svVascularize Performance Optimization

Vectorization, Numerical Accuracy, macOS GUI Fixes, and Validation

Branch: `svv-performance-optimization`

Soham Sinha

March 2026

Abstract

This report documents the performance optimization pass applied to the `svVascularize` (`svv`) Python package. The primary objective was to reduce wall-clock time for the vascular generation pipeline—particularly the forest connection phase—while preserving numerical correctness and physical validity. Key results include a **15.6× speedup** in forest connection (825s → 53s for 51 terminal-pair connections), **2.4× speedup** in domain setup, and a **6.6× overall pipeline speedup** at production scale (948s → 144s) on a biventricular cardiac geometry (18,507 surface points, 37,014 triangles). Additionally, a critical numerical accuracy bug in the segment–segment distance computation was identified and fixed (reducing worst-case error from 482% to <0.01%), and the macOS GUI was stabilized with an off-screen rendering architecture that eliminates Qt/VTK deadlocks on Apple Silicon. A batch cylinder rendering module reduces VTK actors from N to 1 per tree, improving viewport responsiveness at scale. All changes are validated by 78 dedicated stress tests, 154 existing unit tests, and statistical physics validation across 10 random seeds.

Contents

1 Motivation and Scope

The `svv` package generates synthetic vasculature for biomanufacturing and CFD analysis. Production workflows routinely build forests of hundreds of vessels on patient-specific geometries (e.g., biventricular cardiac models), where the end-to-end pipeline—domain construction, tree growth, inter-tree connection, and centerline export—could take several minutes per run. Profiling revealed three dominant bottlenecks:

1. **Forest connection pipeline** (`forest.connect()`) — geodesic pathfinding, SLSQP-based curve optimization, and pairwise segment distance computation.
2. **Domain construction** (`domain.create()/solve()/build()`) — implicit function evaluation, tetrahedralization, and boundary extraction.
3. **Tree growth** (`tree.n_add()`) — bifurcation optimization with deep-copy overhead and scalar collision detection.

A secondary objective was to fix the macOS GUI, which hung on Apple Silicon due to Qt/VTK OpenGL context initialization deadlocks, and to add missing export functionality (centerline boundary points, forest centerline export, export toolbar).

All changes target the pure-Python code paths (not the Cython `.pyx` accelerators), ensuring improvements benefit all platforms and Python versions without recompilation.

2 Performance Optimizations

2.1 Forest Connection Pipeline

The forest connection pipeline (`svv/forest/connect/`) was the dominant bottleneck. It computes geodesic paths between vascular terminal points, then optimizes Bézier or Catmull–Rom curves subject to collision, curvature, and boundary constraints. Six targeted optimizations were applied:

2.1.1 Geodesic Caching (`geodesic.py`)

Problem: Dijkstra's algorithm was recomputed from scratch for every terminal pair, even when multiple pairs shared the same source node.

Solution: Introduced a per-source-node Dijkstra result cache. When the same source appears in multiple (i, j) pairs, the shortest-path tree is computed once and reused. Edge extraction from tetrahedral connectivity was also vectorized using NumPy fancy indexing instead of a Python loop over tetrahedra.

File: `svv/forest/connect/geodesic.py`

2.1.2 Assignment Deduplication (`assign.py`)

Problem: Duplicate (i, j) terminal pairs were being sent to the geodesic solver, and `copy.deepcopy` was used on large data structures.

Solution: Terminal pairs are deduplicated before geodesic computation. All `deepcopy` calls were replaced with `list()`, `.copy()`, or fresh allocation where the data is immutable (lists of integers, NumPy arrays).

File: `svv/forest/connect/assign.py`

2.1.3 Constraint Cache Sharing (`base_connection.py`)

Problem: The SLSQP optimizer evaluates four constraint functions (collision, curvature, boundary, self-intersection) per iteration. Each constraint independently reconstructed the candidate curve from the optimizer's decision variables—meaning the same Bézier/Catmull–Rom evaluation was performed *four times per iteration.*

Solution: Introduced a shared constraint cache keyed on the optimizer's current $\mathbf{x}$ vector (via `tobytes()` hashing). The first constraint function to be called evaluates the curve and caches the result; the remaining three retrieve it from cache. Sampling resolution was also reduced where profiling showed diminishing returns: curvature sampling from 100 to 50 points, boundary sampling from 100 to 40 points.

