SignalShow Modernization Plan

A Research Proposal for Modernizing an Interactive Signal Processing Education Platform

---

About This Document

This is a research proposal for **SignalShow**, a modernization of an existing Java educational signal processing application. The proposal explores bringing the application to modern web and desktop platforms.

Vision: "SignalShow is to DSP what Desmos is to algebra" - making complex signal processing concepts accessible through interactive visualization and hands-on exploration.

Status: Early prototype phase. All designs and timelines are subject to change.

Document Structure

SignalShow Modernization Plan - Table of Contents

SignalShow

Overview

SignalShow is an educational signal processing application, originally built in Java Swing. This proposal explores modernizing it for web and desktop platforms.

Vision: "SignalShow is to DSP what Desmos is to algebra"

Status: Early prototype phase

Versions

1. **Web Version** - Browser-based prototype in React/Plotly.js
2. **Desktop Version** - Proposed Tauri app with optional Julia backend
3. **Java Version** - Original Swing GUI with 80+ functions (legacy)

Web Prototype

Repository: `/nuthatch-desktop`

Features

- 4 signal types: Sine, Cosine, Chirp, Gaussian
- Interactive waveform visualization (Plotly.js)
- FFT analysis
- Window functions: Hanning, Hamming, Blackman
- 5 educational demos
- JSON export
- Responsive 3-column layout

Components

- **SignalDisplay** - Interactive waveform plots
- **FunctionGenerator** - Signal parameter controls
- **OperationsPanel** - FFT and windowing operations
- **DemoGallery** - Educational examples

Development

```
cd /Users/julietfiss/src/nuthatch-desktop
npm install
npm run dev
# Open http://localhost:5173
```

Planned (Phase 2)

- Rust/WebAssembly backend for performance
- Additional signal types and operations

- Advanced filtering
- 3D visualizations

Java Desktop Version

The original implementation with 80+ functions and 40+ operations, now using Maven.

Key Features

- Analytic function generators (Gaussian, Chirp, Delta, Bessel, windows, noise models, etc.)
- 1-D / 2-D interactive plotting (zoom, pan, overlays)
- Frequency-domain and convolution operations
- Educational demos (sampling, filtering, holography, more)
- Modular operation architecture

Building with Maven

```
# Compile
mvn clean compile

# Run directly
mvn exec:java

# Or use convenience scripts
./compile.sh    # Compile the project
./run-maven.sh  # Run with Maven
./package.sh    # Create executable JAR
./run.sh        # Run from JAR
```

Project Structure

```
src/main/java/           # Java source files
  SignalShow.java        # Main entry point
  signals/               # Application packages
src/main/resources/      # Images, icons, documentation
pom.xml                  # Maven configuration
legacy-build/            # Original build artifacts
  jai_core.jar           # JAI core classes (installed to local Maven)
  jai_codec.jar          # JAI codec classes (installed to local Maven)
```

Prerequisites

- Java Development Kit (JDK) 11+
- Bundled JAI jars: `jai_core.jar`, `jai_codec.jar`

Quick Start

```
./run-signalshow.sh
```

This script verifies JAI jars, sets the classpath, and launches the main class.

Building from Source

```
find SignalShow -name "*.java" > /tmp/sigshow-srcs.txt
javac -d . @/tmp/sigshow-srcs.txt
java -cp "SignalShow:SignalShow/jai_core.jar:SignalShow/jai_codec.jar" SignalShow
```

Or one-liner:

```
find SignalShow -name "*.java" -print0 | xargs -0 javac -d . && \
java -cp "SignalShow:SignalShow/jai_core.jar:SignalShow/jai_codec.jar" SignalShow
```

Troubleshooting

Symptom	Likely Cause	Resolution
`NoClassDefFoundError: javax/media/jai/PlanarImage`	JAI jars not on classpath	Ensure classpath includes JAI jars. Remove any extracted `SignalShow/javax` directory.
`SecurityException: sealing violation`	Duplicate package definition	Delete stray `javax/` directory.
`UnsatisfiedLinkError` for JAI native libs	Platform mismatch (x86 vs arm64)	Run under Rosetta with x86 JDK, or obtain arm64 native builds.

Diagnostics

List native libraries:

```
find . -type f \( -name "*.so" -o -name "*.dylib" -o -name "*.jnilib" -o -name "*.dll" \) -
print
```

Check jar contents:

```
jar tf SignalShow/jai_core.jar | grep javax/media/jai/PlanarImage.class
```

Remove accidentally extracted classes:

```
rm -rf SignalShow/javax
```

Extending the Application

New analytic functions or operations can be added by creating classes under `signals/...` subpackages and wiring them into existing registries or menus.

Contributing

1. Fork the repository
2. Create a feature branch (`git checkout -b feature/your-feature`)
3. Commit changes with clear messages
4. Open a pull request

License

Refer to LICENSE file or header comments in source. JAI libraries are bundled in binary form; their original licensing terms apply.

SignalShow Web Modernization - Research Overview

Vision: "SignalShow is to DSP what Desmos is to algebra"

Goal: Transform SignalShow from a Java Swing application into a modern web platform for university teaching, research figure generation, and educational content creation.

Status: Web UI prototype complete; Julia backend and desktop version proposed

Current Status

Phase 1 (Web UI): Four React components implemented and tested

- SignalDisplay - Interactive waveform plots (Plotly.js)
- FunctionGenerator - Signal parameter controls (4 signal types)
- OperationsPanel - FFT and windowing operations
- DemoGallery - 5 educational examples

Backend: JavaScript implementation functional (zero installation)

Next Priorities:

1. **v1.0**: WAV/CSV I/O, accessibility (WCAG 2.2 AA), guided mode, timeline recording, concept packs
2. **v1.5**: WASM performance, LMS integration, Docker appliance
3. **v2.0**: Tauri app, plugin API, Manim export, Julia runtime

See Chapter 17: Roadmap Revisions for timeline and Chapter 16: Strategic Recommendations for positioning strategy.

Product Positioning

Market Position

SignalShow occupies a unique niche:

- Only web-based DSP education tool with publication-quality exports
- Only platform coupling 1D signal analysis with 2D optics simulations
- Only tool offering reproducible figure pipelines with provenance tracking
- Only modern stack for DSP labs (J-DSP is legacy Java)

Competitive Differentiation

vs. MATLAB: Free, web-based, beautiful UI, instant sharing
vs. Desmos: Deep DSP/optics vs. general math; advanced operations
vs. GeoGebra: Focused expertise in signals vs. broad math
vs. Observable: GUI-first vs. code-first; Julia backend option
vs. J-DSP: Modern React/Julia stack vs. legacy Java; story-driven demos
vs. PhET: Specialist DSP/optics vs. general physics; publication workflow

Full competitive analysis in Chapter 9: Competitive Analysis.

Target Partnerships

- Textbook publishers (interactive companion demos)
- OpenStax (open-source DSP textbook integration)
- IEEE Education Society (workshop co-sponsorship)
- ABET (align concept packs with accreditation outcomes)

User Personas

Four primary audiences with distinct needs:

1. Dr. Elena Martinez - Instructor (40%)
Goal: Make abstract DSP concepts tangible
Needs: Pre-built concept packs, presentation mode, LMS integration, progress tracking

2. Alex Chen - Student (45%)
Goal: Build intuition through exploration
Needs: Guided mode, undo/redo, WAV/CSV import, lab report export

3. Dr. Raj Patel - Researcher (10%)
Goal: Publication-quality figures with reproducibility
Needs: High-DPI exports, CLI batch generation, expert mode, Python/Julia API

4. Maya Rodriguez - Content Creator (5%)
Goal: Educational video production
Needs: Timeline recording, Manim export, custom branding, 4K rendering

Full personas in Chapter 3: User Personas.

Project Context

Current State (Java Version)

- Technology: Java Swing GUI (2005-2009 codebase)
- Files: ~395 Java files
- Features:
  - 80+ analytic functions (Gaussian, Chirp, Bessel, Delta, windows, noise)
  - 40+ operations (FFT, filtering, convolution, correlation, derivatives)
  - Interactive demos (sampling, filtering, holography, Lissajous curves)
  - 1D & 2D support (signals and images)
  - Java Advanced Imaging (JAI) integration

Proposed Future State

- Platform: Nuthatch Desktop modular app (web + desktop hybrid)
- Computation: Julia backend for mathematical operations
- Frontend: React 19+ with modern JavaScript
- Visualization: Plotly.js, D3.js, Three.js (3Blue1Brown-inspired graphics)
- File extensions: `.sig1d`, `.sig2d`, `.sigOp`, `.sigWorkspace`, `.sigDemo`
- Use cases: Live demos, guided labs, publication figures, video production, shareable workspaces

---

Technology Research

Julia-JavaScript Integration

Strategy: Julia backend server + REST/WebSocket API
Rationale: WebAssembly compilation not production-ready
Performance: Native-speed DSP operations
Real-time: WebSocket streaming for interactivity

Stack:

- Julia HTTP.jl + WebSockets.jl for server
- JSON3.jl for data serialization
- DSP.jl, FFTW.jl for signal processing
- Binary transfer (Float32Array) for large datasets
- HTTP.jl performance: ~200 req/sec

Scientific Visualization

Primary: Plotly.js (108k ⭐) for 2D scientific plots
Animations: Framer Motion (23k ⭐) + D3.js (108k ⭐)
3D: Three.js (103k ⭐) + react-three-fiber (29.7k ⭐)
Export: PNG/SVG/PDF via Plotly; video via Manim integration
Performance: 60fps+ with WebGL

See Chapter 13: Animation and 3D Strategy for details.

Signal Processing

Primary: Julia DSP.jl backend
JavaScript: Supplementary (client-side previews)
Performance: Julia is 5-10x faster than JavaScript for FFT
Libraries:

- Julia DSP.jl (primary engine)
- fft.js (client-side previews, 192k downloads/week)
- math.js (basic operations)
- stdlib-js (statistical functions)

Architecture

Pattern: Hybrid modular app in Nuthatch Desktop
State: Zustand (React state management)
Deployment: Tauri (desktop) + Vercel/Netlify (web)
File System: OPFS for browser, native FS for desktop

Data Flow:

- User interaction → React component → Zustand store
- Store → Julia server (HTTP/WebSocket) → DSP computation
- Result → Zustand → Plotly.js → Render

See Chapter 5: System Architecture and Chapter 6: Technology Stack for specifications.

Feature Implementation Strategy

Function Generation (80+ functions from Java)

Categories:

1. Basic waveforms (sine, cosine, chirp, Gaussian)
2. Window functions (Hanning, Hamming, Blackman, Kaiser)
3. Noise models (Gaussian, Poisson, Binomial, Shot)
4. Specialized (Bessel, Airy, Hermite polynomials, Laguerre)
5. Optical apertures (circular, rectangular, annular)
6. 2D functions (cylinders, Cassegrain mirrors)

Implementation:

- Core functions: Julia DSP.jl, SpecialFunctions.jl
- 2D apertures: Custom Julia implementations
- Optical diffraction: Julia FFTW.jl

Phase 1 subset (4 functions):

- Sine, Cosine, Chirp, Gaussian (all implemented)

Operations (40+ from Java)

Categories:

1. Frequency domain (FFT, IFFT, power spectrum)
2. Filtering (lowpass, highpass, bandpass, FIR, IIR)
3. Correlation (cross-correlation, autocorrelation)
4. Convolution (1D, 2D, circular, linear)
5. Time-frequency (STFT, wavelets, spectrograms)
6. Derivatives/integrals (numerical differentiation, integration)
7. Optics (Fresnel propagation, aperture functions)

Implementation:

- Julia DSP.jl (filtering, FFT, windowing)
- Julia FFTW.jl (Fourier transforms)
- Custom Julia code (optics, correlation)

Phase 1 subset (2 operations):

- FFT, windowing (all implemented)

See Chapter 4: Feature Overview for complete mapping.

Demos & Interactive Features

Educational Demos (5 initial):

- Sampling theorem
- Windowing effects on FFT
- Filter frequency responses

- Convolution visualization
- Chirp analysis

Interactive Elements:

- Real-time parameter adjustment
- Drag-and-drop operation chains
- Keyboard shortcuts for presentation
- Touch support for iPad

Modern DSP Features

Time-Frequency Analysis:

- Short-Time Fourier Transform (STFT)
- Spectrograms with dB scaling
- Wavelet transforms (Haar, Daubechies, Morlet, Mexican Hat)
- Hilbert transform (envelope, instantaneous phase/frequency)
- MFCCs (Mel-Frequency Cepstral Coefficients)

Medical Imaging:

- Radon transform (forward/inverse)
- Filtered back-projection for CT reconstruction
- Sinogram visualization

See Chapter 8: Feature Implementation Roadmap for details.

Hybrid Backend Strategy

Desktop (Tauri):

- Full Julia server backend
- Local computation, complete feature set
- Auto-start with 15-minute timeout
- Requirements: Julia (~~500MB) + packages (~~1-2GB)

Web (Browser):

- Rust + WebAssembly + JavaScript DSP
- No installation required
- ~95% feature parity with desktop

Backend Abstraction:

```
const backend = window.__TAURI__
  ? new JuliaServerBackend()  // Desktop: optimal performance
  : new WebBackend();          // Browser: zero install
```

Performance:

Backend	Environment	Speed	FFT (8192)	Bundle

Julia	Desktop	100%	~1ms	Server
JavaScript	Web	5-10%	~100ms	~23KB
Rust WASM	Web	60-95%	~2-5ms	~150KB

See `BACKEND_ABSTRACTION.md` .

UI/UX Design

Framework: shadcn/ui + Tailwind CSS + Radix UI

- Accessible (WCAG 2.1 compliant)
- Dark mode for classroom use
- Responsive with touch support

3Blue1Brown Aesthetic:

- Manim color schemes (blue/yellow, teal/pink)
- CMU Serif typography
- KaTeX for math rendering
- Smooth animations (Framer Motion spring physics)

Workflows:

1. Interactive exploration (student experimentation)
2. Figure generation (research publications)
3. Video production (educational content)

Deployment

Web Hosting:

- Static hosting (Vercel/Netlify/GitHub Pages)
- PWA with offline UI support
- Julia server runs locally or cloud

Desktop Distribution:

- Tauri installers: DMG (macOS), MSI (Windows), AppImage (Linux)
- App signing and auto-updates
- Bundle: ~600KB Tauri + ~400MB Julia + ~200MB Python/Manim

Development Roadmap

v1.0 (4-6 months) - Core Features:

- 6-8 prototype functions (Gaussian, sine, chirp, Gaussian, rect, delta)
- 8-10 essential operations (FFT, basic filters, convolution, correlation)
- Plotly.js 2D visualization
- JSON/PNG/SVG export

- Guided tutorials
- Web version with JavaScript backend

v1.5 (7-9 months) - Complete Library + 3D:

- All 80+ functions in organized batches
- All 40+ operations
- 3D visualization (2D FFT surfaces, holography)
- Demo library (5-10 key demos)
- WASM performance

v2.0 (10-12 months) - Desktop + Video:

- Tauri desktop app with bundled Julia
- Integrated Manim video export
- Python notebook integration
- Advanced 3D (custom shaders)

v2.5+ - Educational Platform:

- Exercise/assignment system
- Real-time collaboration
- LMS integration
- Community demo library

Implementation Status

Research Phase: Complete

Documents Created:

- ✅ TECH_STACK.md - Comprehensive technology decisions
- ✅ ARCHITECTURE.md - System design and API specs
- ✅ ANIMATION_AND_3D_STRATEGY.md - Graphics and animation details
- ✅ SIMILAR_PROJECTS_COMPARISON.md - Competitive analysis
- ✅ FEATURE_MAPPING.md - All 80+ functions and 40+ operations mapped
- ✅ FEATURE_ADDITIONS.md - Modern DSP enhancements
- ✅ BACKEND_ABSTRACTION.md - Multi-backend architecture
- ✅ PERSONAS.md - User stories and needs
- ✅ ROADMAP_REVISIONS.md - Timeline and priorities
- ✅ SIGNALSHOW_STRATEGIC_RECOMMENDATIONS.md - Positioning strategy

Next Steps:

1. Build proof-of-concept (Julia server + React + Plotly)
2. Performance benchmarking
3. v1.0 sprint planning
4. Iterative development with user testing

Success Metrics

Year 1: 10,000 users
Year 2: Adopted in 10 university DSP courses
Year 3: Community library with 100+ demos
Year 5: Standard tool for DSP education

Last Updated: October 26, 2025
Status: Research complete, ready for implementation

Competitive Analysis and Market Positioning

This document analyzes 8 web-based educational/visualization platforms to understand SignalShow's unique market position.

