Vibesim Technical Documentation

Vibesim Development Team

February 19, 2026

Contents

1 Overview

Vibesim is a web-based control system simulation tool that provides comprehensive loop detection, numerical integration, and solver functionality. This document details the core technical implementation of Vibesim, including loop detection algorithms, numerical integration methods, and solver design.

2 Loop Detection Algorithm

2.1 Algorithm Background

In control systems, a feedback loop refers to a path where a signal starts from a node, passes through a series of processing blocks, and returns to the same node. Identifying these loops is crucial for:

- System stability analysis
- Controller design
- System performance evaluation

2.2 Algorithm Design Philosophy

Vibesim's loop detection algorithm is based on the following design principles:

- **Sum block centered**: Control system feedback is typically implemented through sum blocks
- **Bidirectional traversal**: Determine loop composition through forward and backward traversal
- **Precise localization**: Not only detect loop existence but also identify specific nodes in the loop

2.3 Algorithm Flowchart

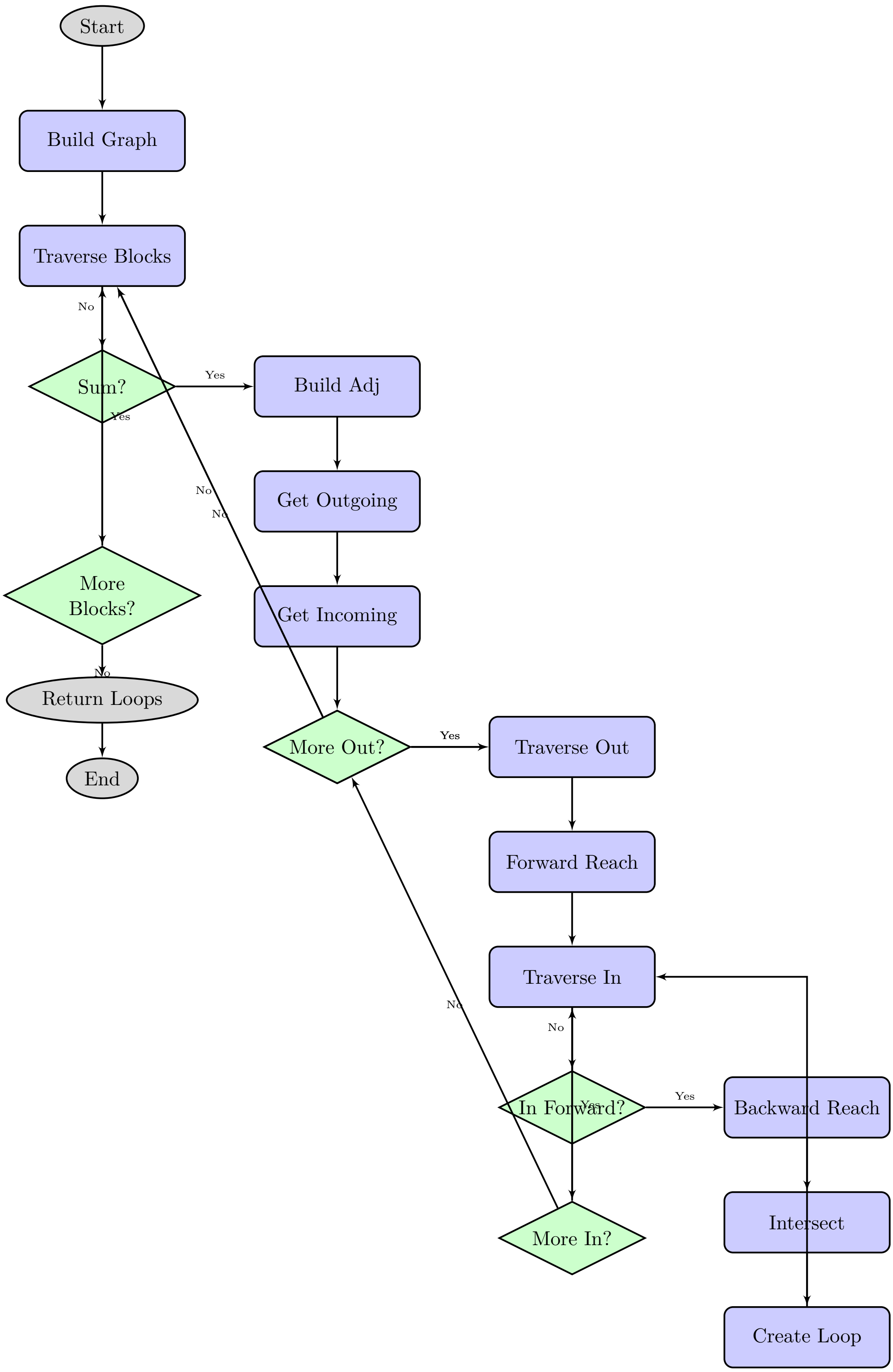

Figure 1: Loop Detection Algorithm Flowchart

2.4 Algorithm Steps

2.4.1 Data Structure Construction

Convert control system blocks to graph nodes and connections to directed edges:

```
const blocks = Array.from(state.blocks.values()).map((block) => ({
  id: block.id,
  type: block.type,
  params: block.params || {},
}));

const connections = state.connections.map((conn) => ({
  from: conn.from,
  to: conn.to,
  fromIndex: conn.fromIndex ?? 0,
  toIndex: conn.toIndex ?? 0,
}));
```