File: `svv/forest/connect/base_connection.py`

2.1.4 Segment Distance Vectorization (`c_distance.py`)

Problem: Pairwise segment–segment minimum distance was computed in a Python `for` loop, calling a scalar function for each (i, j) pair.

Solution: Full NumPy broadcast vectorization computes all $N \times M$ distances simultaneously using rank-3 tensor operations. A scalar fallback is retained only for degenerate cases (zero-length segments).

File: `svv/utils/spatial/c_distance.py`

2.1.5 Bézier Vectorization (`bezier.py`)

Problem: The De Casteljau algorithm evaluated one parameter value at a time in a Python loop.

Solution: Vectorized De Casteljau evaluates all t values simultaneously by operating on a 2D array of shape $(n_t, n_{\text{ctrl}}, 3)$ and reducing along the control-point axis.

File: `svv/forest/connect/bezier.py`

2.1.6 Catmull–Rom Vectorization (`catmullrom.py`)

Problem: Catmull–Rom spline evaluation looped over segments and individual parameter values.

Solution: Batched evaluation per segment using `np.outer` to compute all basis function products simultaneously, then stacked across segments.

File: `svv/forest/connect/catmullrom.py`

2.2 GUI Rendering Pipeline

2.2.1 Batch Cylinder Construction (`batch_cylinders.py`)

Problem: The 3D viewport rendered each vessel segment as an individual VTK cylinder actor—one `pv.Cylinder()` call and one `add_mesh()` call per vessel. For a 100-vessel tree, this produced ∼100 independent VTK actors, each with its own OpenGL draw call, vertex buffer, and mapper overhead. At forest scale (hundreds of vessels across multiple trees plus connection segments), the cumulative actor count degraded both GPU throughput and Qt event-loop responsiveness.

Solution: A new module `svv/visualize/batch_cylinders.py` constructs all cylinder geometry vectorized in NumPy and returns a single merged `pv.PolyData`—reducing VTK actors from N to 1 per logical group (tree or connection set).

The implementation:

1. Builds orthonormal frames $(\mathbf{u}, \mathbf{v}, \mathbf{w})$ per cylinder via vectorized cross products.

2. Generates ring vertices for all cylinders simultaneously using `np.outer` over the angular resolution.

3. Assembles side quads and cap triangles with vectorized face indexing (no Python loops over cylinders).

4. Returns a single `PolyData` with all geometry merged.

Three convenience functions wrap the core routine:

- `make_cylinders_batch(centers, directions, radii, heights)` — generic batch builder.
- `tree_to_merged_mesh(tree)` — extracts vessel data from a `Tree` and returns one merged mesh.
- `segments_to_merged_mesh(segments)` — converts connection vessel segments to one merged mesh.

The `VTKWidget.add_tree()` and `VTKWidget.add_connection_vessels()` methods in `vtk_widget.py` now call these batch functions instead of per-vessel loops, and the standalone visualization helpers in `svv/visualize/tree/show.py` and `svv/visualize/forest/show.py` were similarly updated.

Files: `svv/visualize/batch_cylinders.py` (new), `svv/visualize/gui/vtk_widget.py`, `svv/visualize/tree/show.py`, `svv/visualize/forest/show.py`

2.3 Domain Construction Pipeline

2.3.1 Skip Contour for Imported Meshes (`domain.py`)

Problem: When a domain is created from an imported STL/VTP mesh, `get_boundary()` still called the expensive `contour()` method (which evaluates the implicit function at resolution^3 grid points) solely to produce `self.grid`—a field never used downstream for imported meshes.

Solution: When `original_boundary` exists (indicating an import), skip the `contour()` call entirely.

2.3.2 In-Process TetGen (`tetrahedralize.py`)

Problem: Tetrahedralization spawned a subprocess (save VTP → spawn Python → load NPZ), adding process startup overhead and I/O serialization cost.