Platforms Analyzed

1. **Observable** - Data visualization notebooks (professional focus)
2. **Desmos** - Graphing calculator (K-12 education)
3. **GeoGebra** - Dynamic mathematics (geometry + algebra)
4. **Mathigon** - Interactive textbooks (K-12 narrative)
5. **PhET** - Research-based STEM simulations (physics, chemistry)
6. **Wolfram Demonstrations** - Mathematica-powered interactive library
7. **J-DSP / CloudDSP** - Browser-based DSP laboratory (Arizona State)
8. **Falstad Applets** - Lightweight HTML5 explorations (Fourier, circuits, waves)

Key Findings

Market Gap

None of these platforms specialize in signal processing education:

- Observable: General data visualization, code-first
- Desmos: K-12 algebra/calculus graphing
- GeoGebra: Geometry and general mathematics
- Mathigon/PhET: Broad STEM topics
- Wolfram: Demonstrations across all STEM fields
- J-DSP: Legacy Java, limited modern features
- Falstad: Lightweight but not comprehensive

SignalShow's unique position: Only modern, web-based platform focused specifically on DSP/optics education with publication-quality output.

Technology Insights

Frontend:

- Observable, Desmos, GeoGebra: JavaScript/TypeScript
- PhET: HTML5 + JavaScript (formerly Flash/Java)
- Falstad: Minimal HTML5/Canvas

Visualization:

- D3.js: Observable, Mathigon
- Custom engines: Desmos (proprietary), GeoGebra (Java-based WebGL)
- Canvas API: Falstad, PhET

Architecture:

- Client-only: Desmos, Falstad, PhET

- Backend integration: Observable (database connectors), J-DSP (server processing)
- Hybrid: GeoGebra (optional cloud features)

Competitive Advantages

SignalShow vs. Competitors:

Feature	SignalShow	Observable	Desmos	GeoGebra	J-DSP
DSP Focus	✅ Specialist	❌ General	❌ Algebra	❌ General Math	✅ DSP only
Modern Stack	✅ React/Julia	✅ JS/TS	✅ Modern	⚠️ Java legacy	❌ Legacy Java
Publication Export	✅ High-DPI	✅ Yes	⚠️ Limited	✅ Yes	❌ Limited
3D Graphics	✅ Three.js	✅ D3/Observable Plot	❌ 2D only	✅ WebGL	❌ 2D only
Optics Simulation	✅ Planned	❌ No	❌ No	⚠️ Limited	❌ No
GUI-First	✅ Yes	❌ Code-first	✅ Yes	✅ Yes	✅ Yes
Free/Open	✅ Open source	⚠️ Freemium	✅ Free	✅ Open source	✅ Free

---

Lessons for SignalShow

From Observable

Strengths to emulate:

- Shareable URLs (state encoded in URL parameters)
- Collaborative features (future: multiplayer editing)
- Beautiful, minimal UI with dark mode
- Embeddable outputs

Avoid:

- Code-first barrier (SignalShow should be GUI-first)
- Complexity for beginners

From Desmos

Strengths to emulate:

- Extreme simplicity and polish
- Instant responsiveness (<16ms interactions)
- Touch-optimized UI for tablets
- Keyboard shortcuts for power users

- "Activities" for guided learning

Adopt:

- Dual input: Sliders AND text input
- Auto-scaling axes with smart defaults
- Clean, uncluttered interface
- Mobile/tablet support

From GeoGebra

Strengths to emulate:

- Community library (thousands of user-created demos)
- Multi-platform (web, desktop, mobile)
- Extensive documentation and tutorials
- Classroom integration features

Avoid:

- Feature creep (GeoGebra tries to do everything)
- Complex UI for advanced features

From J-DSP (Direct Competitor)

SignalShow advantages:

- Modern React vs. legacy Java applets
- 3D visualization (J-DSP is 2D only)
- Publication-quality exports
- Beautiful UI (J-DSP has dated block-diagram interface)
- Offline desktop app option

J-DSP strengths to match:

- Comprehensive DSP coverage
- Educational demos aligned to curriculum
- Free and widely adopted in universities

Strategic Positioning

Elevator Pitch

"SignalShow is to DSP what Desmos is to algebra"

SignalShow fills a clear market gap as the only modern, web-based tool for signal processing and optics education that combines:

- Beautiful, intuitive UI (like Desmos)
- Publication-quality exports (like Observable)
- 3D visualization (like GeoGebra)
- Specialist DSP/optics focus (unique)

Target Partnerships

1. **Textbook Publishers**
    - Companion interactive demos (like Wolfram Demonstrations)
    - Embedded in digital textbooks
2. **OpenStax**
    - Integration into open-source DSP textbooks
    - Free, accessible education
3. **IEEE Education Society**
    - Workshop co-sponsorship
    - Community demo development
4. **ABET**
    - Align concept packs with accreditation outcomes
    - Demonstrate student learning objectives

Success Metrics

Based on comparable platforms:

Year 1: 10,000 users (professors + students)

- Desmos: 100M+ users (but K-12 focus, 10+ years)
- GeoGebra: 100M+ users (general math, 20+ years)
- Observable: ~200k users (professional focus, 5 years)

Year 2: Adopted in 10 university DSP courses

- J-DSP: Used in ~50 universities globally
- PhET: Used in ~100M simulations/year

Year 3: Community library with 100+ demos

- GeoGebra: 1M+ community resources
- Wolfram: 10k+ demonstrations

Year 5: Standard tool for DSP education globally

Technical Takeaways

Proven Technologies:

- React/TypeScript for UI (Observable, modern tools)
- Plotly.js or D3.js for scientific visualization
- WebGL for 3D (GeoGebra, Three.js)
- URL state encoding for sharing (Desmos, Observable)

Avoid:

- Flash/Java applets (PhET migrated away, J-DSP stuck with legacy)
- Server-side rendering for real-time (keep computation client-side or backend)
- Over-complexity (GeoGebra's 400+ features can overwhelm)

Best Practices:

- Mobile/touch support from day one (Desmos, GeoGebra)
- Keyboard shortcuts for classroom presentations
- Dark mode for long sessions
- Community demo library (GeoGebra model)
- Guided tutorials (Mathigon, PhET)

Key Insight: No platform currently combines Desmos's polish, Observable's exports, and signal processing specialization. This is SignalShow's market opportunity.

Strategic Recommendations

Purpose: Long-term product vision and roadmap guardrails for SignalShow modernization

1. Product Positioning

Core identity: Signal processing + optics + pedagogy

Differentiation:

- Deep frequency-domain tools (Fourier, wavelets, holography) with live explanations
- Coupled 1D ↔ 2D workflows (time traces → diffraction patterns)
- Reproducible figure pipelines for educators and authors

Signature experiences:

1. **Concept Studio** - Curated interactive narratives (sampling theorem, interferometry, modulation)
2. **Optics Lab** - 2D aperture & propagation playground with exportable animations
3. **Signal Clinic** - Guided diagnosis of noisy/distorted signals

Partnership angle: Complement MATLAB/Python workflows with instant visual explainers that export back to those ecosystems

2. Personas & Expectations

Persona	Day-One Goals	Advanced Needs	Platform Implications
Instructor	Launch pre-built demos; annotate live; export slides	Build custom sequences, push to LMS	Offline availability, hotkeys, shareable lesson bundles
Student	Explore parameters safely; capture lab notes	Compare theory vs measured data, collaborate	Guardrails, scaffolded steps, sandbox mode, cross-device sync
Researcher/Author	Publication-quality figures with provenance	Re-run pipelines when data changes	Versioned exports, metadata, CLI/headless mode
Content Creator	Script storyboards, export video narration	Integrate with editing suites	JSON timeline, Manim/After Effects bridges

Recommendation: UI layers scaling from Guided Mode (step-by-step) to Expert Mode (full control + scripting)

3. Experience Architecture

Progressive Onboarding:

- First-run tour: signal selection, operations chain, comparison panes
- Contextual help tied to textbook chapters/IEEE curriculum

Workspace Metaphor:

- Tabbed workspaces (Time, Frequency, Spatial, Parameter Study) with synchronized cursors
- Split-view comparisons (Original vs Processed vs Reference)

Narrative Timeline:

- Timeline strip recording parameter changes → replay/export
- Export as JSON/Markdown for lesson plans

Auditory Feedback:

- Optional sonification (play waveform before/after)
- Accessibility support for visually impaired users

---

4. Content & Curriculum Strategy

Concept packs aligned to ABET-accredited DSP courses:

- Interactive demos
- Instructor talking points
- Student exercises with answer keys
- Real-world datasets (audio, optics, biomedical)

Lesson template format: Markdown + JSON (shareable, version-controlled)

Partnerships: OpenStax, IEEE TryEngineering for curated material

---

5. Interoperability & Data Standards

File formats:

- Import/export: WAV, CSV, NumPy `.npy`, MATLAB `.mat`, HDF5
- Preserve metadata (sample rate, units, processing chain)

External pipeline hooks:

- CLI for headless rendering (CI/LaTeX workflows)
- Jupyter/Pluto integration (Python/Julia client libraries)
- LTI 1.3 deep-linking for LMS (Canvas, Blackboard)

Plugin API:

- Sandboxed JS/WASM modules for custom operations
- Universities can extend without forking

---

6. Performance & Technical Roadmap

Tier 1 (v1.0): JavaScript backend

- Target: <100ms FFT (4096 samples)
- Sufficient for teaching demos
- Zero installation

Tier 2 (v1.5): Rust WASM

- Target: <5ms FFT (4096 samples)
- Near-native performance
- Advanced filtering

Tier 3 (v2.0): Julia server (desktop optional)

- Target: <1ms FFT (4096 samples)
- Complete DSP library
- Research-grade computations

Progressive enhancement: Start browser-only → Add WASM → Optionally enable Julia

---

7. Sustainability & Business Model

Open core model:

- Core engine: Open source (MIT/Apache)
- Premium features: Institutional licensing
  - LMS integration
  - Headless rendering
  - Priority support

Revenue streams:

1. University site licenses ($500-2000/year)
2. Textbook publisher partnerships (bundled access codes)
3. Professional training workshops
4. Custom consulting for industry

Community building:

- GitHub for development
- Discord for support/discussion
- Annual conference/workshop (virtual)

---

8. Risk Mitigation

Technical risks:

- Browser performance limits → Progressive backend tiers
- Browser API changes → Feature detection + fallbacks
- Julia server complexity → Make optional in v2.0

Market risks:

- MATLAB dominance → Position as complementary, not replacement
- Free alternatives (Python) → Emphasize ease-of-use, pedagogy focus
- Institutional inertia → Pilot programs, faculty champions

Execution risks:

- Scope creep → Strict v1.0 feature freeze

- Developer burnout → Sustainable roadmap, community contributions
- Documentation debt → Write docs concurrently with code

---

9. Success Metrics

v1.0 (6 months):

- 100 active users (students/instructors)
- 10 universities piloting
- 50% feature parity with Java version
- <2 second load time

v1.5 (12 months):

- 1,000 active users
- 50 universities
- 85% feature parity + modern features
- First textbook integration

v2.0 (18 months):

- 5,000 active users
- 200 universities
- 100% feature parity + advanced features
- 5 textbook partnerships
- Self-sustaining community

---

10. Go-to-Market Strategy

Phase 1: Academic Validation (Months 1-6)

- Beta with 3-5 friendly universities
- Gather feedback from instructors/students
- Refine based on classroom use

Phase 2: Controlled Launch (Months 7-12)

- Public release (web + desktop)
- Conference presentations (IEEE, ASEE)
- Academic journal publications
- Social media campaign

Phase 3: Scale (Months 13-18)

- LMS integrations
- Textbook partnerships
- International expansion
- Industry pilots

Marketing channels:

- Academic conferences
- Faculty mailing lists

- IEEE Education Society
- Reddit (r/DSP, r/AcademicProgramming)
- YouTube tutorials

---

11. Competitive Advantages

vs MATLAB:

- Free/low-cost
- Instant browser access (no installation)
- Better pedagogy focus
- Exportable to MATLAB workflows

vs Python (NumPy/SciPy):

- No coding required
- Guided learning experience
- Instant visualization
- Lower barrier to entry

vs Desmos/GeoGebra:

- Domain-specific (DSP/optics)
- Advanced signal processing
- Research-grade computations
- Optics simulation

Unique value: Only tool combining visual DSP pedagogy + optics + reproducible pipelines + multi-backend performance

---

12. Key Decisions for v1.0

Must have:

- 15 core signal generators
- 20 essential operations
- FFT visualization
- File I/O (.sig1d, WAV, CSV)
- Basic plotting (Plotly.js)
- Desktop app (Tauri)

Nice to have (defer to v1.5):

- STFT/spectrograms
- Advanced filters
- 2D operations
- WASM backend
- LMS integration

Explicitly exclude (defer to v2.0):

- Julia server
- Wavelet analysis

- Machine learning
- Video export
- 3D visualization

Quality bar:

- All features fully documented
- Test coverage >80%
- Load time <2 seconds
- No critical bugs
- Accessible (WCAG 2.2 AA)

Bottom line: Focus v1.0 on rock-solid core DSP pedagogy. Add modern features incrementally in v1.5/v2.0 based on user feedback. Prioritize sustainability over feature count.

