, effectively treating programs as electrical topologies rather than as symbolic sequences.

18.2 Quantum Mappings

The mathematical structure of VPL naturally aligns with quantum computation, particularly through its vector-based state representation and linear transformation semantics. In the quantum domain, the VPL global state vector $\mathbf{S}$ is interpreted as a quantum state $|\psi\rangle$, and the transformation matrix M becomes a unitary operator U, ensuring conservation of probability amplitudes.

$$|\psi_{t+1}\rangle = U|\psi_t\rangle.$$

The key correspondence lies in the interpretation of control and data blocks:

- Control logic (branching, iteration) maps to conditional unitary operations or controlled gates.
- Data manipulation (assignments, arithmetic) maps to linear or tensor operations over quantum registers.
- Superposition and entanglement correspond to multi-dimensional couplings in the expanded VPL vector space.

Thus, a VPL program can be viewed as a generalized quantum circuit, where classical control flow becomes embedded within the linear operator structure itself. By enforcing unitarity constraints on M, one obtains quantum-coherent implementations directly from the same high-level VPL description. This dual interpretation positions VPL as a **bridge between classical algorithmic design and quantum circuit synthesis**.

18.3 Vector Evolution Machines (VEMs)

The culmination of the VPL formalism is the *Vector Evolution Machine* (VEM), a conceptual and implementable architecture where the state, control, and data are all unified as evolving components of a single vector space.

Unlike traditional machines that separate memory, control, and computation, the VEM operates as a dynamical system governed by an evolution law:

$$\mathbf{S}_{t+1} = M\mathbf{S}_t + \mathbf{c}.$$

Here, M acts as the machine's structural definition (analogous to both program and hardware), and $\mathbf{S}_t$ represents the complete computational state.

A VEM can be realized in multiple domains:

- **Digital VEM:** implemented as discrete matrix updates on digital processors, interpretable as general-purpose vector processors.
- **Analog VEM:** instantiated as continuous electrical systems or neuromorphic substrates, where time evolution is physical rather than symbolic.
- **Quantum VEM:** realized through unitary evolution, where matrix operations correspond to quantum gate sequences and superpositions.

These implementations share a common mathematical backbone, allowing hybrid architectures that mix digital precision, analog dynamics, and quantum superposition within the same theoretical framework.

The VEM therefore represents not only a unifying model of computation but also a **constructive foundation for post-von-Neumann machines**, where computation emerges as the evolution of structured vector states rather than as the sequential manipulation of symbolic instructions.

18.4 Unified Perspective on Physical Computation

The three domains discussed — electric, quantum, and vector-evolutional — reveal a common substrate for computation that transcends implementation details. In each case, the Vector Programming Language (VPL) model expresses computation as a transformation in a structured vector space, with distinct physical realizations corresponding to different interpretations of the global transformation matrix M.

- In **electrical and analog systems**, the entries of M are implemented as coupling strengths between circuit nodes, and the state evolution occurs through physical voltage or current propagation.
- In **quantum systems**, M becomes a unitary operator acting on a complex Hilbert space, and state evolution follows the Schrödinger dynamics.

- In **Vector Evolution Machines**, M defines a fully synthetic computational medium, either simulated or hardware-embedded, where digital and analog behaviors coexist under a shared algebraic law.

Thus, the VPL model provides a unifying mathematical structure that subsumes the classical, analog, and quantum computational paradigms under a single expression of state evolution:

$$\mathbf{S}_{t+1} = M\mathbf{S}_t + \mathbf{c}.$$

This abstraction demonstrates that what traditionally distinguishes different forms of computation — logical discreteness, analog continuity, or quantum superposition — can all be seen as manifestations of the same algebraic principle with varying constraints on M:

- $M \in \mathbb{R}^{n\times n}$ with arbitrary weights for analog systems.
- $M \in \{0,1\}^{n\times n}$ for purely logical or digital machines.
- $M \in \mathbb{C}^{n\times n}$ and $M^\dagger M = I$ for quantum systems.

In this sense, the Vector Programming Language offers not merely a new programming formalism but a universal algebraic representation of computation itself — capable of unifying hardware architectures, physical dynamics, and algorithmic semantics within a single mathematical framework. This realization opens the path toward a *generalized theory of physical computation*, where programs, machines, and physical laws can be expressed through the same vectorial formalism.

19 Electrical Circuit Mappings

20 Quantum Circuit Mappings

21 Simulation

22 Discussion

23 Conclusion

24 Future Work

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.

25 Appendix A

25.1 Linear affine operator