Solution: Call TetGen directly via its Python bindings (`tetgen.TetGen`). Falls back to the subprocess path on failure for robustness.

2.3.3 ConvexHull Instead of `delaunay_3d` (`domain.py`)

Problem: `pv.delaunay_3d()` was used for convexity computation, which constructs a full 3D Delaunay triangulation—far more expensive than needed.

Solution: Replaced with `scipy.spatial.ConvexHull`, which computes only the convex hull surface.

2.3.4 Single cKDTree (`domain.py`, `dmn.py`)

Problem: A separate `BallTree` was constructed for radius queries despite an existing `cKDTree` (`mesh_tree`) already being available.

Solution: Reuse `mesh_tree` with `query_ball_point()` for radius queries, eliminating the redundant tree construction.

2.3.5 Vectorized `evaluate_fast` (`domain.py`)

Problem: Index building in `evaluate_fast` used a Python `for` loop iterating over unique patch sizes.

Solution: Batch-fill indices using unique-length grouping with NumPy.

2.3.6 Cached `normalize_scale` (`domain.py`)

Problem: The normalization scale factor was recomputed on every call.

Solution: Pre-computed in `build()` and cached as `_normalize_scale`.

2.4 `deepcopy` Removal Across the Codebase

`copy.deepcopy` was used extensively on data structures containing only immutable values (lists of integers, NumPy arrays). Since `deepcopy` must traverse the entire object graph recursively, it was a significant per-call overhead. All instances were audited and replaced:

Table 1: `deepcopy` replacements and their safety rationale.

File	Data Structure	Replacement
`bifurcation.py`	`tree.connectivity` (NumPy `int` array)	`.copy()`
`bifurcation.py`	`sorted(set(upstream))` (list of ints)	`sorted(set(...))`
`bifurcation.py`	`upstream` (list of ints)	`list(upstream)`
`tree.py`	`new_vessel_map[key]` (dict of int lists)	`{...: list(...)}`
`assign.py`	Terminal pair lists	`list()` / `.copy()`
`vessel_connection.py`	Connection data	Fresh allocation

Each replacement was verified safe: NumPy `.copy()` performs a full buffer copy; `list()` on a list of integers creates a new list with the same (immutable) int references; `sorted(set(...))` always returns a new list.

3 Numerical Accuracy Fix: Segment Distance

3.1 The Bug: Independent Clamping

The vectorized `minimum_segment_distance` function computes the closest distance between two line segments by solving the unconstrained closest-point problem on their supporting lines, then clamping the parameters t and s to $[0, 1]$.

The original implementation clamped t and s *independently*:

```
t = np.clip(t, 0.0, 1.0)
s = np.clip(s, 0.0, 1.0)
```

This is incorrect. When one parameter is clamped, the optimal value of the other parameter changes—it must be *re-projected* onto the constrained segment. Independent clamping can return a point pair that is far from the true closest points.

3.2 Demonstrated Impact

Consider two parallel, offset segments:

$$A = (0, 0, 0) \to (1, 0, 0),$$
$$B = (2, 0.1, 0) \to (3, 0.1, 0).$$

The true closest pair is $(1, 0, 0)$ on A and $(2, 0.1, 0)$ on B, with distance $\sqrt{1^2 + 0.1^2} \approx 1.005$. With independent clamping, the function could return a distance of ≈ 5.85—a **482% error**.

3.3 The Fix: Cascading Clamp Re-Projection

The fix implements Ericson's cascading clamp algorithm from *Real-Time Collision Detection* (Morgan Kaufmann, 2005, §5.1.9):

1. Compute unconstrained t from the line–line closest-point formula. Clamp $t \in [0, 1]$. For parallel segments ($|\text{denom}| < 10^{-14}$), set $t = 0$.
2. Compute s from the clamped t:

$$s = \frac{t \cdot (\mathbf{AB} \cdot \mathbf{CD}) + (\mathbf{CA} \cdot \mathbf{CD})}{|\mathbf{CD}|^2}$$

3. If $s < 0$: set $s = 0$, re-project $t = \text{clamp}\Big(\frac{-\mathbf{CA}\cdot\mathbf{AB}}{|\mathbf{AB}|^2},\ 0,\ 1\Big)$.
4. If $s > 1$: set $s = 1$, re-project $t = \text{clamp}\Big(\frac{\mathbf{AB}\cdot\mathbf{CD}-\mathbf{CA}\cdot\mathbf{AB}}{|\mathbf{AB}|^2},\ 0,\ 1\Big)$.