User Personas and Use Cases

Purpose: Define target users to guide feature prioritization

Persona Distribution

- **Instructor**: 40%
- **Student**: 45%
- **Researcher/Author**: 10%
- **Content Creator**: 5%

1. Dr. Elena Martinez - The Instructor

Profile: Associate Professor, mid-size university, teaches DSP courses (45-60 students)

Primary Goal: Make abstract DSP concepts tangible through interactive demonstrations

Key Needs:

- Pre-built concept packs for common topics
- Presentation mode with keyboard shortcuts
- Quick demo setup (<5 minutes)
- LMS integration for student access
- Student progress tracking

Pain Points:

- MATLAB: Expensive, clunky GUI, code-first barrier
- Python/Jupyter: Installation issues, not presentation-friendly
- Desmos/GeoGebra: Lack DSP-specific operations

Critical Features:

- Fullscreen presentation mode
- Hotkey navigation
- Real-time annotations (arrows, text)
- PNG/SVG export for slides
- Shareable URLs
- Workspace review (verify student work)

Success Metrics:

- Demo prep time: <5 minutes (vs. 30 min with MATLAB)
- Student engagement: 80%+ find demos helpful
- Adoption in 3+ courses by semester 2

2. Alex Chen - The Student

Profile: Junior EE major, taking DSP course, visual learner, moderate programming skills

Primary Goal: Build intuition for signal processing through hands-on exploration

Key Needs:

- Guided mode with parameter constraints
- Undo/redo for fearless experimentation
- WAV/CSV import for homework datasets
- Lab report export (figures + workspace JSON)
- Example gallery with explanations

Pain Points:

- MATLAB: Confusing syntax errors, expensive license
- Python: Environment setup headaches
- Theory-first textbooks: Hard to connect equations to reality

Critical Features:

- Guided tutorials with hints
- Visual parameter sliders
- Real-time preview
- Error recovery (undo/redo)
- Data import (WAV, CSV)
- Export figures for reports
- Mobile/tablet access

Success Metrics:

- Uses tool voluntarily outside assignments
- Homework completion time reduced 30%
- Conceptual quiz scores improve 15%

---

3. Dr. Raj Patel - The Researcher/Author

Profile: Research scientist, writes papers on signal processing, needs publication-quality figures

Primary Goal: Generate reproducible, publication-quality figures with provenance tracking

Key Needs:

- High-DPI exports (300+ DPI PNG/SVG)
- CLI for batch figure generation
- Expert mode (no guardrails)
- Python/Julia API for scripting
- Metadata embedding in exports

Pain Points:

- MATLAB: Figures look dated, hard to customize
- Python matplotlib: Time-consuming styling
- Illustrator: Manual work, not reproducible

Critical Features:

- High-resolution export

- Custom styling (fonts, colors)
- Headless CLI mode
- Provenance metadata
- Batch processing
- API for automation

Success Metrics:

- Publishes ≥1 paper with SignalShow figures
- Figure generation time reduced 50%
- Reproducibility: Can regenerate figures from saved config

---

4. Maya Rodriguez - The Content Creator

Profile: Educational YouTuber, creates DSP tutorials, 50k subscribers

Primary Goal: Create visually stunning educational videos explaining DSP concepts

Key Needs:

- Timeline recording of parameter changes
- Manim export for 3Blue1Brown-style animations
- Custom branding (colors, fonts)
- 4K/8K rendering
- Smooth animations

Pain Points:

- Manual animation is tedious
- Existing tools lack DSP depth
- After Effects learning curve steep

Critical Features:

- Timeline recording
- Video export (MP4, WebM)
- Manim integration
- Custom color schemes
- High-resolution rendering
- Smooth parameter transitions

Success Metrics:

- Produces video with 50%+ time savings
- Animation quality rivals 3Blue1Brown
- Creates ≥2 tutorials/month using SignalShow

Feature Mapping by Persona

Feature	Instructor	Student	Researcher	Creator
Presentation mode	⭐ Critical	-	-	-

Guided tutorials	⭐ Critical	⭐ Critical	-	-
Pre-built demos	⭐ Critical	✅ Important	-	✅ Important
LMS integration	⭐ Critical	-	-	-
Data import (WAV/CSV)	✅ Important	⭐ Critical	⭐ Critical	-
High-DPI export	✅ Important	✅ Important	⭐ Critical	⭐ Critical
CLI/headless mode	-	-	⭐ Critical	-
Timeline recording	-	-	-	⭐ Critical
Manim export	-	-	-	⭐ Critical
Undo/redo	✅ Important	⭐ Critical	✅ Important	✅ Important
Mobile/tablet	✅ Important	⭐ Critical	-	-
Custom branding	-	-	✅ Important	⭐ Critical

Legend: ⭐ Critical | ✅ Important | - Not needed

Implementation Priorities

v1.0 (Months 1-6) - Focus on Instructor + Student:

- Pre-built concept packs (10+ demos)
- Presentation mode + hotkeys
- Guided tutorials
- Data import (WAV/CSV)
- PNG/SVG export
- Undo/redo
- Shareable URLs

v1.5 (Months 7-9) - Add Researcher features:

- High-DPI export (300+ DPI)
- Custom styling
- Metadata embedding
- Batch processing

v2.0 (Months 10-12) - Add Creator features:

- Timeline recording
- Video export
- Manim integration
- Custom branding

Key Insight: First two personas (Instructor + Student) represent 85% of user base and share many feature needs, making them ideal v1.0 focus.

Roadmap and Implementation Priorities

Date: October 26, 2025
Purpose: Integration of strategic recommendations into development timeline

---

Strategic Adjustments

Based on expert review and competitive analysis of 8 platforms, key additions to the modernization plan:

1. **Product Positioning**: "SignalShow is to DSP what Desmos is to algebra"
2. **Persona-Driven Development**: Design for 4 user types (Instructor, Student, Researcher, Creator)
3. **Interoperability First**: WAV/CSV/MATLAB import/export in v1.0
4. **Accessibility**: WCAG 2.2 AA from day one
5. **Progressive Disclosure**: Guided mode → Expert mode
6. **Timeline/Provenance**: Record and replay parameter changes
7. **Plugin Architecture**: University extensibility
8. **Community Library**: Plan from v1.0

Revised Timeline

v1.0 - Public Release (Months 1-6)

Core Platform:

- React 19 + Plotly.js visualization
- JavaScript DSP backend (zero installation)
- Zustand state management
- shadcn/ui components

Essential Features:

- 6-8 signal generators (sine, cosine, chirp, Gaussian, rect, delta)
- 8-10 operations (FFT, filters, convolution, correlation, windowing)
- Interactive waveform plots
- Parameter sliders with real-time preview
- PNG/SVG export

Accessibility (NEW):

- WCAG 2.2 AA compliance
- Keyboard navigation
- Screen reader support
- Colorblind-safe palettes

Interoperability (NEW):

- WAV/CSV import/export
- JSON workspace format
- Metadata embedding

Educational (NEW):

- Guided mode with hints
- 10 pre-built demos
- Presentation mode (fullscreen + hotkeys)

Deployment:

- Static web app (Vercel/Netlify)
- PWA for offline UI

Success Metrics:

- 1,000 users in first 3 months
- Used in 2 university courses

---

v1.5 - Classroom Release (Months 7-9)

Performance (NEW PRIORITY):

- Rust/WASM DSP kernels (rustfft, dasp)
- Web Workers for background processing
- 60-95% of native performance

Complete Feature Set:

- All 80+ signal generators
- All 40+ operations
- Time-frequency analysis (STFT, spectrograms, wavelets)

Educational Enhancements:

- LMS integration (Canvas, Blackboard)
- Student progress tracking
- Custom demo authoring
- Timeline recording/replay

3D Visualization (MOVED from v2.0):

- 2D FFT as 3D surfaces (Three.js)
- Holographic patterns
- Filter responses

Advanced Export:

- High-DPI (300+ DPI)
- Batch processing
- CLI for automation

Deployment:

- Docker classroom appliance
- SSO integration

Success Metrics:

- 10,000 users

- 10 university partnerships
- 50+ community demos

---

v2.0 - Desktop + Extensions (Months 10-12)

Desktop App:

- Tauri with bundled Julia runtime
- Auto-updates
- Native file I/O

Video Production:

- Manim export
- Timeline → animation pipeline
- 4K rendering

Plugin Ecosystem:

- Sandboxed plugin API
- Community marketplace
- University extensions

Advanced Computation:

- Optional Julia server backend
- GPU acceleration (WebGPU)
- Large dataset support (>10M samples)

Success Metrics:

- Desktop: 5,000 downloads
- Video: 100+ educational videos created
- Plugins: 20+ community extensions

---

v2.5+ - Platform Maturity

Exercise System:

- Student assignments
- Auto-grading
- Classroom management

Collaboration:

- Real-time multiplayer editing
- Cloud save with accounts
- Shared workspaces

Community:

- SignalShow Commons library
- Peer review system
- Quarterly workshops

Feature Prioritization Framework

v1.0 Criteria (Must Ship)

Feature	Instructor	Student	Researcher	Creator	Priority
Presentation mode	⭐	-	-	-	**P0**
Guided tutorials	⭐	⭐	-	-	**P0**
WAV/CSV import	✓	⭐	⭐	-	**P0**
Basic DSP ops	⭐	⭐	✓	✓	**P0**
PNG export	⭐	✓	✓	✓	**P0**
Keyboard shortcuts	⭐	-	✓	-	**P1**
Mobile/tablet	✓	⭐	-	-	**P1**

Legend: ⭐ Critical | ✓ Important | - Not needed

Development Phases

Months 1-2: Foundation

- React architecture
- JavaScript DSP backend
- Basic UI components
- 4 signal types implemented

Months 3-4: Core Features

- All v1.0 functions and operations
- Plotly.js visualization
- Export capabilities
- Accessibility compliance

Months 5-6: Polish & Launch

- Pre-built demos
- Documentation
- Performance optimization
- Beta testing with 2 universities

Months 7-9: Performance & Scale

- WASM implementation
- 3D visualization
- LMS integration
- Community library

Months 10-12: Desktop & Video

- Tauri app
- Julia backend option
- Manim export
- Plugin system

Risk Mitigation

Risk	Mitigation
Performance with JavaScript DSP	Ship WASM in v1.5; progressive enhancement
Accessibility gaps	Include disabled users in beta testing
LMS integration complexity	Partner with 2 universities early
Plugin ecosystem fragmentation	Versioned API, certification program
Community content quality	Peer review, moderation queue

Partnership Strategy

v1.0 Launch Partners (2 universities):

- Co-develop concept packs
- Beta testing with real students
- Feedback on workflows

v1.5 Expansion (10 universities):

- IEEE Education Society sponsorship
- OpenStax integration
- Textbook publisher demos

v2.0 Ecosystem (Community):

- Plugin marketplace
- Content creator partnerships
- Conference workshops

Success Metrics by Version

v1.0:

- 1,000 users (first 3 months)
- 80% accessibility score
- 2 university courses

v1.5:

- 10,000 users
- 10 university partnerships
- 50+ community demos

- 90% accessibility score

v2.0:

- 50,000 users
- 50 universities
- 500+ demos
- 100+ educational videos

v3.0 (Year 2):

- 100,000 users
- Standard tool for DSP education
- 1,000+ demos
- IEEE endorsement

Implementation Principles

1. **Persona-first**: Design every feature for specific user workflows
2. **Accessibility-native**: Not retrofitted, built in from start
3. **Community-driven**: User contributions from v1.0
4. **Performance-conscious**: Set budgets, track regressions
5. **Interoperable**: Work with existing tools, don't replace them
6. **Reproducible**: Every workspace exportable with full provenance

---

Key Change: Shifted from "build it and they will come" to persona-driven, accessibility-first, community-enabled platform from day one.

System Architecture

Purpose: Technical design for SignalShow modernization

Design Principles

Based on analysis of Desmos, Observable, and GeoGebra:

1. **Shareable URLs** - Encode state in URL for instant sharing
2. **Real-time interaction** - Zero-latency slider feedback
3. **Touch support** - Works on tablets/iPads (44px minimum targets)
4. **Keyboard shortcuts** - Power user efficiency
5. **Clean UI** - Minimal, professional aesthetic
6. **Universal export** - PNG/SVG/JSON/MP4 one-click
7. **Community library** - User-submitted demos

System Overview

Layers:

1. **Presentation** - React + TypeScript UI
2. **State** - Zustand store with file-based persistence
3. **Computation** - Hybrid backend (Julia/JavaScript/WASM)
4. **Visualization** - Plotly.js + D3.js + Three.js

File Formats:

- `.sig1d` - 1D signals (JSON)
- `.sig2d` - 2D images (JSON)
- `.sigOp` - Operation configs
- `.sigWorkspace` - Complete sessions
- `.sigDemo` - Pre-built examples

Backend Abstraction

Strategy: Automatic environment detection

```
const backend = window.__TAURI__
  ? new JuliaServerBackend()  // Desktop: optimal
  : new WebBackend();          // Browser: zero install
```

Desktop Mode (Tauri):

- Julia server (localhost:8080)
- 100% performance, complete features
- Graceful fallback to JavaScript

Web Mode (Browser):

- Pure JavaScript (Math.sin, fft.js)
- 5-10% native performance
- ~60% feature coverage
- Zero installation

Future: Rust + WASM (60-95% performance)

Component Architecture

Core React Components:

1. **FunctionGenerator**
    - Signal type selector
    - Parameter sliders
    - Real-time preview
    - Preset library
2. **OperationsPanel**
    - Operation selector (FFT, filters, etc.)
    - Operation chain builder
    - Parameter configuration
3. **SignalDisplay**
    - Plotly.js waveform plots
    - Time/frequency domain
    - Zoom, pan, annotations
    - Export controls
4. **DemoGallery**
    - Pre-built examples
    - Topic browsing
    - One-click load
5. **WorkspaceManager**
    - File I/O (save/load)
    - Import (WAV/CSV)
    - Export (PNG/SVG/JSON)
    - Undo/redo history

Data Flow

User Interaction:

```
User adjusts slider
  → Zustand store update
  → Backend computation (debounced)
  → Plot update (optimistic UI)
```

Operation Chain:

```
Signal → Operation 1 → Operation 2 → Result
                ↓
         Intermediate cached
```

File Operations:

```
Load → Parse JSON → Restore state → Render
Save → Serialize state → Write JSON
```

API Design

Julia Backend (HTTP + WebSocket)

REST Endpoints:

- `POST /api/generate/sine` - Generate signal
- `POST /api/operations/fft` - Compute FFT
- `POST /api/operations/filter` - Apply filter
- `GET /health` - Server status

WebSocket:

- Real-time streaming for heavy operations
- Progress updates for long computations
- Binary transfer for large arrays

Data Format:

```json
{
  "type": "sine",
  "params": {"frequency": 440, "duration": 1.0},
  "samples": [0.0, 0.1, ...],
  "sampleRate": 8000
}
```

JavaScript Backend

Internal API (same interface):

```javascript
await backend.generateSine(440, 1.0, 8000);
await backend.fft(signal);
await backend.filter(signal, 'lowpass', {cutoff: 1000});
```

State Management

Zustand Store:

```
{
  signals: {
    'sig1': {data: [...], sampleRate: 8000},
    'sig2': {data: [...], sampleRate: 8000}
  },
  operations: [
    {type: 'fft', params: {...}, result: 'sig2'}
  ],
  ui: {
    selectedSignal: 'sig1',
    plotMode: 'time-frequency'
  }
}
```

Persistence:

- Auto-save to localStorage (web)
- Native file system (desktop)
- URL encoding for sharing

Performance

Targets:

- Slider interaction: <16ms (60fps)
- FFT (4096 samples): <100ms
- Plot render: <50ms
- Startup: <2 seconds

Optimization:

- Debounce heavy computations
- Web Workers for background processing
- Signal decimation for large datasets
- Plotly.js streaming mode
- WebGL rendering for 3D

Budgets:

Operation	Target	JavaScript	Julia	WASM (planned)
FFT 4096	<100ms	~100ms	~1ms	~5ms
Filter	<50ms	~50ms	~1ms	~3ms
Render	<50ms	~30ms	N/A	N/A

Deployment

Web (Static Hosting)

- React build → Vercel/Netlify
- PWA with service worker (offline UI)
- No backend server needed
- CDN for assets

Desktop (Tauri)

- Single installer: DMG/MSI/AppImage
- Optional Julia runtime (on-demand download)
- Auto-updates
- Native file I/O

Classroom (Docker)

- Pre-configured environment
- Julia server + web UI
- LMS integration ready
- Multi-user support

Security

Web Mode:

- Client-side only (no server)
- CSP headers
- HTTPS required

Desktop Mode:

- Julia server localhost only
- No external network access
- Sandboxed file operations

Future Considerations:

- Plugin sandboxing (WASM)
- Cloud save encryption
- SSO integration

Extensibility

Plugin System (v2.0):

```
// Custom operation plugin
export default {
  name: 'MyFilter',
  operation: async (signal, params) => {
    // Custom DSP logic
    return processedSignal;
  }
}
```

UI Extensions:

- Custom visualizations
- New signal generators
- Export formats

Testing Strategy

Unit Tests:

- Backend abstraction layer
- Signal generators
- Operations (compare Julia vs JS)

Integration Tests:

- End-to-end workflows
- File I/O
- API contracts

Performance Tests:

- Benchmark suite
- Regression tracking
- Browser compatibility

Visual Regression:

- Plot rendering consistency
- UI component screenshots

Migration from Java

Approach: Incremental migration

Phase 1 - Core functions (v1.0):

- 6-8 essential generators
- 8-10 basic operations
- Simple 2D plots

Phase 2 - Complete library (v1.5):

- All 80+ functions
- All 40+ operations
- 3D visualization

Phase 3 - Advanced features (v2.0):

- Desktop app
- Julia backend

- Plugin system

Code Reuse:

- Reference Java for algorithms
- Port formulas to Julia/JavaScript
- Maintain test compatibility

Technical Debt Management

Avoid:

- Monolithic components
- Hard-coded configurations
- Tight coupling to backends
- Browser-specific code

Best Practices:

- TypeScript for type safety
- ESLint + Prettier
- Component testing
- Documentation as code
- Semantic versioning

Key Architecture Decisions:

1. Backend abstraction enables web/desktop from single codebase
2. File-based data model ensures reproducibility
3. Hybrid computation supports both power users (Julia) and accessibility (JavaScript)
4. Progressive enhancement from JavaScript → WASM → Julia

Technology Stack

Purpose: Technology selection for SignalShow modernization

Core Stack

Frontend:

- React 19 + TypeScript
- shadcn/ui components (Radix UI base)
- Tailwind CSS

State Management:

- Zustand (lightweight, performant)

Computation:

- Hybrid backend (Julia/JavaScript/WASM)
- Backend abstraction layer

Visualization:

- Plotly.js (2D scientific plots)
- D3.js (custom animations)
- Three.js + react-three-fiber (3D)

Desktop:

- Tauri (v2.0+)
- Native file I/O

Backend Strategy

Three-Tier Approach

Tier 1: JavaScript (v1.0 - Production Ready)

- Pure browser implementation
- Math.sin/cos for signal generation
- fft.js library (192k downloads/week)
- Direct convolution algorithms
- Bundle: ~23KB
- Performance: 5-10% of native
- Features: ~60% coverage
- **Advantage**: Zero installation

Tier 2: Rust + WebAssembly (v1.5 - Planned)

- rustfft (FFT optimized)
- dasp (DSP operations)
- ndarray (array processing)

- Bundle: ~150KB
- Performance: 60-95% of native
- Features: ~85% coverage
- **Advantage**: Near-native speed in browser

Tier 3: Julia Server (v2.0 - Desktop Optional)

- HTTP.jl + WebSockets.jl
- DSP.jl, FFTW.jl, SpecialFunctions.jl
- localhost:8080
- Performance: 100% (native)
- Features: 100% coverage
- **Advantage**: Complete DSP library

Backend Abstraction

Automatic selection:

```
const backend = window.__TAURI__
  ? new JuliaServerBackend()  // Desktop (if available)
  : new WebBackend();          // Browser (always works)
```

Unified API (same interface for all backends):

```
await backend.generateSine(440, 1.0, 8000);
await backend.fft(signal);
await backend.filter(signal, 'lowpass', {cutoff: 1000});
```

Visualization Libraries

Plotly.js (Primary 2D)

- **Stars**: 17k
- **Use**: Scientific plots, waveforms, spectrograms
- **Exports**: PNG, SVG, PDF
- **Performance**: 60fps for <10k points
- **Bundle**: ~3MB (use plotly.js-dist-min)

D3.js (Custom Animations)

- **Stars**: 108k
- **Use**: Custom visualizations, transitions
- **Animation**: Smooth parameter changes
- **Integration**: Works with React

Three.js + react-three-fiber (3D)

- **Stars**: 103k (Three.js), 30k (R3F)
- **Use**: 3D FFT surfaces, holography, diffraction
- **Performance**: 60fps with WebGL
- **Future**: v1.5 (deferred from v1.0)

Julia DSP Libraries

Core Packages:

- **FFTW.jl** - Fastest FFT implementation
- **DSP.jl** - Filters, windows, spectral analysis
- **SpecialFunctions.jl** - Bessel, erf, gamma functions
- **Distributions.jl** - Noise generation
- **Images.jl** - 2D image operations

HTTP Server:

- **HTTP.jl** - REST API (~200 req/sec)
- **WebSockets.jl** - Real-time streaming
- **JSON3.jl** - Fast serialization

Performance:

- Julia startup: ~3-5 seconds
- First computation (JIT): ~1-2 seconds
- Subsequent calls: <10ms (FFT 4096 samples)

JavaScript DSP Libraries

FFT:

- **fft.js** - Pure JavaScript FFT (192k downloads/week)
- **jsfft** - Alternative implementation
- Performance: ~100ms for 4096 samples

Math:

- **math.js** - General mathematical operations
- **stdlib-js** - Statistical functions, special functions
- Complex number support

Audio:

- **Web Audio API** - Native browser audio processing
- **wavefile** - WAV file I/O

Not Using (unmaintained):

- dsp.js - Last updated 2014
- numeric.js - Last updated 2013

UI Components

shadcn/ui (Recommended):

- Radix UI primitives (accessible)
- Tailwind CSS styling
- Dark mode built-in

- Customizable components
- Tree-shakeable

Key Components:

- Sliders (parameter controls)
- Dropdowns (signal/operation selection)
- Tabs (workspace organization)
- Modals (settings, help)
- Command palette (keyboard shortcuts)

Animation & Video

Web Animations:

- **Framer Motion** (23k stars)
  - Smooth parameter transitions
  - Spring physics
  - Gesture support
  - React integration

Video Export (v2.0):

- **Python Manim** (desktop only)
  - 3Blue1Brown-style animations
  - JSON config → MP4 rendering
  - Requires desktop app with bundled Python

Development Tools

Build:

- Vite (fast dev server, optimized builds)
- TypeScript (type safety)
- ESLint + Prettier (code quality)

Testing:

- Vitest (unit tests)
- Playwright (E2E tests)
- React Testing Library (component tests)

Version Control:

- Git + GitHub
- Semantic versioning
- Conventional commits

Deployment

Web Hosting:

- Vercel / Netlify (static hosting)
- GitHub Pages (alternative)
- PWA with service worker

Desktop Packaging:

- Tauri build system
- DMG (macOS)
- MSI (Windows)
- AppImage (Linux)
- Code signing support

Container:

- Docker (classroom deployment)
- Julia + web UI + nginx
- LMS integration ready

Performance Targets

Operation	Target	JavaScript	Julia	WASM (planned)
Generate signal (4096)	<10ms	~5ms	<1ms	~2ms
FFT (4096)	<100ms	~100ms	~1ms	~5ms
Filter	<50ms	~50ms	~1ms	~3ms
Plot render	<50ms	~30ms	N/A	N/A
Startup	<2s	instant	~5s	instant

Bundle Size Targets

v1.0 (JavaScript):

- React + UI: ~500KB gzipped
- Plotly.js: ~800KB gzipped
- JavaScript DSP: ~23KB
- **Total**: <2MB gzipped

v1.5 (+ WASM):

- Add Rust WASM: ~150KB
- **Total**: ~2.2MB gzipped

v2.0 (Desktop):

- Tauri app: ~600KB
- Julia runtime: ~400MB (optional download)
- **Total installer**: ~1MB (without Julia)

Browser Support

Target:

- Chrome/Edge 90+
- Firefox 88+
- Safari 14+
- Modern mobile browsers

Required Features:

- ES2020
- WebGL (for 3D)
- Web Workers
- IndexedDB / localStorage
- File System Access API (desktop)

Accessibility

Standards:

- WCAG 2.2 Level AA compliance
- Keyboard navigation (all features)
- Screen reader support (ARIA)
- High contrast mode
- Resizable text

Tools:

- axe DevTools (automated testing)
- Screen reader testing (NVDA, VoiceOver)
- Keyboard-only testing

Security

Web:

- CSP headers (content security policy)
- HTTPS only
- No eval() or dangerous innerHTML
- Dependency scanning (npm audit)

Desktop:

- Sandboxed Julia server (localhost only)
- File access controls
- No network access by default
- Code signing (macOS, Windows)

Key Technology Decisions

1. **Hybrid backend** enables progressive enhancement (JavaScript → WASM → Julia)
2. **Plotly.js** for scientific visualization (proven, widely used)
3. **Zustand** over Redux (simpler, less boilerplate)
4. **shadcn/ui** over Material-UI (more customizable, better performance)
5. **Tauri** over Electron (smaller, faster, more secure)
6. **Julia** over Python (faster DSP, better type system, easier parallelization)

Implementation: Start with Tier 1 (JavaScript) for v1.0, add Tier 2 (WASM) in v1.5, offer Tier 3 (Julia) optionally in v2.0 desktop app.

File-Based Architecture

Purpose: Transform SignalShow into a file-based system where all user objects (signals, operations, workspaces) are portable files

---

Current vs Proposed

Java SignalShow (Current):

- All objects exist in memory only
- No persistence between sessions
- Cannot share work easily

Nuthatch SignalShow (Proposed):

- All objects stored as files
- Automatic persistence
- Portable across sessions/systems
- External editing supported
- Version control compatible

Design Goals

1. **Portability** - Share signals, functions, workspaces as files
2. **Persistence** - Work auto-saved and recoverable
3. **Editability** - Power users can edit files externally
4. **Composability** - Files can be combined and referenced
5. **Version Control** - Works with Git
6. **File System Integration** - Leverages Nuthatch Desktop's Files.app
7. **Educational** - Students can examine/modify files to learn

---

File Types

1D Signals (`.sig1d`)

Example: `chirp_signal.sig1d`

```
{
  "type": "signal1d",
  "version": "1.0",
  "metadata": {
    "name": "Chirp Signal",
    "description": "Linear frequency sweep 10Hz to 100Hz",
    "created": "2025-10-25T14:30:00Z",
    "tags": ["chirp", "frequency-sweep"]
  },
  "parameters": {
    "sampleRate": 1000,
    "duration": 1.0,
    "generator": "chirp",
    "generatorParams": {"startFreq": 10, "endFreq": 100, "amplitude": 1.0}
  },
  "data": {
    "format": "json-array",
    "x": [0, 0.001, 0.002, ...],
    "y": [0, 0.0314, 0.0628, ...]
  }
}
```

Alternative compact (large datasets):

```
{
  "data": {
    "format": "binary-base64",
    "encoding": "float32-le",
    "x": "AAAAAAAA8D8AAABAAACAQAAAwEA...",
    "y": "zcxMPZqZmT4AAIA+mpmZPgAA..."
  }
}
```

Use cases:

- User-generated signals (chirps, pulses, noise)
- Imported audio samples
- Intermediate results

2D Signals (`.sig2d`)

Example: `lena_image.sig2d`

```
{
  "type": "signal2d",
  "version": "1.0",
  "metadata": {"name": "Lena Image", "tags": ["image", "test-pattern"]},
  "parameters": {
    "width": 512,
    "height": 512,
    "channels": 1,
    "colorSpace": "grayscale",
    "generator": "imported",
    "sourceFile": "lena.png"
  },
  "data": {
    "format": "binary-base64",
    "encoding": "uint8",
    "values": "iVBORw0KGgoAAAANSUhEUg..."
  }
}
```

Use cases:

- Imported images
- 2D FFT results
- Spectrograms
- Diffraction patterns

Operation Chains (`.sigop`)

Example: `lowpass_filter_chain.sigop`

```json
{
  "type": "operation-chain",
  "version": "1.0",
  "metadata": {"name": "Lowpass Filter + Normalize", "tags": ["filter"]},
  "operations": [
    {
      "id": "op1",
      "type": "filter",
      "params": {"filterType": "butterworth", "order": 4, "cutoff": 1000}
    },
    {
      "id": "op2",
      "type": "normalize",
      "params": {"method": "peak", "targetAmplitude": 1.0}
    }
  ]
}
```

Use cases:

- Reusable processing pipelines
- Teaching examples (convolution, FFT, filtering)
- Batch processing recipes

Workspaces (`.sigws`)

Example: `lecture_5_fourier_analysis.sigws`

```json
{
  "type": "workspace",
  "version": "1.0",
  "metadata": {"name": "Lecture 5: Fourier Analysis", "created": "2025-10-25T09:00:00Z"},
  "signals": [
    {"id": "sig1", "file": "chirp_signal.sig1d", "position": {"x": 100, "y": 50}},
    {"id": "sig2", "file": "square_wave.sig1d", "position": {"x": 100, "y": 200}}
  ],
  "operations": [
    {"id": "fft1", "type": "fft", "input": "sig1", "output": "fft_result"}
  ],
  "plots": [
    {"id": "plot1", "type": "waveform", "source": "sig1", "position": {"x": 400, "y": 50}},
    {"id": "plot2", "type": "spectrum", "source": "fft_result", "position": {"x": 400, "y":
200}}
  ]
}
```

Use cases:

- Lecture demonstrations
- Student assignments
- Complex multi-signal analysis

Project Files (`.sigproj`)

Example: `acoustic_analysis_project.sigproj`

```
{
  "type": "project",
  "version": "1.0",
  "metadata": {"name": "Acoustic Analysis Project", "description": "Room impulse response
study"},
  "structure": {
    "signals/": ["microphone_1.sig1d", "microphone_2.sig1d"],
    "operations/": ["noise_reduction.sigop", "cross_correlation.sigop"],
    "results/": ["impulse_response.sig1d", "frequency_response.sig1d"],
    "workspaces/": ["analysis_workspace.sigws"]
  },
  "settings": {
    "defaultSampleRate": 48000,
    "audioBackend": "julia-server"
  }
}
```

Use cases:

- Complex multi-file projects
- Research work
- Student portfolios

File Operations

Create

```
// Generate signal → auto-save as .sig1d
const signal = await backend.generateChirp(10, 100, 1.0, 1000);
await fileManager.save(signal, 'chirp_signal.sig1d');
```

Load

```
// Load .sig1d → display
const signal = await fileManager.load('chirp_signal.sig1d');
plotManager.plotWaveform(signal);
```

Edit

```
// Modify parameters → save
signal.metadata.tags.push('modified');
await fileManager.save(signal, 'chirp_signal.sig1d');
```

Reference

```
// Workspace references signals by path
workspace.addSignal({file: 'signals/chirp_signal.sig1d'});
```

Storage Locations

Desktop App (Tauri)

- **User Files**: `~/Documents/SignalShow/`
- **System ROM**: `/Applications/SignalShow.app/Contents/Resources/system-rom/`
- **Temp Files**: `~/.signalshow/temp/`

File tree:

```
~/Documents/SignalShow/
  Projects/
    acoustic_analysis/
      acoustic_analysis_project.sigproj
      signals/
        microphone_1.sig1d
      operations/
        noise_reduction.sigop
      results/
  Signals/
    chirp_signal.sig1d
    square_wave.sig1d
  Workspaces/
    lecture_5_fourier.sigws
```

Web App (Browser)

- **IndexedDB**: Primary storage (unlimited quota)
- **localStorage**: Settings/preferences (10MB limit)
- **File System Access API**: Optional native file I/O (Chrome/Edge)

IndexedDB structure:

```
{
  "signals": {
    "chirp_signal": {blob, metadata},
    "square_wave": {blob, metadata}
  },
  "workspaces": {
    "lecture_5": {blob, metadata}
  }
}
```

Integration with Nuthatch Desktop

Files.app Integration

- `.sig1d` / `.sig2d` files appear in Files.app
- Double-click opens in SignalShow
- Drag-and-drop support
- Thumbnail previews (waveform icons)

Native File Dialogs

- Use Tauri native dialogs (not browser input)
- File picker shows .sig1d / .sig2d by default
- Save dialog auto-adds extension

Drag-and-Drop

- Drag `.sig1d` from Files.app → SignalShow canvas
- Drag `.wav` / `.png` → auto-convert to `.sig1d` / `.sig2d`
- Drag between SignalShow instances