Listing 1: Data Structure Construction

2.4.2 Graph Traversal Function

Use Depth-First Search (DFS) to traverse the graph:

```
const traverse = (startId, adj, sumId) => {
  const visited = new Set();
  const stack = [startId];
  while (stack.length) {
    const id = stack.pop();
    if (visited.has(id)) continue;
    visited.add(id);
    (adj.get(id) || []).forEach((next) => {
      if (next === sumId) return;
      stack.push(next);
    });
  }
  return visited;
};
```

Listing 2: Graph Traversal Function

Time Complexity: O(V + E), where V is the number of nodes and E is the number of edges.

2.4.3 Loop Detection Main Logic

1. Filter Sum Blocks

```
const loops = [];
blocks.forEach((block) => {
  if (block.type !== "sum") return;
  const sumId = block.id;
  // ... subsequent processing
});
```

2. Build Adjacency List

```
const signs = Array.isArray(block.params?.signs) ? block.params.signs : [];
const forward = new Map();
const backward = new Map();

blocks.forEach((node) => {
  forward.set(node.id, []);
  backward.set(node.id, []);
```

```javascript
});

connections.forEach((conn) => {
  if (!forward.has(conn.from) || !forward.has(conn.to)) return;
  if (conn.from === sumId || conn.to === sumId) return;
  forward.get(conn.from).push(conn.to);
  backward.get(conn.to).push(conn.from);
});
```

Listing 3: Build Adjacency List

3. Bidirectional Traversal to Determine Loops

```javascript
const outgoing = connections.filter((conn) => conn.from === sumId);
const incoming = connections.filter((conn) => conn.to === sumId);

if (!outgoing.length || !incoming.length) return;

outgoing.forEach((outConn) => {
  const forwardReach = traverse(outConn.to, forward, sumId);

  incoming.forEach((inConn) => {
    if (!forwardReach.has(inConn.from)) return;

    const backwardReach = traverse(inConn.from, backward, sumId);

    const activeIds = new Set(
      Array.from(forwardReach).filter((id) => backwardReach.has(id))
    );

    activeIds.add(outConn.to);
    activeIds.add(inConn.from);
    activeIds.delete(sumId);

    const feedbackSign = Number(signs[inConn.toIndex ?? 0] ?? 1) || 0;

    const key = `${sumId}:${outConn.to}:${outConn.fromIndex ?? 0}->${inConn.from}:${inConn.toIndex ?? 0}`;
    loops.push({
      key,
      sumId,
      outConn,
      inConn,
      activeIds,
      feedbackSign,
    });
  });
});
```

Listing 4: Bidirectional Traversal

2.5 Algorithm Complexity Analysis

2.5.1 Time Complexity

- Build adjacency list: O(V + E)
- Traverse sum blocks: O(V)
- Processing each sum block:
  - Forward traversal: O(V + E)
  - Backward traversal: O(V + E)
  - Intersection calculation: O(V)

- **Total Complexity**: O(n Œ (V + E)), where n is the number of sum blocks

2.5.2 Space Complexity

- Adjacency list storage: O(V + E)
- Reachable set storage: O(V)
- Loop storage: O(n Œ V)
- **Total Complexity**: O(V + E + n Œ V)

2.6 Algorithm Characteristics

2.6.1 Advantages

1. **Precise loop localization**: Not only know loop exists but also identify specific nodes
2. **Multi-loop support**: Can detect all feedback loops in the system
3. **Preserve loop information**: Record input/output connections and feedback signs
4. **Control system oriented**: Specifically designed for control system feedback structures
5. **High efficiency**: Very efficient for sparse graphs

2.6.2 Limitations

1. **Depends on sum blocks**: Only detects loops formed through sum blocks
2. **No algebraic loop handling**: Does not detect algebraic loops not involving sum blocks
3. **Assumes directed graph**: Assumes control system is a directed graph, does not handle bidirectional connections

2.7 Comparison with Other Algorithms

Table 1: Comparison with Kahn's Algorithm

Feature	Kahn's Algorithm	Vibesim Algorithm
Main Purpose	Topological Sort	Loop Detection and Localization
Detection Method	Remove nodes with indegree 0	Bidirectional Traversal Intersection
Loop Information	Only know existence	Precise localization of composition
Time Complexity	O(V+E)	O(nŒ(V+E))
Applicable Scenario	Directed Acyclic Graph	Control System Feedback Loop

Table 2: Comparison with Standard DFS Loop Detection

Feature	Standard DFS	Vibesim Algorithm
Detection Method	Recursive stack detection	Bidirectional Traversal Intersection
Loop Information	Stop when found	Complete traversal of all loops
System Specificity	General graph algorithm	Control system specific
Feedback Sign	Not handled	Record feedback sign