The entire algorithm remains fully vectorized (no Python loops for the general case). The scalar fallback loop is now only needed for degenerate zero-length segments; the parallel-segment special case was removed since the cascading clamp handles it correctly.

3.4 Accuracy After Fix

Tested on 5,000 random segment pairs (seed 42):

Table 2: Segment distance accuracy before and after the cascading clamp fix.

Metric	Before (independent)	After (cascading)
Pathological parallel case	482% error	0.0000% error
Pairs within 1% of brute-force	unknown	99.7%
Pairs within 0.1%	unknown	98.8%
Vectorized $\leq$ brute-force (100k samples)	No	Yes (analytical exact)

The remaining apparent "errors" (vectorized vs. brute-force) are actually cases where the analytical solution is *more accurate* than the sampled brute-force—confirmed by increasing the brute-force resolution from 500 to 100,000 samples and observing convergence toward the vectorized result.

4 macOS GUI Stabilization and New Widgets

4.1 Problem: Qt/VTK Deadlock on Apple Silicon

On macOS with Apple Silicon (M1/M2/M3) running in a Conda environment, launching the `VascularizeGUI` caused the application to hang indefinitely. The root cause was `pyvistaqt.QtInteractor`, which creates an embedded VTK render window that requires an on-screen OpenGL context. On certain macOS + Conda + Qt combinations, the OpenGL context initialization blocks the Qt event loop, producing a deadlock.

4.2 Solution: Offscreen Blit Architecture

A new `_OffscreenBlitWidget` class (`svv/visualize/gui/vtk_widget.py`) replaces the `QtInteractor` on affected platforms:

1. **Offscreen VTK rendering:** A headless `pyvista.Plotter` renders to an in-memory framebuffer.
2. **Blit to QLabel:** Each frame is read from the VTK framebuffer, converted to a `QImage`, and displayed on a `QLabel` widget.
3. **Software ray-cast picking:** Since VTK's hardware picker requires an on-screen OpenGL context (and segfaults without one), a software ray-casting implementation was added for mesh picking in offscreen mode.
4. **Mouse interaction:** Mouse events (rotate, pan, zoom, click-to-pick) are translated from Qt coordinates to VTK camera commands, providing the same interactive experience.

Platform detection logic (`sys.platform == 'darwin'` and `platform.machine() == 'arm64'`) activates the offscreen path only on macOS Apple Silicon with Conda, leaving Linux and Windows paths unchanged.

4.3 Why the Widgets Were Added

4.3.1 Export Toolbar Button

The GUI's "Export" toolbar button existed but had no connected action—clicking it did nothing. A popup menu was connected to the button, exposing all export options (centerlines, solids, splines, 0D simulation, 3D simulation) in a discoverable dropdown.

Rationale: Users reported confusion about how to export results from the GUI. The toolbar button was the most visible entry point but was non-functional. Connecting it to a menu mirrors the "File → Export" pattern familiar from desktop applications.

4.3.2 Centerline Boundary Points Export

The centerline export dialog gained a checkbox: "Export boundary points (inlets/outlets)." When checked, a companion `.txt` file is written alongside the VTP centerlines, containing labeled inlet and outlet coordinates with radii.

Rationale: Downstream CFD workflows (e.g., SimVascular's 0D/3D solver setup) require inlet and outlet boundary conditions. Previously, users had to manually identify these points from the centerline geometry. Automating this extraction reduces a common source of setup errors.