Import/Export

Import Formats

Format	Extension	Converts To	Notes
WAV audio	`.wav`	`.sig1d`	Sample rate preserved
PNG/JPEG	`.png`, `.jpg`	`.sig2d`	Grayscale conversion
CSV data	`.csv`	`.sig1d`	Two-column (x, y) format

NumPy binary	`.npy`	`.sig1d` / `.sig2d`	1D or 2D arrays

Export Formats

Format	Extension	Exports From	Notes
WAV audio	`.wav`	`.sig1d`	For playback
PNG image	`.png`	`.sig1d`, `.sig2d`	Plot snapshots
CSV data	`.csv`	`.sig1d`	(x, y) columns
JSON	`.json`	All	Human-readable

Version Control

Git-friendly:

- All files are JSON text (diffable)
- Binary data in base64 (not ideal but manageable)
- `.gitignore` template:

```
# SignalShow
.signalshow/temp/
*.sig1d.bak
*.sig2d.bak
```

Large file storage:

- Use Git LFS for large `.sig2d` files
- Recommended for images >1MB
- Configuration:

```
git lfs track "*.sig2d"
```

Performance Considerations

File size targets:

- `.sig1d` (1000 samples): ~10KB JSON, ~4KB binary
- `.sig2d` (512×512 image): ~300KB JSON, ~256KB binary
- `.sigws` (workspace): ~5-20KB (references only)

Loading strategy:

- Lazy load: Load metadata first, data on demand
- Streaming: For large files, load chunks progressively
- Caching: Keep recently used files in memory

Auto-save:

- Debounced writes (500ms after edit)
- Dirty flag tracking
- Background save (non-blocking)

Security

Tauri app:

- File access restricted to `~/Documents/SignalShow/`
- No arbitrary file system access
- Sandboxed Julia server (localhost only)

Web app:

- IndexedDB isolated per-origin
- File System Access API requires user permission
- No automatic network access

File validation:

- JSON schema validation on load
- Reject malformed files
- Sanitize user-provided metadata

Educational Benefits

1. **Transparency** - Students see exact signal parameters in JSON
2. **Experimentation** - Edit files manually to learn effects
3. **Sharing** - Email/post signal files for homework
4. **Reproducibility** - Exact parameters preserved
5. **Version control** - Track changes over time (Git)
6. **Portability** - Works across desktop/web versions

Example use case:

```
Professor creates chirp_example.sig1d
↓
Posts to LMS (Moodle/Canvas)
↓
Students download and open in SignalShow
↓
Students modify parameters
↓
Students submit modified .sig1d files
```

Implementation Phases

Phase 1 (v1.0) - Basic File I/O

- Load/save `.sig1d` files
- JSON format only
- Desktop file dialogs (Tauri)
- IndexedDB storage (web)

Phase 2 (v1.5) - Advanced Features

- `.sig2d` image support
- `.sigop` operation chains
- Binary format (base64 encoding)
- Import/export WAV, PNG, CSV

Phase 3 (v2.0) - Full Integration

- `.sigws` workspaces
- `.sigproj` projects
- Drag-and-drop between apps
- Thumbnail previews
- Auto-save
- Git LFS support

Key Decision: Start with simple JSON `.sig1d` files in v1.0, expand to full file-based architecture in v2.0.

Feature Mapping

Purpose: Map 80+ Java functions and 40+ operations to Julia/JavaScript implementations

---

Implementation Strategy

Three-tier backend:

1. **JavaScript** (v1.0) - Pure browser, ~60% coverage
2. **Rust WASM** (v1.5) - Near-native speed, ~85% coverage
3. **Julia Server** (v2.0) - Complete DSP library, 100% coverage

Priority: v1.0 features marked **bold**

1D Function Generators

Basic Waveforms (v1.0 JavaScript)

Function	Parameters	Implementation	Notes
Sinc	Amplitude, Center, Width	`A * sinc(π/width * (x - center))`	Cardinal sine
Sine	Amplitude, Center, Period, Phase	`A * sin(2π/width * (x - center) + φ)`	Basic sinusoid
Cosine	Amplitude, Center, Period, Phase	`A * cos(2π/width * (x - center) + φ)`	Basic cosinusoid
Chirp	Amplitude, Center, Width, Phase, Exp	`A * cos(π/width^n * (x - center)^n + φ)`	Frequency sweep
Gaussian	Amplitude, Width (σ), Center, Exp	`A * exp(-((x - center)/σ)^n)`	Bell curve
Rectangle	Amplitude, Width, Center	Custom rect function	Rectangular pulse
Triangle	Amplitude, Width, Center	`A * max(0, 1 - abs(x - center)/width)`	Triangular pulse
Delta	Amplitude, Center	Kronecker delta	Discrete impulse
Step	Amplitude, Center	Heaviside step	Step function
Constant	Amplitude	`fill(A, n)`	DC signal
Zero	None	`zeros(n)`	All zeros
Square Wave	Amplitude, Center, Period, Phase	`A * sign(cos(2π/width * (x - center) + φ))`	Periodic pulses
Data	Data array	Load from file/array	User-supplied

v1.5 additions: Complex sinusoid, Line, Monomial, Lorentzian, Comb, Double slit, Besinc

Backend:

- JavaScript: Math.sin/cos, custom functions
- Julia: SpecialFunctions.jl (sinc, Bessel), DSP.jl

Window Functions

Function	Parameters	Implementation	Backend
Hamming	Amplitude, Center, Width	`A * (0.54 + 0.46*cos(π(x-c)/w))` for |x-c|≤w	JavaScript → WASM
Hanning	Amplitude, Center, Width	`A * 0.5 * (1 + cos(π(x-c)/w))` for |x-c|≤w	JavaScript → WASM
Welch	Width, Center	Parabolic window	WASM
Parzen	Width, Center	Piecewise triangular	WASM

Backend:

- JavaScript: Custom implementations
- Rust WASM: DSP library (v1.5)
- Julia: DSP.jl (v2.0)

Noise Functions

Function	Parameters	Distribution	Backend
Gaussian	σ (std dev), μ (mean), seed	N~(μ, σ)	JavaScript (Box-Muller)
Uniform	μ (mean), Half-Width, seed	U~(μ-hw, μ+hw)	JavaScript (Math.random)
Poisson	λ (rate), seed	Poisson(λ)	JavaScript (Knuth) → Julia
Binomial	n (trials), p (prob), seed	Binomial(n, p)	JavaScript → Julia
Exponential	β (mean), seed	Exp(1/β)	JavaScript (inverse transform)
Rayleigh	σ (scale), seed	Rayleigh(σ)	Julia only
Salt & Pepper	density, seed	Impulse noise	JavaScript

Backend:

- JavaScript: Box-Muller (Gaussian), inverse transform (Exponential)
- Julia: Distributions.jl (all distributions, v2.0)

Special Functions

Function	Parameters	Implementation	Backend
Bessel J0	Order, Amplitude, Width, Center	`besselj(order, x)`	Julia only (SpecialFunctions.jl)

Bessel Y0	Order, Amplitude, Width, Center	`bessely(order, x)`	Julia only
Error	Amplitude, Center, Width	`erf(x)`	Julia / stdlib-js
Airy	Amplitude, Width, Center	`airy(x)`	Julia only
Legendre	Degree, Amplitude	`legendre(n, x)`	Julia only
Hermite	Degree, Amplitude	Custom polynomial	Julia only

Backend:

- JavaScript: Limited (stdlib-js for erf)
- Julia: SpecialFunctions.jl (v2.0 required for advanced functions)

2D Function Generators

Apertures & Optical Elements (v1.5+)

Function	Parameters	Implementation	Backend
Circle	Radius, Center	`r < radius ? 1 : 0`	JavaScript
Rectangle	Width, Height, Center	Rect comparison	JavaScript
Annulus	Inner/Outer Radius	`inner < r < outer ? 1 : 0`	JavaScript
Gaussian 2D	σx, σy, Center	$\exp(-(x^2/\sigma x^2 + y^2/\sigma y^2))$	JavaScript
Cassegrain	Primary/Secondary radii	Annulus + center block	JavaScript
Zernike	n, m coefficients	Zernike polynomials	Julia only (v2.0)
Airy Disk	Radius	`2*J1(x)/x`	Julia only (Bessel)

Backend:

- JavaScript: Basic apertures (v1.5)
- Julia: Advanced optical functions (Images.jl, v2.0)

Image/Data Functions

Function	Parameters	Implementation	Backend
Data 2D	Image array	Load from file	JavaScript (v1.0)
Checkerboard	Size, Frequency	Periodic pattern	JavaScript
Gaussian Noise 2D	σ, μ, seed	2D randn	JavaScript
Sinusoid 2D	fx, fy, Phase	`sin(2π(fx*x + fy*y) + φ)`	JavaScript

Import formats: PNG, JPEG → grayscale conversion (JavaScript, v1.0)

Operations

Transform Operations

Operation	Parameters	Implementation	Backend
FFT	None	fft(signal)	**fft.js (v1.0)** → rustfft (v1.5) → FFTW.jl (v2.0)
IFFT	None	ifft(spectrum)	Same as FFT
FFT2	None	fft2(image)	Julia only (FFTW.jl, v2.0)
IFFT2	None	ifft2(spectrum)	Julia only (v2.0)
DCT	None	dct(signal)	WASM → Julia
Hilbert	None	Analytic signal	Julia only (DSP.jl, v2.0)

Performance:

- JavaScript (fft.js): ~100ms for 4096 samples
- Rust WASM (rustfft): ~5ms for 4096 samples (v1.5)
- Julia (FFTW.jl): <1ms for 4096 samples (v2.0)

Convolution/Correlation

Operation	Parameters	Implementation	Backend
Convolve	Signal2	Direct convolution	JavaScript (v1.0, slow) → WASM (v1.5)
Correlate	Signal2	Cross-correlation	JavaScript → WASM
Autocorrelate	None	Self-correlation	WASM
Convolve 2D	Image2	2D convolution	Julia only (Images.jl, v2.0)

Performance:

- JavaScript: O(N^2) direct (slow for N>1000)
- WASM: FFT-based O(N log N) (v1.5)
- Julia: FFTW-optimized (v2.0)

Arithmetic & Manipulation

Operation	Parameters	Implementation	Backend
Add	Signal2, scalar	`signal1 + signal2` or `signal + scalar`	**JavaScript (v1.0)**
Subtract	Signal2, scalar	`signal1 - signal2`	JavaScript
Multiply	Signal2, scalar	`signal1 * signal2`	JavaScript
Divide	Signal2, scalar	`signal1 / signal2`	JavaScript

Normalize	Method (peak, RMS, energy)	Scale to target amplitude	JavaScript
Scale	Factor	`signal * factor`	JavaScript
Shift	Δx (time/space shift)	Index offset	JavaScript
Reverse	None	Flip array	JavaScript
Absolute	None	`abs(signal)`	JavaScript
Square	None	`signal^2`	JavaScript
Sqrt	None	`sqrt(signal)`	JavaScript
Log	Base	`log(signal)`	JavaScript
Exp	None	`exp(signal)`	JavaScript

All arithmetic operations: JavaScript (v1.0), vectorized in Julia (v2.0)

Complex Number Operations

Operation	Parameters	Implementation	Backend
Real	None	Extract real part	JavaScript (v1.0)
Imaginary	None	Extract imag part	JavaScript
Magnitude	None	`sqrt(re² + im²)`	JavaScript
Phase	None	`atan2(im, re)`	JavaScript
Conjugate	None	Flip imaginary sign	JavaScript
Complex Multiply	Signal2	`(a+bi)(c+di)`	JavaScript

Implementation:

- JavaScript: Separate real/imag arrays or complex library
- Julia: Native complex number support (v2.0)

Spatial Operations

Operation	Parameters	Implementation	Backend
Rotate 2D	Angle (degrees)	2D rotation matrix	Julia (Images.jl, v2.0)
Scale 2D	Factor	Interpolation	Julia (v2.0)
Crop 2D	x, y, w, h	Extract region	JavaScript (v1.5)
Pad 2D	Width, Height	Zero-padding	JavaScript (v1.5)
Transpose	None	Swap x/y	JavaScript (v1.5)

Flip H/V	Axis	Mirror	JavaScript (v1.5)

Calculus Operations

Operation	Parameters	Implementation	Backend
Derivative	None	Finite differences	JavaScript (v1.0)
Integral	None	Cumulative sum	JavaScript
Gradient 2D	None	Sobel/Prewitt	WASM → Julia
Laplacian 2D	None	Second derivative	Julia (v2.0)

Filtering & Processing

Operation	Parameters	Implementation	Backend
Lowpass	Cutoff, Order	Butterworth filter	WASM (DSP lib, v1.5) → Julia (DSP.jl, v2.0)
Highpass	Cutoff, Order	Butterworth filter	WASM → Julia
Bandpass	Low/High Cutoff, Order	Butterworth filter	WASM → Julia
Median	Window size	Median filter	WASM → Julia
Moving Average	Window size	Convolution	JavaScript (v1.5)

Backend:

- JavaScript: Simple FIR filters (v1.5)
- Rust WASM: dasp library (v1.5)
- Julia: DSP.jl (Butterworth, Chebyshev, elliptic filters, v2.0)

Holography Operations (v2.0)

Operation	Parameters	Implementation	Backend
Fresnel Propagate	Distance, Wavelength	Fresnel transform	Julia only
Fraunhofer Propagate	Distance, Wavelength	FFT-based	Julia only
Phase Unwrap	None	2D phase unwrapping	Julia only
Reconstruct Hologram	Reference beam	Complex division	Julia only

Requires: Julia (v2.0), Images.jl, FFTW.jl

Implementation Libraries

JavaScript (v1.0)

Math:

- Native: `Math.sin/cos/exp/log/sqrt/abs`
- math.js: General math operations
- stdlib-js: Special functions (erf, etc.)

DSP:

- **fft.js** (192k downloads/week) - FFT/IFFT
- Custom: Convolution, filters, windows

Performance: 5-10% of native, sufficient for demos

Rust WASM (v1.5)

DSP:

- **rustfft** - Optimized FFT (60-95% native speed)
- **dasp** - Filters, convolution, interpolation
- **ndarray** - Array operations

Build: wasm-pack, ~150KB bundle

Julia (v2.0 Desktop)

Core:

- **FFTW.jl** - Fastest FFT (100% native)
- **DSP.jl** - Filters, windows, spectral analysis
- **SpecialFunctions.jl** - Bessel, erf, gamma
- **Distributions.jl** - Noise generation
- **Images.jl** - 2D operations

Server: HTTP.jl (localhost:8080)

Priority Roadmap

v1.0 Essential Features (JavaScript)

1D Generators (15 functions):

- Sine, Cosine, Chirp, Sinc, Gaussian
- Rectangle, Triangle, Delta, Step
- Constant, Zero, Square Wave
- Gaussian Noise, Uniform Noise, Data

Operations (20 operations):

- FFT, IFFT
- Add, Subtract, Multiply, Divide, Normalize
- Real, Imag, Magnitude, Phase
- Derivative, Integral
- Shift, Reverse, Scale

- Absolute, Square, Sqrt, Log, Exp

2D:

- Data 2D (load images)

Performance target: <100ms FFT (4096 samples), <50ms arithmetic ops

---

v1.5 Expansion Features (+ WASM)

1D Generators (10 additions):

- Complex Sinusoid, Besinc, Lorentzian
- Poisson/Binomial/Exponential Noise
- Line, Monomial, Comb

Operations (8 additions):

- FFT-based convolution (fast)
- Lowpass/Highpass/Bandpass filters
- Moving Average, Median
- Crop, Pad, Flip

2D Generators (8 additions):

- Circle, Rectangle, Annulus, Gaussian 2D
- Cassegrain, Checkerboard
- Gaussian Noise 2D, Sinusoid 2D

Performance target: <5ms FFT (4096 samples), FFT-based convolution

---

v2.0 Complete Features (+ Julia)

1D Generators (All 80+ functions):

- All Bessel functions (J, Y, I, K)
- Airy, Legendre, Hermite
- Rayleigh/Lorentz noise
- All window functions

Operations (All 40+ operations):

- FFT2, IFFT2, DCT, Hilbert
- 2D convolution, correlation
- Butterworth/Chebyshev/Elliptic filters
- Gradient, Laplacian
- Holography operations

2D Generators (All optical elements):

- Zernike polynomials
- Airy Disk
- Advanced apertures

Performance target: <1ms FFT (4096 samples), real-time 2D processing

---

Backend Decision Tree

```
User Action → Check capability
├─ Basic waveform (sine, gauss, etc.)
│  └─ JavaScript ✓ (v1.0)
├─ FFT < 4096 samples
│  └─ JavaScript ✓ (v1.0, ~100ms)
├─ FFT > 4096 samples
│  └─ WASM ✓ (v1.5, ~5ms) OR Julia (v2.0, <1ms)
├─ Filtering (lowpass, highpass)
│  └─ WASM ✓ (v1.5) OR Julia (v2.0)
├─ Bessel functions
│  └─ Julia ONLY (v2.0)
├─ 2D FFT, holography
│  └─ Julia ONLY (v2.0)
```

Progressive enhancement: Start JavaScript → Upgrade to WASM → Optionally use Julia

---

Implementation priority: Focus on v1.0 features (35 total: 15 generators + 20 operations) for production release, defer advanced features to v1.5/v2.0.