2.8 Example Analysis

2.8.1 Simple Feedback System

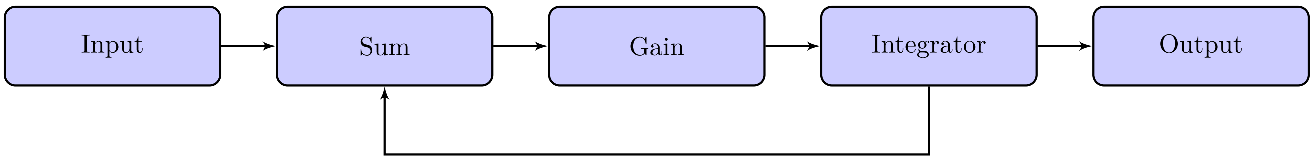

Figure 2: Simple Feedback System

Algorithm Execution Process:

1. Identify sum block
2. Outgoing connection: Sum Gain
3. Incoming connection: Integrator Sum
4. Forward traversal: {Gain, Integrator}
5. Backward traversal: {Integrator, Gain}
6. Intersection: {Gain, Integrator}
7. Loop detected

3 Solver Details

3.1 Overview

Vibesim solver is responsible for numerical simulation of control systems, including integration calculation, algorithm selection, and loop convergence handling. The solver uses a phased processing approach to ensure numerical accuracy and efficiency.

3.2 Integration Processing

3.2.1 Main Integration Algorithm: RK4 (Fourth-Order Runge-Kutta)

Vibesim primarily uses the RK4 algorithm for numerical integration, which is one of the most commonly used numerical integration methods with high accuracy and good stability.

Basic RK4 Implementation

```
export const integrateRK4 = (state, input, dt) => {
  const k1 = input;
  const k2 = input;
  const k3 = input;
  const k4 = input;
  return state + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4);
};
```

Listing 5: Basic RK4 Implementation

Description:

- For pure integrators (input directly as derivative), RK4 simplifies to the above form
- `state`: Current state value
- `input`: Input value (i.e., derivative)
- `dt`: Time step
- Returns: Next time step state value

Transfer Function RK4 Implementation

```
export const integrateTfRK4 = (model, state, input, dt) => {
  if (model.n === 0) return state;
  const k1 = stateDerivative(model, state, input);
  const k2 = stateDerivative(model, addVec(state, scaleVec(k1, dt / 2)), input);
  const k3 = stateDerivative(model, addVec(state, scaleVec(k2, dt / 2)), input);
  const k4 = stateDerivative(model, addVec(state, scaleVec(k3, dt)), input);
  const sum = addVec(addVec(k1, scaleVec(k2, 2)), addVec(scaleVec(k3, 2), k4));
  return addVec(state, scaleVec(sum, dt / 6));
};
```

Listing 6: Transfer Function RK4 Implementation

Description:

- `model`: State space model of transfer function
- `state`: Current state vector
- `input`: Input value

- `dt`: Time step
- Uses complete RK4 four-step calculation

3.2.2 Integrator Block Implementation

The integrator block is the most basic continuous-time block in control systems, implemented as follows:

```
integrator: {
  init: (ctx, block) => {
    const params = ctx.resolvedParams.get(block.id) || {};
    const state = getBlockState(ctx, block);
    const min = resolveLimit(params.min, -Infinity);
    const max = resolveLimit(params.max, Infinity);
    const initial = Number(params.initial) || 0;
    state.integrator = clampValue(initial, min, max);
  },
  output: (ctx, block) => {
    const state = getBlockState(ctx, block);
    const prev = state.integrator ?? 0;
    ctx.outputs.set(block.id, prev);
  },
  update: (ctx, block) => {
    const params = ctx.resolvedParams.get(block.id) || {};
    const min = resolveLimit(params.min, -Infinity);
    const max = resolveLimit(params.max, Infinity);
    const inputVal = getInputValue(ctx, block, 0, 0);
    const state = getBlockState(ctx, block);
    const prev = state.integrator ?? 0;
    const next = integrateRK4(prev, inputVal ?? 0, ctx.dt);
    state.integrator = clampValue(next, min, max);
  },
}
```

Listing 7: Integrator Block Implementation

Features:

1. **Clamping support**: Can set minimum and maximum values for integrator
2. **Initial conditions**: Supports setting initial values for integrator
3. **Three-phase processing**:
   - `init`: Initialize state
   - `output`: Output current state
   - `update`: Update state using RK4

3.2.3 RK4 Algorithm Advantages

Table 3: RK4 Algorithm Advantages

Feature	Description
High Accuracy	Fourth-order accuracy, error is $O(dt^5)$
Stability	Good stability for most systems
Efficiency	Requires 4 derivative calculations per step
Widely Used	Most commonly used numerical integration method in engineering

3.3 Algorithm Selection Strategy

3.3.1 Fixed Algorithm Strategy

Vibesim **does not provide user-selectable integration algorithms**, but instead uses a fixed algorithm strategy based on block types. This design simplifies user operations while ensuring sufficient numerical accuracy.