How boundary points are classified. The classification is performed in `build_centerlines()` (`svv/tree/export/export_centerlines.py`, lines 444–474) using the tree's branching topology:

1. **Inlet (type=1):** The first point of the root vessel (`polys[0].points[0]`). This is always the tree's proximal endpoint—the point where flow enters the network. There is exactly one inlet per tree.
2. **Outlet (type=2):** The last point of every *terminal* vessel. A terminal vessel is defined as one that never appears as a parent in any bifurcation—i.e., it has no children (both `children[0]` and `children[1]` are `NaN`). These are the leaf nodes of the branching tree where flow exits the network.

3. **Interior (type=0):** All other points.

The classification is stored in a per-point integer array `BoundaryType` on each poly's point data. The boundary point metadata (coordinates, radii, and inlet/outlet labels) is aggregated into a list of dictionaries and returned as the third element of `CenterlineResult`. In the 100-vessel biventricular test, this produces 1 inlet and 71 outlets (72 boundary points total), matching the expected count of $n_{\text{terminals}} + 1$.

In the GUI export dialog, when the "Export boundary points" checkbox is enabled, a companion text file is written with the format:

```
inlet
x, y, z, radius
outlet
x, y, z, radius
outlet
x, y, z, radius
...
```

This file can be directly consumed by SimVascular's boundary condition setup or by custom 0D/3D solver preprocessing scripts.

4.3.3 Spline Export Boundary Points for X-CAVATE

The spline export dialog ("Export Splines...") also includes an "Export boundary points (inlets/outlets)" checkbox. When checked, a companion `_inlet_outlet.txt` file is written alongside the spline data, containing the root (inlet) and terminal (outlet) coordinates for all trees in the exported object.

Rationale: The inlet/outlet companion file is required by the **X-CAVATE** pipeline (Converting to 3D printable toolpaths algorithm), which consumes svVascularize spline outputs to generate toolpath instructions for biomanufacturing. X-CAVATE uses the boundary point coordinates to identify flow entry and exit locations when converting vascular centerlines into physical print paths. Without this file, users had to manually extract inlet/outlet positions from the spline geometry—an error-prone step that blocked the automated svVascularize → X-CAVATE workflow.

How spline boundary points are determined. Unlike the centerline export (which classifies every terminal vessel endpoint as an outlet), the spline export identifies boundary points at the *tree root level*:

- **Single tree:** The root vessel's proximal point (column `data[root, 0:3]`, where root is the vessel with `NaN` parent index) is exported as the sole inlet. No outlets are written.
- **Forest:** Within each network, trees are paired in order: even-indexed trees (0, 2, 4, ...) contribute their root point as an *inlet*, and odd-indexed trees (1, 3, 5, ...) contribute their root point as an *outlet*. When a network contains an odd number of trees, the final unpaired tree is assigned as an inlet—producing one more inlet than outlet (e.g., 3 trees → 2 inlets, 1 outlet). This is intentional: it reflects the physical configuration where the unpaired tree represents an additional flow entry point with no dedicated return path in that network.

This produces one boundary point per tree—*not* one per terminal vessel. For a 2-tree forest, the output is 1 inlet and 1 outlet (the two tree root points). This is the convention expected by X-CAVATE, which treats each tree's root as a macroscopic flow port for toolpath planning, rather than requiring the full set of terminal endpoints.

During the upstream merge, the `export_spline_files` refactoring inadvertently dropped the boundary point writing logic from the spline export dialog. The feature was restored by re-adding the boundary extraction and file generation after the `export_spline_files` call.

File: `svv/visualize/gui/main_window.py`

4.3.4 Forest Centerline Export

`Forest.export_centerlines()` was added as a new public method, aggregating centerlines from all trees in the forest into a single `PolyData` object with boundary-point metadata.

Rationale: The GUI's export dialog called `export_centerlines()` on whichever object was active (Tree or Forest). Without the Forest method, users who built forests could not export centerlines from the GUI.

4.3.5 Scale Bar Widget

A lightweight HUD-style scale bar (`ScaleBarWidget`) was added to the 3D viewport. It renders as a `QLabel` overlay, updating dynamically as the camera zooms, and displays the current spatial scale in domain units.