Feature Additions & Enhancements

Purpose: Modern signal processing features to enhance SignalShow's educational and practical value beyond original Java implementation

---

Overview

Original SignalShow provides ~80 1D/2D generators and ~40 operations. These additions target modern DSP applications (audio, communications, machine learning) while maintaining educational focus.

Implementation timeline:

- v1.0: Core legacy features only
- v1.5: Selected modern additions (marked *)
- v2.0: Advanced features (marked **)

---

1. Time-Frequency Analysis

STFT (Short-Time Fourier Transform) *

Rationale: Analyze non-stationary signals (time-varying frequency content)

Use cases:

- Audio spectrograms
- Speech analysis
- Vibration monitoring

Implementation:

- JavaScript: Custom windowed FFT (v1.5)
- Julia: DSP.jl `stft()` (v2.0)

Parameters: Window size, hop size, window type (Hanning, Hamming)

Educational value: Time-frequency resolution trade-off, windowing effects

---

Wavelet Transform **

Rationale: Multi-resolution time-frequency analysis

Types:

- Continuous Wavelet Transform (CWT)
- Discrete Wavelet Transform (DWT)

Wavelets: Morlet, Mexican hat, Daubechies, Haar

Implementation: Julia only (Wavelets.jl, v2.0)

Use cases:

- Signal denoising
- Feature extraction
- Multi-scale analysis

Educational value: Scale vs frequency, mother wavelets, filter banks

---

Spectrogram Visualization *

Rationale: Visual time-frequency representation

Implementation:

- JavaScript: Plotly.js heatmap from STFT output (v1.5)
- Parameters: Color scale (dB, linear), time/freq resolution

Features:

- Interactive zoom/pan
- Colormap selection (viridis, hot, jet)
- dB scale conversion

---

2. Audio Processing & Speech

Mel-Frequency Cepstral Coefficients (MFCCs) **

Rationale: Standard feature extraction for speech/music

Implementation: Julia only (v2.0)

Steps:

1. Pre-emphasis filter
2. Windowing + FFT
3. Mel filterbank
4. Log + DCT

Use cases:

- Speech recognition
- Speaker identification
- Music genre classification

Educational value: Perceptual frequency scales, cepstral analysis

---

Pitch Detection **

Algorithms:

- Autocorrelation method
- Cepstral method
- Harmonic product spectrum

Implementation: Julia (v2.0)

Use cases:

- Music tuning
- Speech analysis
- Vocal training

Audio Effects *

Basic effects (v1.5):

- Reverb (convolution with impulse response)
- Echo/Delay (time-shifted copy)
- Tremolo/Vibrato (amplitude/frequency modulation)

Implementation: JavaScript (delay lines, modulation)

Educational value: LTI systems, modulation, impulse responses

---

3. Advanced Image Processing

Edge Detection *

Algorithms:

- Sobel operator (v1.5)
- Canny edge detector (v2.0)
- Laplacian of Gaussian (v2.0)

Implementation:

- JavaScript: Sobel (convolution kernels, v1.5)
- Julia: Images.jl (Canny, LoG, v2.0)

Educational value: Gradient operators, multi-stage processing

---

Morphological Operations **

Operations:

- Erosion, Dilation
- Opening, Closing
- Structuring elements

Implementation: Julia (Images.jl, v2.0)

Use cases:

- Noise removal
- Shape analysis
- Binary image processing

Image Filtering *

Filters:

- Gaussian blur (v1.5)

- Median filter (noise removal, v1.5)
- Bilateral filter (edge-preserving, v2.0)
- Wiener filter (optimal denoising, v2.0)

Implementation:

- JavaScript: Gaussian, Median (v1.5)
- Julia: All filters (Images.jl, v2.0)

---

4. Interactive Visualization

Animated Parameter Sweeps *

Rationale: Visualize effects of parameter changes dynamically

Example animations:

- Chirp frequency sweep
- Filter cutoff variation
- Window size effects on STFT

Implementation:

- Framer Motion (React, v1.5)
- Auto-generate frames, smooth transitions
- Playback controls (play/pause, speed)

Educational value: Intuitive understanding of parameter effects

---

3D FFT Surface Plots **

Rationale: Visualize 2D FFT results in 3D

Implementation:

- Three.js + react-three-fiber (v2.0)
- Interactive camera (orbit, pan, zoom)
- Colormap height mapping

Use cases:

- 2D Fourier analysis
- Optical transfer functions
- Frequency response surfaces

Real-Time Audio Visualization *

Features:

- Live waveform display
- Real-time spectrogram
- Frequency spectrum (FFT)

Implementation:

- Web Audio API (v1.5)
- Canvas/WebGL rendering
- Low-latency (~50ms)

Use cases:

- Music visualization
- Acoustic analysis
- Teaching demonstrations

---

5. Machine Learning Integration

Signal Classification **

Rationale: Demonstrate ML on signal data

Examples:

- Audio event detection (clap, speech, music)
- ECG heartbeat classification
- Gesture recognition (accelerometer)

Implementation:

- TensorFlow.js (v2.0)
- Pre-trained models (import)
- Simple custom models (train in browser)

Educational value: Feature extraction → ML pipeline

Noise Reduction (Denoising) **

Algorithms:

- Wiener filtering
- Wavelet thresholding
- ML-based (denoising autoencoders)

Implementation: Julia (v2.0)

Educational value: Signal vs noise separation, trade-offs

---

6. Modern Communication Systems

Digital Modulation *

Schemes:

- Amplitude Shift Keying (ASK)
- Frequency Shift Keying (FSK)
- Phase Shift Keying (PSK, QPSK)
- Quadrature Amplitude Modulation (QAM)

Implementation:

- JavaScript: ASK, FSK, PSK (v1.5)
- Julia: All schemes including QAM (v2.0)

Features:

- Modulate bit streams
- Add channel noise
- Demodulate + BER calculation

Educational value: Digital communications, constellation diagrams, BER vs SNR

Channel Coding **

Codes:

- Hamming codes
- Reed-Solomon
- Convolutional codes

Implementation: Julia only (v2.0)

Educational value: Error correction, redundancy, coding gain

Matched Filtering *

Rationale: Optimal detection in noise

Implementation: JavaScript (cross-correlation, v1.5)

Use cases:

- Pulse detection
- Radar/sonar
- Communications

Educational value: SNR maximization, correlation receivers

---

7. Numerical Methods & Optimization

Curve Fitting **

Methods:

- Polynomial regression
- Least-squares fitting
- Non-linear fitting (Levenberg-Marquardt)

Implementation: Julia (LsqFit.jl, Optim.jl, v2.0)

Use cases:

- Model parameter estimation
- Data interpolation
- Trend analysis

Numerical Integration/Differentiation *

Enhancements to existing:

- Adaptive integration (Julia, v2.0)
- Higher-order differentiation schemes (v1.5)
- 2D integration (v2.0)

Educational value: Numerical stability, accuracy vs complexity

---

Root Finding **

Algorithms:

- Newton-Raphson
- Bisection
- Secant method

Implementation: Julia (Roots.jl, v2.0)

Use cases:

- Solve f(x) = 0
- Optimization
- Parameter estimation

---

8. Educational Enhancements

Interactive Tutorials *

Format: Step-by-step guided explorations

Examples:

- "Understanding FFT" (DFT → FFT, visualization)
- "Nyquist Theorem" (sampling, aliasing demo)
- "Filtering Basics" (lowpass, highpass, frequency response)
- "Convolution Explained" (visual convolution animation)

Implementation:

- React components (v1.5)
- Embedded code snippets
- Interactive parameter sliders
- Progress tracking

Educational value: Self-paced learning, hands-on experimentation

---

Example Signal Library *

Categories:

- Audio: Speech samples, music clips, bird calls
- Biomedical: ECG, EEG, EMG signals

- Communications: BPSK, QPSK modulated signals
- Natural: Earthquakes, weather data
- Synthetic: Test patterns, noise samples

Implementation:

- Bundled `.sig1d` files (v1.0)
- Cloud storage for large files (v1.5)
- User-contributed library (v2.0)

Educational value: Real-world data exposure, reproducible examples

---

Assessment Mode **

Features:

- Quiz questions embedded in tutorials
- "Predict the output" challenges
- Signal identification games
- Parameter estimation exercises

Implementation: React + state management (v2.0)

Use cases:

- Classroom assignments
- Self-assessment
- Gamified learning

LaTeX Math Rendering *

Rationale: Display equations properly in documentation/help

Implementation:

- KaTeX or MathJax (v1.5)
- Inline and block equations
- Markdown support

Examples:

```
FFT: X[k] = \sum_{n=0}^{N-1} x[n] e^{-i2\pi kn/N}
Convolution: (f * g)(t) = \int f(\tau)g(t-\tau)d\tau
```

Educational value: Professional mathematical notation

Priority Summary

v1.0 (Launch) - Core Legacy Features

- Focus: Replicate Java SignalShow functionality
- No new feature additions
- Goal: Working browser-based DSP tool

v1.5 (Expansion) - Selected Modern Features

Time-frequency:

- STFT
- Spectrogram visualization

Audio:

- Audio effects (reverb, echo)
- Real-time visualization

Image:

- Edge detection (Sobel)
- Gaussian/Median filtering

Communications:

- Digital modulation (ASK, FSK, PSK)
- Matched filtering

Education:

- Interactive tutorials
- LaTeX rendering
- Animated parameter sweeps

Estimated: +15 new features, ~30% more functionality

---

v2.0 (Advanced) - Full Modern DSP Suite

All v1.5 features plus:

- Wavelet analysis
- MFCCs, pitch detection
- Advanced image processing (Canny, morphology)
- 3D visualizations
- ML integration (classification, denoising)
- Communication coding
- Numerical optimization
- Assessment mode

Estimated: +25 new features, ~80% more functionality than Java original

---

Implementation Notes

Library choices:

- Audio: Web Audio API (JavaScript), DSP.jl (Julia)
- Image: Plotly.js (JavaScript), Images.jl (Julia)
- ML: TensorFlow.js (v2.0 only)
- Math: math.js, stdlib-js (JavaScript), SpecialFunctions.jl (Julia)

Performance targets:

- STFT (4096 samples, 512 window): <200ms (JavaScript), <10ms (Julia)
- Spectrogram plot: <100ms render
- Real-time audio: <50ms latency

Educational design principles:

1. **Progressive disclosure**: Start simple, reveal complexity gradually
2. **Immediate feedback**: Changes update visualizations instantly
3. **Exploration encouraged**: Safe to experiment, undo/reset available
4. **Multiple representations**: Show same concept in different ways (time, frequency, equations)
5. **Real-world context**: Use practical examples and applications

---

Key decision: Prioritize educational value over feature count. Each addition must clearly demonstrate a DSP concept or have strong practical application.

JavaScript DSP Libraries Research

Purpose: Evaluate JavaScript DSP libraries for v1.0 web deployment

Executive Summary

JavaScript libraries are **10-100x slower** than Rust WASM but provide:

- 100% browser compatibility (no WASM required)
- Small bundle sizes (10-150KB total)
- Zero-installation fallback

Strategy: Use JavaScript for v1.0, add WASM in v1.5 for performance

Recommended Libraries

1. fft.js (FFT/IFFT)

Stats:

- Version: 4.0.4
- License: MIT
- Stars: 315
- Downloads: ~30k/week
- Bundle: ~3KB gzipped

Features:

- Radix-4 FFT (optimized for power-of-4 sizes)
- Real FFT optimization (~25% faster for real signals)
- Inverse FFT
- **Limitation**: Requires power-of-2 sizes only

Performance:

- 4096 samples: ~~15,676 ops/sec (~~64µs per transform)
- 8192 samples: ~~6,896 ops/sec (~~145µs)
- **2-3x faster** than alternative JS FFT libraries

Usage:

```
const FFT = require("fft.js");
const fft = new FFT(2048); // Power of 2 required
const output = fft.createComplexArray();
fft.realTransform(output, realInput); // 25% faster for real signals
```

Recommendation: ✅ Use for v1.0

2. dsp.js (Filters, Windows, Oscillators)

Stats:

- License: MIT
- Stars: 1.7k
- Bundle: ~50KB
- ⚠️ **Unmaintained since 2015**

Features:

- FIR/IIR filters (lowpass, highpass, bandpass)
- Window functions (Hamming, Hanning, Blackman)
- Oscillators (sine, square, sawtooth, triangle)
- Envelope generators (ADSR)
- Convolution

Limitations:

- No security updates since 2015
- Slower than modern implementations
- Limited documentation

Usage:

```
const lowpass = new IIRFilter(DSP.LOWPASS, 1000, 44100, 2);
lowpass.process(buffer);
```

Recommendation: ⚠️ Use carefully, consider rewriting filters

---

3. math.js (General Math)

Stats:

- Version: 12.x
- License: Apache-2.0
- Stars: 14k
- Downloads: ~2M/week
- Bundle: ~150KB minified

Features:

- Complex numbers (native support)
- Matrices and linear algebra
- Statistics functions
- Expression parser
- Unit conversions

Performance:

- Slower than specialized libraries
- Large bundle size

Usage:

```
const math = require('mathjs');
const complex = math.complex(2, 3); // 2 + 3i
const matrix = math.matrix([[1, 2], [3, 4]]);
```

Recommendation: ✅ Use for complex arithmetic, avoid for performance-critical ops

---

4. stdlib-js (Special Functions)

Stats:

- License: Apache-2.0
- Modular: ~1-5KB per function
- Active maintenance

Features:

- Special functions (erf, gamma, bessel)
- Statistical distributions
- Random number generators
- BLAS-level operations

Performance: Moderate (pure JavaScript)

Usage:

```
const erf = require('@stdlib/math-base-special-erf');
const result = erf(1.0); // 0.8427...
```

Recommendation: ✅ Use for special functions in v1.0

Performance Comparison

Operation	JavaScript	Rust WASM	Julia	Notes
FFT (4096)	~64µs	~5µs	<1µs	fft.js vs rustfft vs FFTW
Sine generation (4096)	~500µs	~50µs	<10µs	Math.sin() vs native
Convolution (1000)	~50ms	~1ms	<1ms	Direct O(N^2) vs FFT-based
Filter (4096)	~2ms	~200µs	<50µs	IIR/FIR filtering

Conclusion: JavaScript is 10-100x slower, but sufficient for teaching demos (4096 samples in <100ms total)

Bundle Size Analysis

Minimal v1.0 bundle:

- fft.js: ~3KB gzipped
- Custom filters: ~5KB

- Custom generators: ~10KB
- math.js (complex): ~150KB
- **Total**: ~170KB gzipped

Optimization:

- Tree-shake math.js (use only complex module): ~20KB
- **Optimized total**: ~40KB gzipped

Browser Compatibility

All libraries support:

- Chrome/Edge 90+
- Firefox 88+
- Safari 14+
- Mobile browsers (iOS 14+, Android 10+)

No WASM required - pure JavaScript

Implementation Strategy

v1.0 (JavaScript)

Use:

- fft.js for FFT/IFFT
- Custom implementations for filters (simple FIR/IIR)
- math.js for complex arithmetic
- stdlib-js for special functions (erf, etc.)