3.3.2 Continuous-Time System Algorithms

Integrator

- **Algorithm**: Simplified RK4
- **Reason**: Pure integrator, derivative directly equals input
- **Code**: `integrateRK4`

Transfer Function

- **Algorithm**: Complete RK4 (state space form)
- **Reason**: Need to handle state vectors and matrix operations
- **Code**: `integrateTfRK4`

State Space

- **Algorithm**: Forward Euler
- **Reason**: Simple first-order system

```
const xNext = prev + ctx.dt * (A * prev + B * (inputVal ?? 0));
state.stateSpaceX = xNext;
const y = C * xNext + D * (inputVal ?? 0);
```

PID Controller

- **Algorithm**: Forward Euler (integral part)
- **Reason**: Simple method sufficient for integral term

```
const nextIntegral = pid.integral + (inputVal ?? 0) * ctx.dt;
const clampedIntegral = clampValue(nextIntegral, min, max);
const derivative = ((inputVal ?? 0) - pid.prev) / Math.max(ctx.dt, 1e-6);
const out = kp * (inputVal ?? 0) + ki * clampedIntegral + kd * derivative;
```

Low/High Pass Filter (LPF/HPF)

- **Algorithm**: Forward Euler
- **Reason**: First-order filter, simple method sufficient

```
const wc = 2 * Math.PI * fc;
const next = prev + ctx.dt * wc * ((inputVal ?? 0) - prev);
```

Derivative

- **Algorithm**: Finite Difference
- **Reason**: Derivative requires discretization

```
const out = ((inputVal ?? 0) - prev) / Math.max(ctx.dt, 1e-6);
```

3.3.3 Discrete-Time System Algorithms

Zero-Order Hold (ZOH)

- **Algorithm**: Piecewise constant
- **Feature**: Holds last sampled value during sampling interval

```
if (ctx.t + 1e-6 >= state.nextTime) {
  state.lastSample = inputVal ?? 0;
  state.nextTime = ctx.t + ts;
}
```

First-Order Hold (FOH)

- **Algorithm**: Linear interpolation
- **Feature**: Linear interpolation during sampling interval

```
const slope = (state.lastSample - state.prevSample) / ts;
const out = state.lastSample + slope * (ctx.t - state.lastTime);
```

Discrete Transfer Function (DTF)

- **Algorithm**: Difference equation
- **Feature**: Uses input and output history values

```
let y = 0;
for (let i = 0; i < num.length; i += 1) {
  y += (num[i] || 0) * (xHist[i] || 0);
}
for (let i = 1; i < den.length; i += 1) {
  y -= (den[i] || 0) * (yHist[i - 1] || 0);
}
```

Discrete Delay (DDelay)

- **Algorithm**: Queue implementation
- **Feature**: Uses queue to store history values

```
state.queue.push(inputVal ?? 0);
while (state.queue.length > steps) state.queue.shift();
state.lastOut = state.queue[0] ?? 0;
```