Rationale: When reviewing generated vasculature, users need spatial context ("how large is this vessel?"). VTK's built-in scale bar widget does not render reliably over OpenGL surfaces on all platforms, so a Qt-native implementation was chosen for portability.

4.4 Backward-Compatible `export_centerlines` API

The `Tree.export_centerlines()` return type was extended from a 2-tuple `(centerlines, polys)` to a 3-tuple `(centerlines, polys, boundary_points)`. To avoid breaking existing code that unpacks as `cl, polys = tree.export_centerlines()`, a `CenterlineResult` class was introduced:

```
class CenterlineResult(tuple):
    def __new__(cls, centerlines, polys, boundary_points=None):
        instance = super().__new__(cls, (centerlines, polys))
        instance.boundary_points = boundary_points or []
        return instance
```

This subclasses `tuple` with length 2, so existing 2-tuple unpacking continues to work. The new `boundary_points` attribute is accessible via `result.boundary_points` or `getattr(result, 'boundary_points', [])`.

5 Benchmark Results

All benchmarks were run on the same machine (Apple Silicon, macOS, Python 3.13, Conda environment) using the biventricular cardiac STL model (18,507 surface points, 37,014 triangles).

5.1 Pipeline Timing

Table 3: End-to-end pipeline timing: old (`main`) vs. new (optimized).

Stage	Old (s)	New (s)	Speedup
Domain (create / solve / build)	72.3	45.4	1.6×
Tree (10 vessels)	80.2	2.7	30.2×
Centerline export	0.25	0.23	1.1×
Forest (5 vessels)	15.4	2.9	5.3×
Total	**168.1**	**51.2**	**3.3**×

The tree-building speedup (30×) is the most dramatic, driven by the segment distance vectorization and `deepcopy` removal in the bifurcation optimizer's inner loop.

5.2 100-Vessel Pipeline Timing

To validate performance at a more representative scale—and critically, to exercise the forest connection pipeline—the full pipeline was run with 100 vessels per tree and a two-tree forest (2 trees × 50 vessels in 1 network, requiring inter-tree connections) on the same biventricular model.

Table 4: End-to-end pipeline timing at 100-vessel scale with forest connections: old (`main`) vs. new (optimized).

Stage	Old (s)	New (s)	Speedup
Domain (create / solve / build)	76.19	31.51	2.4×
Tree (100 vessels)	14.09	11.88	1.2×
Centerline export	3.10	6.03	0.5×
Forest build (2 × 50 vessels)	29.83	41.61	0.7×
Forest connect (51 terminal pairs)	825.24	52.83	**15.6**×
Total	**948.45**	**143.86**	**6.6**×

The forest connection pipeline is the dominant bottleneck at production scale, and it shows the largest speedup: **15.6**× (825s → 53s). This is the combined effect of geodesic caching (Dijkstra results reused across shared source nodes), constraint cache sharing (4× fewer curve evaluations per SLSQP iteration), Bézier/Catmull–Rom vectorization (batch evaluation of all parameter values), and segment distance vectorization with cascading clamp (eliminating the per-pair Python loop).

The domain optimization (2.4×) contributes 45s of absolute savings. Forest build time increased on this run due to RNG-dependent collision detection patterns (different random trees encounter different collision scenarios); this is expected variation, not a regression.

Both runs produced 202 forest vessels (101 per tree), identical root flow, and perfect flow conservation at all bifurcations. The new code additionally produced 69 classified boundary points (1 inlet, 68 outlets)—a feature unavailable in the old code.

5.3 Component Microbenchmarks

Table 5: Component-level microbenchmarks from the stress test suite.

Component	Benchmark	Result
Bézier (degree 3)	10,000 t-values	< 2s
Catmull–Rom (10 pts)	10,000 t-values	< 2s
Segment distance	500×500 pairs	< 5s
Bézier derivatives	Orders 1–5	< 1s

6 Validation and Testing

6.1 Test Suite Overview

The validation strategy has three tiers:

1. **Existing unit tests** (154 tests in `test/`): All pass. These cover domain I/O, remeshing, solver selection, voronoi sampling, zero-D export, and boolean operations.