Avoid:

- dsp.js (unmaintained)
- Heavy libraries (tone.js, wavesurfer.js) - overkill for DSP core

Performance target: <100ms for typical operations (4096 samples)

---

v1.5 (+ Rust WASM)

Upgrade:

- rustfft for FFT (~10x faster)
- dasp for filters (~50x faster)
- Keep JavaScript as fallback

Progressive enhancement:

```
const backend = window.WebAssembly
  ? new WasmBackend()
  : new JavaScriptBackend();
```

Missing Features

Not available in JavaScript (require Julia/WASM):

- Bessel functions (J, Y, I, K) - use Julia SpecialFunctions.jl
- Wavelet transforms - use Julia Wavelets.jl
- Advanced filters (elliptic, Chebyshev) - use Julia DSP.jl or Rust dasp
- 2D FFT optimization - use Julia FFTW.jl

Workarounds:

- Implement basic Bessel J0/Y0 approximations (limited accuracy)
- Use FFT-based convolution for filtering
- Defer advanced features to v1.5/v2.0

Code Examples

FFT Example

```
// Generate 4096-sample sine wave
const sampleRate = 8000;
const freq = 440; // A4 note
const duration = 0.512; // 4096 samples
const signal = new Float32Array(4096);

for (let i = 0; i < signal.length; i++) {
  signal[i] = Math.sin(2 * Math.PI * freq * i / sampleRate);
}

// Compute FFT
const FFT = require('fft.js');
const fft = new FFT(4096);
const out = fft.createComplexArray();
fft.realTransform(out, signal);
fft.completeSpectrum(out);

// Extract magnitude spectrum
const magnitude = new Float32Array(2048);
for (let i = 0; i < 2048; i++) {
  const real = out[i * 2];
  const imag = out[i * 2 + 1];
  magnitude[i] = Math.sqrt(real * real + imag * imag);
}

// Peak should be at bin 440 / (8000/4096) ≈ 225
```

Custom Filter Example

```
// Simple lowpass FIR filter (moving average)
function movingAverage(signal, windowSize) {
  const filtered = new Float32Array(signal.length);

  for (let i = 0; i < signal.length; i++) {
    let sum = 0;
    let count = 0;

    for (let j = Math.max(0, i - windowSize + 1); j <= i; j++) {
      sum += signal[j];
      count++;
    }

    filtered[i] = sum / count;
  }

  return filtered;
}
```

Recommendations Summary

1. **Use fft.js** for FFT/IFFT in v1.0 (fastest, lightweight)
2. **Avoid dsp.js** (unmaintained) - implement filters manually
3. **Use math.js** selectively (complex numbers only, tree-shake)
4. **Use stdlib-js** for special functions (erf, gamma)
5. **Target <100ms** for operations on 4096 samples (acceptable for teaching)
6. **Plan WASM upgrade** in v1.5 for 10-100x speedup
7. **Fallback gracefully** - detect WASM support, use JS if unavailable

---

Bottom line: JavaScript is sufficient for v1.0 teaching demos. Performance limitations (~100ms FFT) are acceptable for interactive learning. Upgrade to WASM in v1.5 for production use.

Rust DSP Crates Research

Research Date: November 2025
Purpose: Evaluate Rust DSP crates for SignalShow web deployment (WebAssembly)

Executive Summary

The combination of **rustfft**, **dasp**, and **ndarray** provides 85-90% of SignalShow's DSP functionality with performance within 10-20% of native implementations and bundle sizes under 150KB gzipped. All three crates support `no_std` and compile to WebAssembly with SIMD acceleration.

1. rustfft - Fast Fourier Transform

Version: 6.4.1
License: MIT/Apache 2.0
Repository: https://github.com/ejmahler/RustFFT

Algorithm Support

Supports radix-2, radix-4, mixed-radix (2^n × 3^m), Bluestein's algorithm for prime sizes, and optimized real FFT. Performance is within 5-10% of FFTW for power-of-two sizes, 10-15% for mixed-radix, and 20% for large primes.

WebAssembly Support

Enable WASM SIMD for 2-3x performance improvement over non-SIMD. Supported in Chrome 91+, Firefox 89+, Safari 16.4+. Bundle size: ~50-80KB gzipped.

```
[dependencies]
rustfft = { version = "6", features = ["wasm_simd"], default-features = false }
```

Important: rustfft does not normalize outputs. Scale by `1/n` after forward FFT or by `1/sqrt(n)` for both forward and inverse transforms.

2. dasp - Digital Audio Signal Processing

Version: 0.11.0
License: MIT/Apache 2.0
Repository: https://github.com/RustAudio/dasp

Capabilities

Modular suite providing signal generation (sine, saw, square, triangle, noise), windowing functions (Hann, rectangle), sample rate conversion (linear and sinc interpolation), envelope detection (peak and RMS), and basic signal operations (add, scale, multiply, clip). Missing Hamming, Blackman, and Kaiser windows require custom implementation.

WebAssembly Support

Fully `no_std` compatible with no platform dependencies. Bundle size: 30-50KB gzipped for selective features, 100-150KB for full suite. Use selective features to minimize bundle size:

```toml
[dependencies]
dasp = {
    version = "0.11",
    default-features = false,
    features = ["signal", "signal-window", "interpolate", "window-hanning"]
}
```

3. ndarray - N-Dimensional Arrays

Version: 0.16.1
License: MIT/Apache 2.0
Repository: https://github.com/rust-ndarray/ndarray

Capabilities

NumPy-like array operations including slicing, broadcasting, element-wise operations, and matrix multiplication. Supports 1D and 2D arrays with reshaping and iteration. Pure Rust matrix multiply is suitable for small to medium matrices without BLAS.

WebAssembly Support

Fully `no_std` compatible. BLAS feature is not compatible with WASM (requires C libraries); use pure Rust operations instead. Bundle size: 30-40KB gzipped.

```toml
[dependencies]
ndarray = { version = "0.16", default-features = false }
# Do NOT enable "blas" feature for WASM
```

Coverage

Provides complete support for 1D/2D arrays, slicing, element-wise operations, matrix multiplication, and statistical operations. Convolution requires custom implementation using FFT-based methods.

Bundle Size and Performance

Production Bundle (Gzipped)

Component	Size
rustfft	60KB
dasp	40KB
ndarray	35KB
Total	150KB

Compared to Julia runtime (500MB + 1-2GB packages) or Pyodide (20-30MB), Rust WASM is 100-3000x smaller.

Performance Benchmarks

FFT performance (1024 samples) with WASM SIMD is 60% of native speed, 10-20x faster than pure JavaScript. Array operations on 1M elements show similar improvements (5-10x faster than JavaScript).

Capability Gaps

Requires Custom Implementation or JavaScript Fallback

- Special functions (Bessel, Airy, elliptic integrals, though erf/gamma available in `libm`)
- Advanced wavelets (CWT with exotic bases, wavelet packet decomposition)
- Exotic filters (elliptic/Cauer, inverse Chebyshev)

Fully Supported in Rust WASM

All FFT operations, basic filters (Butterworth, Chebyshev I, FIR), convolution/correlation, spectrograms, basic wavelets (Haar, Daubechies), signal generators, resampling/interpolation, and 2D operations.

Recommended Configuration

```toml
[dependencies]
rustfft = { version = "6.4", features = ["wasm_simd"], default-features = false }
dasp = {
    version = "0.11",
    default-features = false,
    features = [
        "signal",
        "signal-window",
        "interpolate",
        "interpolate-linear",
        "interpolate-sinc",
        "window-hanning",
        "envelope-peak",
        "rms",
    ]
}
ndarray = { version = "0.16", default-features = false }
num-complex = { version = "0.4", default-features = false }
wasm-bindgen = "0.2"

[profile.release]
opt-level = "s"      # Optimize for size
lto = true           # Link-time optimization
codegen-units = 1    # Better optimization
strip = true         # Strip debug symbols
```

Implementation Strategy

Phase 1: Core DSP (Rust WASM)

FFT/IFFT, basic signal generators (sine, saw, square), windowing (Hann, custom Hamming/Blackman), array operations, convolution.

Phase 2: Advanced Operations (Rust WASM)

Filters (Butterworth, Chebyshev), resampling, spectrogram, 2D FFT, correlation.

Phase 3: Specialized Features (Desktop Only - Julia)

Special functions (Bessel, Airy), advanced wavelets (CWT with exotic bases), exotic filters (elliptic, inverse Chebyshev), very large datasets (>10M samples).

Conclusion

Rust's DSP ecosystem provides production-ready support for 85-90% of SignalShow's web features with performance 60-95% of native and bundle sizes under 150KB gzipped. Missing functionality (10-15% of capabilities) can be handled through graceful degradation or desktop-only features using Julia.

Animation & 3D Graphics Strategy

Date: December 2024

Executive Summary

Multi-pronged approach combining web animations (Framer Motion + D3.js) for real-time interactivity, Python Manim for publication-quality video export, and Three.js + react-three-fiber for WebGL-accelerated 3D visualizations.

Animation Strategy

Manim Integration

No production-ready JavaScript port of Manim exists. Evaluated experimental packages (`vivid-animations` , `react-manim` , `mathlikeanim-rs` , `manichrome`) are all too experimental or abandoned. Instead, use established web animation libraries for interactivity and real Python Manim for video export.

Real-Time Interactivity (Web): Framer Motion (23k stars) for React animations with spring physics, D3.js (108k stars) for custom mathematical visualizations, optional GSAP (19k stars) for complex timeline animations. Provides immediate feedback with zero latency.

Video Production (Python Manim): Export SignalShow configurations as JSON, auto-generate Manim code, render high-quality video (1080p/4K, 60fps) for YouTube, papers, and educational content. Desktop app can bundle Python + Manim for one-click export.

3D Graphics Strategy

Three.js + react-three-fiber

Three.js (103k stars) is the industry standard for WebGL. React-three-fiber (29.7k stars) provides declarative Three.js rendering using React components with zero overhead. Essential packages: `@react-three/fiber` (core renderer), `@react-three/drei` (helpers like OrbitControls), `@react-three/postprocessing` (visual effects), `leva` (GUI controls).

Enhanced 3D Visualizations

The Java version had limited 3D capabilities due to Swing constraints. WebGL enables GPU-accelerated rendering at 60fps+ with modern shader-based effects.

High-Priority Features:

1. **2D FFT as 3D Surface**: Interactive height-mapped surfaces with rotation/zoom instead of static 2D heatmaps. Students can rotate to understand frequency structure spatially.

2. **Holographic Diffraction Patterns**: Volumetric rendering of Cassegrain apertures, multi-arm interferometry, and Fresnel zones. Few educational tools provide this.

3. **Filter Frequency Response**: Combined 3D surface showing magnitude (height) and phase (color) simultaneously instead of separate 2D plots.

4. **Complex Signal Space**: 3D I/Q trajectories for visualizing modulation schemes (QPSK, QAM) and constellation diagrams.
5. **Signal Correlation Volumes**: 3D correlation volumes for 2D signals, useful for stereo imaging and SAR processing.

Performance Optimization

For datasets over 10k vertices: use `BufferGeometry`, transfer data as binary Float32Array via WebSocket, implement GPU-based colormapping with custom shaders, and apply LOD (Level of Detail) for distant surfaces.

Alternative Libraries Considered

3D Graphics: Babylon.js (too heavy), Plotly.js 3D (limited interactivity, no custom shaders, suitable only for simple plots), raw WebGL (too low-level), A-Frame (VR-focused, not React-friendly), PixiJS (2D only). Three.js + react-three-fiber offers the best balance of maturity, React integration, and community support.

Animation: React Spring (steeper learning curve alternative to Framer), GSAP (powerful timelines, commercial license for some features), Anime.js (lightweight but less React-friendly), Remotion (React-based video generation, newer alternative to Manim). Framer Motion + D3.js + Python Manim provides optimal coverage for simple UI animations, complex visualizations, and publication-quality exports.

Julia Backend Integration

Binary data transfer via WebSocket is 5-10x faster than JSON for large numerical arrays. Julia backend computes DSP operations (2D FFT, holographic patterns, filter responses) as Float64 matrices, converts to binary format, and sends via WebSocket. TypeScript frontend receives as ArrayBuffer, converts to Float32Array, generates Three.js geometry, applies GPU shader-based colormapping, and renders at 60fps with WebGL.

Educational Impact

Visual learning benefits 65% of students. 3D spatial understanding improves intuition for abstract concepts. Interactive engagement increases retention over passive lectures. Aligns with Grant Sanderson's (3Blue1Brown) pedagogy: visual and intuitive mathematics, animations revealing process rather than just results, incremental complexity building, strategic use of color and motion. Sanderson used Pluto.jl notebooks for MIT courses, which integrates naturally with our Julia stack for exploration and Manim for publication.

Target use cases: undergraduate DSP and Fourier analysis courses, graduate advanced DSP and holography, self-directed learning via YouTube and MOOCs, high school physics enrichment.

Implementation Roadmap

v1.0 (Months 1-4): Framer Motion + D3.js interactive demos, 2D FFT surfaces with OrbitControls.

v1.5 (Months 5-7): JSON animation export, manual Manim code generation, filter responses and complex signal spaces.

v2.0 (Months 8-10): Auto-generate Manim Python from JSON, holographic patterns, custom shaders.

v2.5 (Months 11-12): Desktop app with bundled Manim for one-click video export, VR support via @react-three/xr, advanced physics simulations.

Key Dependencies

```json
{
  "dependencies": {
    "three": "^0.180.0",
    "@react-three/fiber": "^8.18.0",
    "@react-three/drei": "^9.118.0",
    "framer-motion": "^11.15.0",
    "d3": "^7.9.0"
  }
}
```

Conclusion

This strategy positions SignalShow with WebGL-accelerated 3D visualizations impossible in the Java version, real-time interaction via modern web animation libraries, and publication-quality video export through authentic Manim integration. Provides both interactive exploration and professional content creation within a single educational platform.

SignalShow Port to Nuthatch Desktop Platform

Date: October 25, 2025
Status: Working prototype deployed

About Nuthatch Desktop

Nuthatch Desktop is a web-based desktop environment built with React, Vite, and Tauri. It provides a modular application platform where apps are distributed as self-contained `.app` bundles containing React components, dependencies, and assets. The system includes:

- **Modular app architecture**: Apps are discovered and loaded dynamically from `system-rom/*.app/` directories
- **Window management**: Full desktop windowing with minimize, maximize, snap-to-side, z-ordering, and multi-instance support
- **File system integration**: OPFS (Origin Private File System) for web, native file operations in Tauri
- **Technology stack**: React 19.2, Vite 7.1, Tailwind CSS, optional Julia and WebGPU capabilities
- **Developer experience**: Hot module reloading, standardized APIs, theme system

The platform already includes demonstration apps showcasing Julia computation (Mandelbrot sets, FFT), WebGPU acceleration (compute shaders, particle systems), and Monaco code editor integration.