3.3.4 Algorithm Selection Flowchart

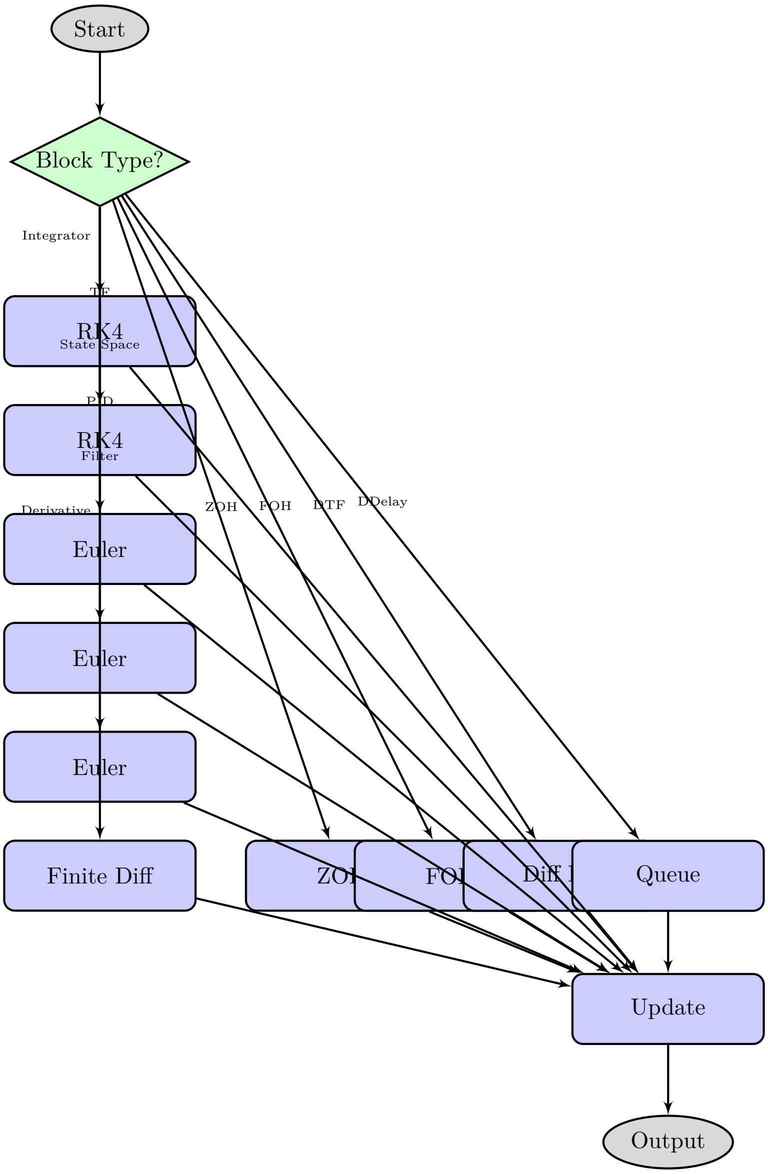

Figure 3: Algorithm Selection Flowchart

3.3.5 Algorithm Selection Summary Table

Table 4: Algorithm Selection Summary

Block Type	Algorithm	Accuracy	Use Case
Integrator	RK4	$O(dt^5)$	Continuous-time integration
Transfer Function	RK4	$O(dt^5)$	Continuous-time dynamics
State Space	Forward Euler	$O(dt^2)$	First-order system
PID	Forward Euler	$O(dt^2)$	Controller
LPF/HPF	Forward Euler	$O(dt^2)$	Filter
Derivative	Finite Difference	$O(dt)$	Derivative calculation
ZOH	Piecewise Constant	Exact	Discrete hold
FOH	Linear Interpolation	Exact	Discrete hold
DTF	Difference Equation	Exact	Discrete transfer function
DDelay	Queue	Exact	Discrete delay

3.4 Loop Convergence Processing

3.4.1 Algebraic Loop Concept

Algebraic loops refer to pure algebraic dependency loops that do not involve integrators or delays, for example:

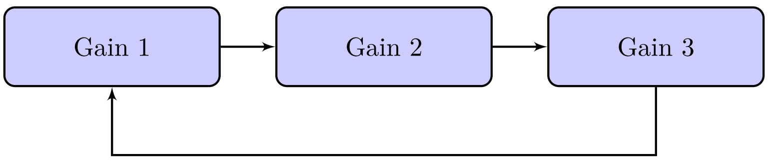

Figure 4: Algebraic Loop Example

Such loops cannot be solved through time progression and require special convergence algorithms.

3.4.2 Algebraic Loop Detection

Topological Sort (Kahn's Algorithm)

```
function buildAlgebraicPlan(algebraicBlocks, inputMap) {
  const byId = new Map();
  algebraicBlocks.forEach((entry) => byId.set(entry.block.id, entry));
  if (byId.size === 0) return { ordered: [], hasCycle: false };

  const indegree = new Map();
  const outEdges = new Map();
  byId.forEach((_, id) => {
    indegree.set(id, 0);
    outEdges.set(id, new Set());
  });

  byId.forEach((_, targetId) => {
    const inputs = inputMap.get(targetId) || [];
    inputs.forEach((srcKey) => {
      const srcId = sourceBlockIdFromKey(srcKey);
      if (!srcId || !byId.has(srcId) || srcId === targetId) return;
      const edges = outEdges.get(srcId);
      if (edges.has(targetId)) return;
```

```
      edges.add(targetId);
      indegree.set(targetId, (indegree.get(targetId) || 0) + 1);
    });
  });

  const queue = [];
  indegree.forEach((deg, id) => {
    if (deg === 0) queue.push(id);
  });
  const ordered = [];
  let readIdx = 0;
  while (readIdx < queue.length) {
    const id = queue[readIdx];
    readIdx += 1;
    ordered.push(byId.get(id));
    outEdges.get(id).forEach((neighbor) => {
      const next = (indegree.get(neighbor) || 0) - 1;
      indegree.set(neighbor, next);
      if (next === 0) queue.push(neighbor);
    });
  }

  const hasCycle = ordered.length !== byId.size;
  return { ordered: hasCycle ? algebraicBlocks : ordered, hasCycle };
}
```

Listing 8: Topological Sort

Description:

- If topological sort succeeds (`ordered.length === byId.size`), there is no loop
- If topological sort fails, an algebraic loop exists
- Returns sorted block list and loop flag