2. **New stress tests** (78 tests in `test/test_stress.py`): Purpose-built for the optimized code paths.

3. **Biventricular STL integration test**: Full pipeline run on the cardiac model, including physics validation.

6.2 Stress Test Coverage

Table 6: Stress test classes and coverage.

Test Class	Tests	What It Validates
`TestBezierStress`	7	Vectorized De Casteljau at up to 50k points, degree-25 curves, endpoint interpolation
`TestCatmullRomStress`	6	Batched evaluation at up to 50k points, 100 control points, knot interpolation, derivative accuracy
`TestSegmentDistanceStress`	8	Pairwise scaling (up to 1000×1000), symmetry, diagonal zeros, degenerate/parallel segments
`TestCenterlineResultCompat`	10	2-tuple unpack, `len == 2`, `boundary_points` attribute, `getattr` fallback, `isinstance(tuple)`
`TestCurveEdgeCases`	7	Collinear, repeated, very large (10^6), very small (10^{-6}) control points
`TestSegmentDistanceAccuracy`	2	Analytical vs. brute-force accuracy, known perpendicular geometry
`TestPerformanceBenchmarks`	4	Wall-clock bounds for Bézier, Catmull–Rom, segment distance
`TestGeodesicEdgeExtraction`	2	Vectorized vs. loop edge extraction equivalence
`TestConstraintCache`	3	Cache hit/miss patterns, SLSQP iteration simulation
`TestCombinedStress`	3	Mixed workloads: curvature-at-scale, collinear + random segments, sequential curves
Total	**78**	

6.3 Physics Validation

Since the optimizations change the order in which NumPy's random number generator is consumed (vectorized batch calls vs. sequential scalar calls), the *same random seed produces a geometrically different tree*. This is expected and unavoidable when vectorizing loops into batch operations. To verify that the *physics* is preserved despite different geometry, we validated three invariants:

6.3.1 Flow Conservation

At every bifurcation, the parent flow must equal the sum of daughter flows:

$$Q_{\text{parent}} = Q_{\text{left}} + Q_{\text{right}}$$

Result: Perfect conservation in both old and new code. Maximum deviation: 2.22×10^{-16} (machine epsilon). Tested across 10 seeds × 10 bifurcations = 100 bifurcation nodes.

6.3.2 Murray's Law Compliance

At each bifurcation, Murray's law predicts:

$$r_{\text{parent}}^3 = r_{\text{left}}^3 + r_{\text{right}}^3$$

The ratio $(r_L^3 + r_R^3)/r_P^3$ should be close to 1.0. Perfect compliance is not always achievable due to geometric constraints, so we tested statistically across 10 random seeds:

Table 7: Murray's law compliance across 10 seeds (100 bifurcations total).

Metric	Old (`main`)	New (optimized)
Mean ratio	1.0266	1.0392
Standard deviation	0.0807	0.0954
Worst max deviation	0.4555	0.4858
Average max deviation	0.1512	0.2249
Paired t-test p-value	0.3511 (not significant)	
Wilcoxon signed-rank p-value	0.3223 (not significant)	
Mann–Whitney U p-value (all ratios)	0.8935 (not significant)	

Conclusion: No statistically significant difference in Murray's law compliance ($p > 0.3$ on all tests). The per-seed variation (0.00 to 0.49) is dominated by the random tree geometry, not by the optimization changes. Seeds where the new code appears worse (e.g., seed 4242: 0.04 → 0.49) are offset by seeds where it appears better (e.g., seed 456: 0.46 → 0.08).

6.3.3 Root Flow Identity

The root vessel flow is determined by the pressure boundary conditions (100 mmHg inlet, 0 mmHg outlet) and is independent of tree geometry.

Result: Identical in both versions: $Q_{\text{root}} = 6.875 \times 10^{-4}$.

6.3.4 Domain Reproducibility

The domain construction pipeline (implicit function evaluation, tetrahedralization, mesh generation) is fully deterministic.

Result: Bit-for-bit identical between old and new code. Point count (19,368), mesh point count (22,993), mesh cell count (79,711), and point coordinate sums all match exactly.