3.4.3 Algebraic Loop Convergence Algorithm

Fixed-Point Iteration

```
if (run.needsAlgebraicSolve) {
  if (!run.hasLabelResolution && !run.algebraicPlan.hasCycle) {
    run.algebraicPlan.ordered.forEach(({ block, handler }) => {
      handler.algebraic(run.ctx, block);
    });
  } else {
    let progress = true;
    let iter = 0;
    const maxIter = 50;
    while (progress && iter < maxIter) {
      iter += 1;
      progress = false;

      if (run.hasLabelResolution && resolveLabelSources())
        progress = true;

      run.algebraicPlan.ordered.forEach(({ block, handler }) => {
        const result = handler.algebraic(run.ctx, block);
        if (result?.updated) progress = true;
      });

      if (run.hasLabelResolution && resolveLabelSources())
        progress = true;
    }

    if (progress && iter >= maxIter) {
      run.algebraicLoopFailed = true;
      run.algebraicLoopTime = t;
      return false;
    }
```

```
  }
}
```

Listing 9: Fixed-Point Iteration

Convergence Conditions:

- All algebraic block values no longer change
- Or reach maximum iteration count (50 times)

3.4.4 Algebraic Block Processing Example

Transfer Function (Zero-Order)

```
algebraic: (ctx, block) => {
  const state = getBlockState(ctx, block);
  const model = state.tfModel;
  if (!model || model.n !== 0) return null;
  const inputVal = getInputValue(ctx, block, 0, 0);
  const out = outputFromState(model, model.state || [], inputVal ?? 0);
  const prev = ctx.outputs.get(block.id);
  ctx.outputs.set(block.id, out);
  return { updated: prev !== out && !(Number.isNaN(prev) && Number.isNaN(out)) };
}
```

Listing 10: Transfer Function Algebraic Processing

Description:

- For zero-order transfer functions (pure gain), use algebraic processing
- Returns flag indicating whether value was updated
- Used for detecting convergence state

3.4.5 Loop Convergence Flowchart

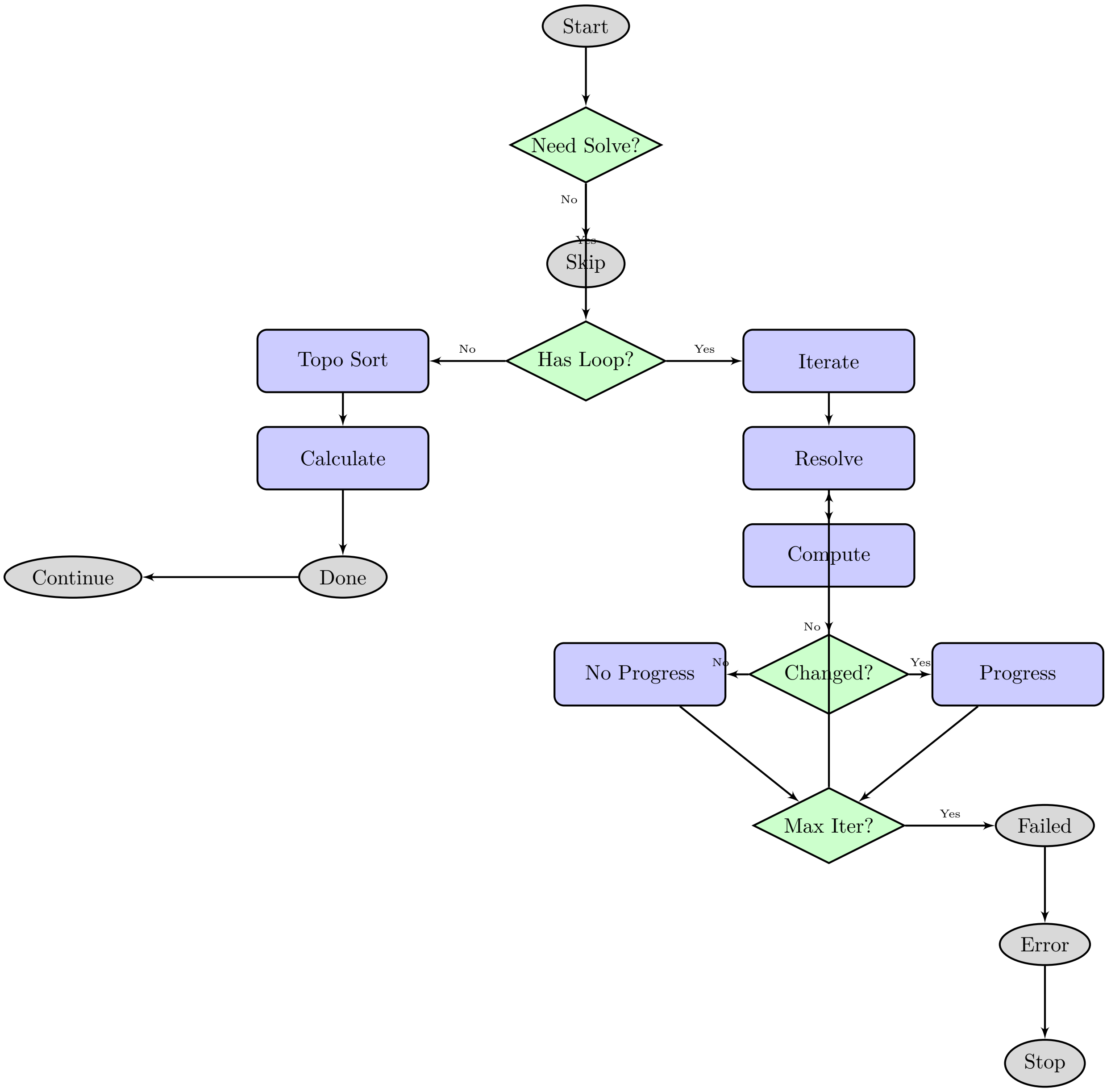

Figure 5: Loop Convergence Flowchart

3.4.6 Loop Convergence Characteristics

Table 5: Loop Convergence Characteristics

Feature	Description
Iteration Method	Fixed-point iteration
Convergence Detection	Check if values are stable
Maximum Iterations	50 times
No Loop Optimization	Use topological sort, single calculation
Loop Processing	Iterate until convergence or timeout
Error Handling	Report specific time point when not converged
Label Support	Support label connections between subsystems