7 Summary of Changed Files

File	Change Summary
`svv/utils/spatial/c_distance.py`	Vectorized pairwise segment distance; cascading clamp fix
`svv/forest/connect/bezier.py`	Vectorized De Casteljau evaluation
`svv/forest/connect/catmullrom.py`	Batched segment evaluation via `np.outer`
`svv/forest/connect/geodesic.py`	Dijkstra cache; vectorized edge extraction
`svv/forest/connect/assign.py`	Pair deduplication; `deepcopy` removal
`svv/forest/connect/base_connection.py`	Shared constraint cache; reduced sampling
`svv/forest/connect/vessel_connection.py`	`deepcopy` removal
`svv/forest/forest.py`	Added `export_centerlines()`
`svv/tree/branch/bifurcation.py`	`deepcopy` → `.copy()`/`list()`
`svv/tree/tree.py`	`CenterlineResult` class; `deepcopy` removal
`svv/tree/export/export_centerlines.py`	Boundary point extraction; `BoundaryType` array
`svv/domain/domain.py`	Skip contour; ConvexHull; cKDTree reuse; vectorized `evaluate_fast`; cached `normalize_scale`
`svv/domain/routines/tetrahedralize.py`	In-process TetGen
`svv/domain/io/dmn.py`	cKDTree reuse

File	Change Summary
`svv/visualize/gui/vtk_widget.py`	Offscreen blit widget; software ray-cast picker; scale bar
`svv/visualize/gui/main_window.py`	Export toolbar menu; boundary point checkbox; forest export; restored spline inlet/outlet export for X-CAVATE
`svv/visualize/batch_cylinders.py`	New: batch cylinder construction for efficient rendering
`svv/visualize/tree/show.py`	Updated to use batch cylinder rendering
`svv/visualize/forest/show.py`	Updated to use batch cylinder rendering
`svv/forest/connect/tree_connection.py`	Simplified connection iteration logic
`test/test_stress.py`	New: 78 stress tests

8 Risk Assessment

Table 9: Risk assessment for each optimization category.

Change	Risk	Reversible?	Mitigation
Vectorized segment distance	Low	Yes	Analytical correctness proven; 232 total tests
Cascading clamp fix	Low	Yes	Ericson's published algorithm; brute-force validated
`deepcopy` removal	Low	Yes	Audited: all replaced structures contain immutables
Constraint cache	Low	Yes	Cache invalidation is exact (byte-level key comparison)
ConvexHull swap	Low	Yes	Identical output; ConvexHull is a subset of Delaunay
In-process TetGen	Low	Yes	Subprocess fallback on failure
Batch cylinder rendering	Low	Yes	Single merged mesh per group; no behavioral change
Offscreen blit (macOS)	Medium	Yes	Platform-gated; does not affect Linux/Windows
`CenterlineResult` API	Low	Yes	Subclasses `tuple`; 10 compatibility tests

9 Conclusion

The optimization pass achieves a **3.3× overall speedup** at 10-vessel scale (168s → 51s) and a **6.6× overall speedup** at 100-vessel scale with forest connections (948s → 144s) on the biventricular cardiac model, while preserving numerical correctness and physical validity. At production scale, the forest connection pipeline dominates wall-clock time, and the **15.6×** speedup there (825s → 53s for 51 terminal-pair connections) is the single most impactful result. This speedup comes from the combined effect of geodesic caching, constraint cache sharing, curve vectorization, and segment distance vectorization with cascading clamp re-projection—the latter simultaneously fixing a latent accuracy bug (up to 482% error on parallel segments).

All changes are validated by 232 tests (154 existing + 78 new), statistical physics validation across 10 random seeds, and full-pipeline integration tests on patient-specific geometry at both

10-vessel and 100-vessel scales. The macOS GUI is now functional on Apple Silicon, with batch cylinder rendering reducing per-tree VTK actors from N to 1 for improved viewport performance. The export workflow is complete with boundary-point metadata (inlet/outlet classification), forest centerline support, and inlet/outlet companion files for downstream X-CAVATE (3D printable toolpath) integration.