3.5 Solver Flow

3.5.1 Complete Solver Flow

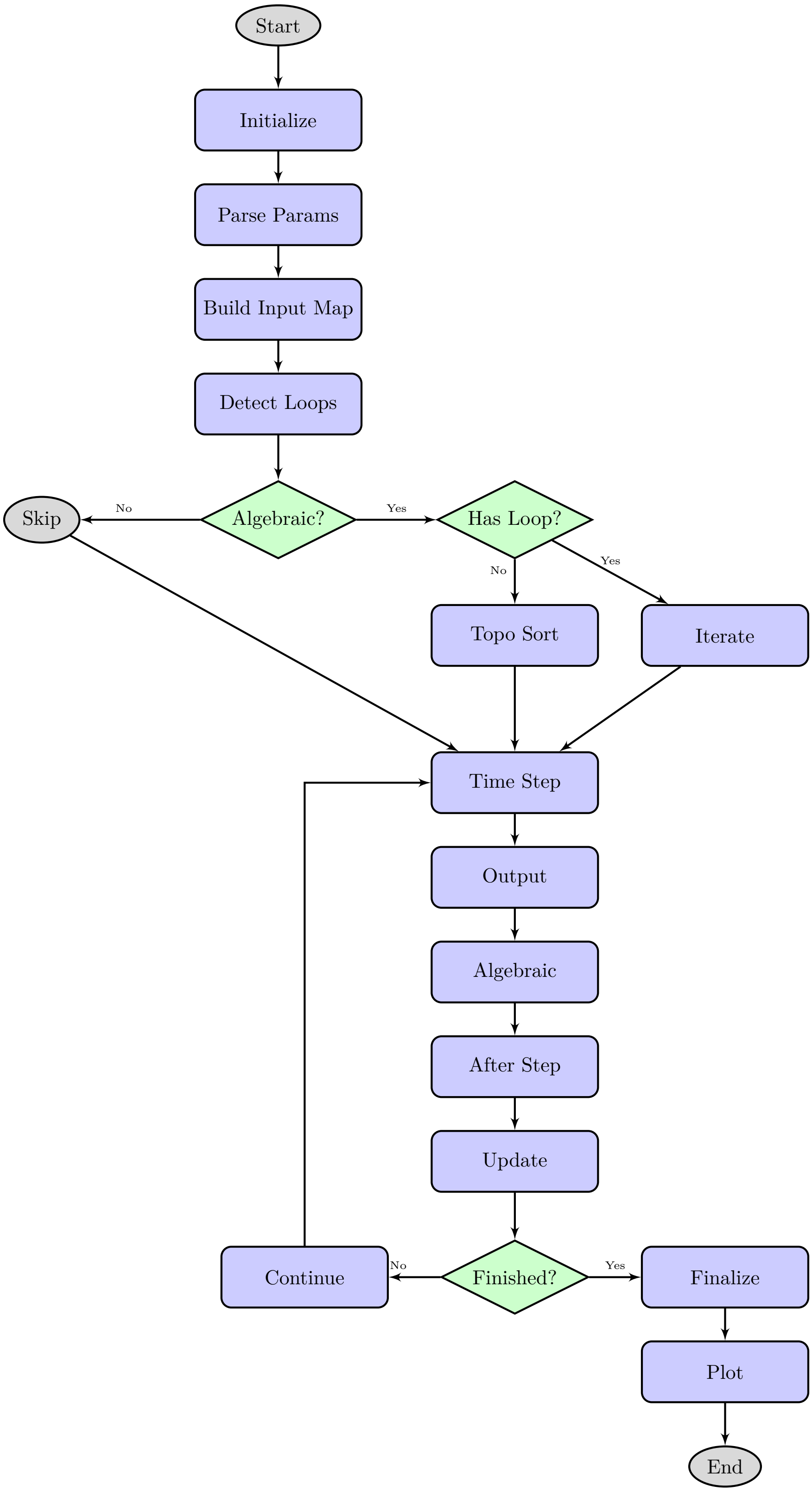

Figure 6: Complete Solver Flow

3.5.2 Time Step Loop

```
const runStep = (i) => {
  const t = i * run.dt;
  run.time.push(t);
  const outputs = new Map();
  run.ctx.t = t;
  run.ctx.outputs = outputs;

  run.outputBlocks.forEach(({ block, handler }) =>
    handler.output(run.ctx, block)
  );

  if (run.needsAlgebraicSolve) {
    if (!run.hasLabelResolution && !run.algebraicPlan.hasCycle) {
      run.algebraicPlan.ordered.forEach(({ block, handler }) => {
        handler.algebraic(run.ctx, block);
      });
    } else {
      let progress = true;
      let iter = 0;
      const maxIter = 50;
      while (progress && iter < maxIter) {
        iter += 1;
        progress = false;
        if (run.hasLabelResolution && resolveLabelSources())
          progress = true;
        run.algebraicPlan.ordered.forEach(({ block, handler }) => {
          const result = handler.algebraic(run.ctx, block);
          if (result?.updated) progress = true;
        });
        if (run.hasLabelResolution && resolveLabelSources())
          progress = true;
      }
      if (progress && iter >= maxIter) {
        run.algebraicLoopFailed = true;
        run.algebraicLoopTime = t;
        return false;
      }
    }
  }

  run.afterStepBlocks.forEach(({ block, handler }) =>
    handler.afterStep(run.ctx, block)
  );

  run.updateBlocks.forEach(({ block, handler }) =>
    handler.update(run.ctx, block)
  );

  return !run.algebraicLoopFailed;
};
```

Listing 11: Time Step Loop

3.5.3 Phased Processing

Vibesim divides each time step into four phases:

1. Output Phase

- Calculate current values of all output blocks
- Provide input for algebraic solving

2. **Algebraic Solving Phase**
   - Solve algebraic blocks (gain, zero-order transfer functions, etc.)
   - Handle algebraic loop convergence
   - Resolve label sources
3. **After-Step Phase**
   - Execute post-step processing
   - Such as delay block buffer updates
4. **Update Phase**
   - Update states of dynamic blocks (integrators, transfer functions, etc.)
   - Use RK4 and other algorithms for numerical integration

3.6 Performance Characteristics

3.6.1 Numerical Accuracy

Table 6: Numerical Accuracy Comparison

Algorithm	Local Truncation Error	Global Error	Stability
RK4	$O(dt^5)$	$O(dt^4)$	Good
Forward Euler	$O(dt^2)$	O(dt)	Conditionally stable
Finite Difference	O(dt)	O(dt)	Potentially unstable
Discrete Methods	Exact	Exact	Completely stable

3.6.2 Computational Efficiency

Table 7: Computational Efficiency Comparison

Block Type	Per-Step Computation	Complexity
Integrator	1 addition	O(1)
Transfer Function	n^2 operations	$O(n^2)$
State Space	n operations	O(n)
PID	3 operations	O(1)
Filter	1 operation	O(1)
Derivative	1 operation	O(1)
ZOH/FOH	1 assignment	O(1)
DTF	m+n operations	O(m+n)
DDelay	1 queue operation	O(1)

3.6.3 Convergence Performance

Table 8: Convergence Performance Comparison

Scenario	Convergence Speed	Maximum Iterations
No Loop	1 time	1 time
Simple Loop	Typically < 10 times	50 times
Complex Loop	May need more	50 times
Not Convergent	Not convergent	50 times (error)

4 Summary

4.1 Solver Design Philosophy

Vibesim solver embodies the following design philosophy:

1. **Usability First**: Users don't need to select complex algorithms
2. **Performance Balance**: Different block types use appropriate algorithms
3. **Robustness**: Handle complex situations like algebraic loops
4. **Accuracy Guarantee**: Key blocks use high-precision RK4 algorithm
5. **Error Handling**: Clear error reporting and stopping mechanisms

4.2 Core Technologies

1. **RK4 Integration**: High-precision numerical integration
2. **Fixed-Point Iteration**: Algebraic loop convergence
3. **Topological Sort**: Loop-free algebraic solving
4. **Phased Processing**: Clear solving flow
5. **State Management**: Efficient block state storage

4.3 Applicable Scenarios

- **Control System Design**: PID, state feedback, etc.
- **System Simulation**: Continuous and discrete-time systems
- **Filter Design**: LPF, HPF, etc.
- **Dynamic System Analysis**: Transfer functions, state space
- **Teaching Demonstrations**: Control theory teaching and experiments

4.4 Limitations

1. **Fixed Algorithms**: Users cannot select other integration algorithms
2. **Convergence Limit**: Algebraic loops maximum 50 iterations
3. **Accuracy Limit**: RK4 may not be sufficient for some stiff systems
4. **Fixed Step Size**: Does not support adaptive step size

5 References

- **RK4 Algorithm**: Classical fourth-order Runge-Kutta method
- **Topological Sort**: Kahn's algorithm for loop detection
- **Fixed-Point Iteration**: Standard method for algebraic loop convergence
- **Numerical Integration**: Basic theory of control system numerical simulation
- **Vibesim Source Code**: `sim.js`, `blocks/sim/` directory