Executive Summary

SignalShow has been successfully ported to Nuthatch Desktop as a working prototype. The modular app system provides ideal infrastructure for educational signal processing with access to Julia DSP libraries and WebGPU acceleration. Current prototype demonstrates basic signal generation and visualization; full feature parity with Java version requires additional development.

Current Status: Phase 1 prototype functional **Remaining Work**: Phases 2-3 for full Java feature parity and enhanced capabilities

Prototype Status

Current Implementation

SignalShow.app is deployed in `nuthatch-desktop/system-rom/` with basic functionality:

Implemented Features:

- Signal generation (Gaussian, sinusoid, chirp, square wave, delta, rectangular pulse, exponential, ramp, noise, sawtooth)
- 1D plotting with Plotly.js (zoom, pan, interactive controls)
- Basic operations (FFT, filtering, convolution)
- Nuthatch Desktop integration (window management, theming, file associations)
- Julia backend bridge for computationally intensive operations

Architecture:

```
system-rom/SignalShow.app/
├── app.json              # Manifest
├── app.jsx              # Main React component
├── components/          # UI components
├── lib/                 # Signal processing
├── assets/              # Icons, presets
└── dependencies/        # Plotly.js, DSP libraries
```

Platform Integration Points

Window Management: Multi-instance support allows side-by-side comparison of time/frequency domain, original/filtered signals, or 1D/2D visualizations.

Julia Backend: Leverages existing Julia.app infrastructure. SignalShow can offload heavy DSP computations (large FFTs, 2D operations) to Julia HTTP server for 10-100x performance improvement over pure JavaScript.

WebGPU: Can utilize WebGPU.app patterns for GPU-accelerated 2D FFT, real-time spectrograms, and 3D surface rendering.

File System: Integration with Files.app for signal import/export, preset management, and demo library.

Development Roadmap

Phase 1: Prototype (Completed)

Basic signal generation, 1D visualization, fundamental operations, and Nuthatch Desktop integration. Demonstrates feasibility and provides foundation for expansion.

Phase 2: Feature Parity (8-12 weeks)

Advanced Operations: Complete FFT/IFFT implementation with magnitude/phase plots, comprehensive filtering (low-pass, high-pass, band-pass, Butterworth, Chebyshev), convolution, correlation, and window functions.

2D Signals and Images: 2D function generators (Gaussian, sinc, diffraction patterns), 2D FFT using WebGPU compute shaders, image operations (edge detection, blur, sharpening, morphological operations).

Educational Demos: Port Java version demonstrations including sampling theorem, aliasing visualization, Fourier series builder, filter design tool, holography simulation, Doppler effect, and convolution animator.

Deliverable: Full-featured SignalShow matching Java version capabilities with 50+ generators, complete operation library, 2D processing, and demo collection.

Phase 3: Enhanced Features (4-6 weeks)

Julia Backend: Dedicated HTTP server for heavy computations using DSP.jl, FFTW.jl, and Images.jl. Provides 10-100x performance improvement for large datasets and enables real-time processing.

WebGPU Acceleration: GPU-accelerated FFT using compute shaders, real-time spectrograms at 60 FPS, 3D FFT surface rendering, particle-based visualizations.

Advanced UI: Operation chain visualization showing signal processing pipeline, multi-window workflows with automatic side-by-side comparison, file associations for `.sig` / `.signalshow` / `.wav` formats, batch processing capabilities.

Deliverable: Enhanced SignalShow surpassing Java version with GPU acceleration, modern web features, and seamless Nuthatch Desktop integration.

Technical Challenges and Solutions

Performance: Java version uses JAI for optimized image operations. Phase 1 prototype uses pure JavaScript (acceptable for educational datasets). Phase 2 implements WebGPU compute shaders (matches/exceeds JAI performance). Phase 3 adds Julia backend (surpasses Java).

2D Image Operations: Canvas API handles basic operations. WebGPU provides advanced capabilities demonstrated in WebGPU.app. Web image processing libraries (jimp, sharp via WASM) available as fallback.

Real-time Interactivity: React hooks (useState, useEffect) manage state. RequestAnimationFrame enables smooth animations. Debouncing optimizes expensive computations. Optimistic UI updates maintain responsiveness.

Mathematical Notation: KaTeX renders equations. SVG generates custom diagrams. Canvas handles complex visualizations.

Platform Advantages

Nuthatch Desktop provides approximately 70% of required infrastructure:

Provided by Platform:

- Window management (minimize, maximize, snap, z-order, multi-instance)
- File system integration via Files.app and OPFS
- Theme system with light/dark mode
- Julia computation infrastructure (Julia.app HTTP server pattern)
- WebGPU examples and patterns (WebGPU.app compute shaders)
- Build system and hot module reloading
- Discovery via Start Menu
- Component sharing across apps

SignalShow Contributions:

- DSP library (reusable by other apps)
- Scientific plotting utilities
- Educational demo framework
- Signal processing components

Cross-App Integration:

- Julia.app: Export signals for custom analysis, import computation results
- Files.app: Signal file management, preset library
- Monaco Editor.app: Custom function scripting, batch operations

Development Status

Phase 1 (Completed): Working prototype with signal generation, 1D visualization, basic operations, and platform integration.

Phase 2 (8-12 weeks remaining): Advanced DSP operations, 2D image processing, educational demos, full Java feature parity.

Phase 3 (4-6 weeks): Julia backend integration, WebGPU acceleration, advanced UI features, enhanced capabilities beyond Java version.

Total Remaining: 12-18 weeks for production-ready implementation with enhanced features.

Conclusion

SignalShow prototype successfully demonstrates Nuthatch Desktop's suitability for educational signal processing applications. The modular app architecture, existing Julia and WebGPU infrastructure, and comprehensive platform services significantly reduce development effort compared to standalone web deployment. Completing Phases 2-3 will deliver a production-ready application surpassing the original Java implementation's capabilities through modern web technologies and GPU acceleration.

Desktop Backend & Implementation Reference

Status: Early-stage proposals and historical reference documentation

---

Part V: Desktop Version Backend Proposal (Optional)

Note: Early-stage concept for Nuthatch Desktop/Tauri deployment. May be deferred or abandoned in favor of web-only deployment.

Julia Server Lifecycle Management

Proposed Behavior: Julia backend server persists across app sessions with automatic shutdown after 15 minutes of inactivity.

Startup Process:

1. Check Julia installation in PATH and common locations (`~/.juliaup/bin/julia` , `/usr/local/bin/julia` , `/opt/homebrew/bin/julia` , `C:\Users\%USERNAME%\.juliaup\bin\julia.exe`)
2. Ping `http://localhost:8080/health` to detect running server
3. If not running, spawn `julia server.jl --port 8080`
4. Verify startup with health check after 3 seconds

Runtime: Health check pings every 60 seconds reset inactivity timer. Server tracks `last_activity` timestamp on all HTTP requests.

Shutdown: Automatic after 15 minutes of inactivity (configurable via `INACTIVITY_TIMEOUT` constant). Server persists when app closes but stops after timeout expires. Manual shutdown via Ctrl+C.

Auto-Start Implementation

Proposed Architecture: Tauri backend (`src-tauri/src/julia_server.rs`) provides commands for Julia process management. React frontend (`SignalShow.app/app.jsx`) implements auto-start logic on component mount.

Tauri Commands:

- `check_julia_server` : HTTP health check to `http://localhost:8080/health`
- `start_julia_server` : Spawn Julia process with platform-independent handling
- `stop_julia_server` : Terminate server process
- `get_julia_path` : Locate Julia executable in PATH

Dependencies: `reqwest` (HTTP client), `tokio` (async runtime)

User Experience: App displays server status (checking, running, Julia not installed, failed to start). Automatic startup on launch, cleanup on unmount.

Part VI: Implementation Reference

Julia Installation

Recommended Method: juliaup (official version manager)

macOS/Linux:

```
curl -fsSL https://install.julialang.org | sh
source ~/.bashrc  # or ~/.zshrc
julia --version
```

Windows:

```
winget install julia -s msstore
# or download from julialang.org
```

System Requirements: 64-bit OS (macOS 10.9+, Windows 7+, Linux), 2GB RAM minimum (8GB recommended), 500MB disk space for Julia + 1-2GB for packages.

Package Installation (for backend):

```
using Pkg
Pkg.add(["HTTP", "JSON", "DSP", "FFTW", "Images"])
```

Verification:

```
julia --version
julia -e 'using HTTP, DSP, FFTW; println("Packages loaded")'
```

Quick Start

Manual Server Testing:

```
# Start server
julia signalshow-backend/server.jl --port 8080

# Test health endpoint
curl http://127.0.0.1:8080/health

# Stop server
pkill -f "julia.*server.jl"
```

Development Mode (Tauri):

```
cd nuthatch-desktop/src-tauri
cargo build

cd ..
npm run tauri dev
```

Expected Behavior: SignalShow app automatically checks Julia installation, detects/starts server, displays "Server running" status.

BUGS

This file documents known bugs and issues in the SignalShow-Java repository. Use the entries below to collect bug reports, with reproducible steps, impact, and any temporary workarounds.

ID format: BUG-### (e.g. BUG-001)

Template

- ID: BUG-XXX
- Title: Short descriptive title
- Status: open / in-progress / resolved / wontfix
- Reported by:
- Date: YYYY-MM-DD
- Environment: OS, Java version, branch, steps to reproduce
- Description: Detailed description
- Steps to reproduce: 1. 2. 3.
- Expected behavior:
- Actual behavior:
- Logs / stacktrace:
- Temporary workaround:
- Notes / PR:

Example

- ID: BUG-001
- Title: Script fails when class files missing
- Status: resolved
- Reported by: julietfiss
- Date: 2025-10-09
- Environment: macOS, openjdk 11, branch main
- Description: Running `./run-signalshow.sh` failed with "Error: Could not find or load main class SignalShow" when class files were not compiled.
- Steps to reproduce:
  1. Clone repository
  2. Remove `SignalShow/SignalShow.class` if present
  3. Run `./run-signalshow.sh`
- Expected behavior: Script should compile sources and run the application.
- Actual behavior: Java threw ClassNotFoundException because `.class` not present.
- Logs / stacktrace: Error: Could not find or load main class SignalShow Caused by: java.lang.ClassNotFoundException: SignalShow
- Temporary workaround: Manually compile with `javac SignalShow.java signals/**/*.java` before running.
- Notes / PR: Updated `run-signalshow.sh` to automatically compile missing classes and committed as "Add automatic compilation to run-signalshow.sh script" (commit b996cd6).

Maven Build System and Distribution

Overview

SignalShow has been successfully migrated from legacy shell-script-based compilation (circa 2005-2009) to a modern Maven build system with native installer distribution. The migrated codebase is hosted on GitHub at:

Repository: `https://github.com/Nuthatch-Inc/SignalShow-Java`
Branch: `feature/maven-migration`

Build System Architecture

Maven Configuration

Group ID: `org.signalshow`
Artifact ID: `signalshow`
Version: `1.0.0-SNAPSHOT`
Java Target: Java 25 LTS

The `pom.xml` configures:

- **Maven Shade Plugin**: Creates executable JAR with bundled dependencies
- **Maven Exec Plugin**: Runs application via `mvn exec:java`
- **JPackage Plugin**: Generates native installers for macOS and Windows

Directory Structure

```
SignalShow-Java/
├── pom.xml                      # Maven build configuration
├── src/
│   ├── main/
│   │   ├── java/                # Source code (395 files)
│   │   │   ├── SignalShow.java  # Main class
│   │   │   └── signals/         # Application packages
│   │   └── resources/           # Application resources (339 files)
│   │       ├── images/
│   │       ├── guiIcons/
│   │       ├── demoIcons/
│   │       ├── plotIcons/
│   │       ├── functiondoc/
│   │       └── operationdoc/
│   └── test/java/               # Test sources (future)
├── legacy-build/                # Archived original structure
│   ├── jai_core.jar           # JAI dependency (installed to local Maven repo)
│   └── jai_codec.jar          # JAI dependency (installed to local Maven repo)
├── assets/icons/              # Application icons
│   ├── SignalShowIcon.png     # Source icon (48x48)
│   ├── SignalShow.icns        # macOS icon
│   └── SignalShow.ico          # Windows icon
├── compile.sh                  # Helper: compile only
├── run-maven.sh                # Helper: compile and run
├── package.sh                  # Helper: build JAR
├── run.sh                      # Helper: run from JAR
├── create-mac-icon.sh          # Generate .icns from PNG
├── create-windows-icon.sh      # Generate .ico from PNG
├── build-installer-mac.sh      # Build macOS .dmg
├── build-installer-windows.sh  # Build Windows .exe/.msi
└── .github/workflows/
    └── build-installers.yml    # Automated builds on version tags
```

Dependencies

Java Advanced Imaging (JAI)

SignalShow depends on JAI libraries (`jai_core.jar` , `jai_codec.jar`) for image processing operations. Since these are not available in Maven Central, they are:

1. Stored in `legacy-build/` directory
2. Installed to local Maven repository during build setup
3. Referenced in `pom.xml` with coordinates:
    - `javax.media.jai:jai-core:1.1.3`
    - `javax.media.jai:jai-codec:1.1.3`

Installation command:

```
mvn install:install-file -Dfile=legacy-build/jai_core.jar \
  -DgroupId=javax.media.jai -DartifactId=jai-core \
  -Dversion=1.1.3 -Dpackaging=jar
```

Build Commands

Development

Compile: `mvn clean compile` or `./compile.sh`
Run: `mvn exec:java` or `./run-maven.sh`
Package: `mvn clean package` or `./package.sh`
Run JAR: `java -jar target/signalshow-1.0.0-SNAPSHOT.jar` or `./run.sh`

Distribution

macOS Installer: `./build-installer-mac.sh` → `target/dist/SignalShow-1.0.0.dmg`
Windows Installer: `./build-installer-windows.sh` → `target/dist/SignalShow-1.0.0.exe`

Both installers:

- Bundle Java Runtime Environment (JRE 17+)
- Include application icon
- Create desktop shortcuts

- Add to system application menu
- Allow directory chooser (Windows)

Automated Distribution

GitHub Actions workflow (`.github/workflows/build-installers.yml`) automatically builds both macOS and Windows installers when a version tag is pushed:

```
git tag -a v1.0.0 -m "Release version 1.0.0"
git push origin v1.0.0
```

The workflow:

1. Sets up Java 25 and Maven on both `macos-latest` and `windows-latest` runners
2. Installs JAI dependencies to local Maven repository
3. Generates platform-specific icons (`.icns` for macOS, `.ico` for Windows)
4. Builds native installers using `jpackage`
5. Creates GitHub Release with installers attached as downloadable artifacts

Artifacts are retained for 90 days and attached to the corresponding GitHub Release.

Icon System

SignalShow uses a custom icon derived from the original `guiIcons/SignalShowIcon.png` (48×48 PNG):

- **macOS**: `SignalShow.icns` (979KB, contains sizes: 16×16 to 512×512 @1x and @2x)
- **Windows**: `SignalShow.ico` (70KB, contains sizes: 16×16, 32×32, 48×48, 256×256)

Icons are generated using:

- **macOS**: `sips` and `iconutil` (native tools)
- **Windows**: ImageMagick 7 (`magick convert` command)

Build scripts automatically include icons in generated installers.

Migration Artifacts

The Maven migration preserved all functionality while modernizing the build system:

Source Files Migrated: 395 Java files
Resource Files Migrated: 339 files (images, icons, documentation)
Compilation Status: BUILD SUCCESS (all files compile cleanly)
Runtime Status: Application launches and functions correctly
Distribution Status: Native installers build successfully on macOS and Windows

The original directory structure is preserved in `legacy-build/` for reference.

Key Benefits

1. **Dependency Management**: Maven handles JAI libraries and future dependencies
2. **IDE Integration**: Full support in IntelliJ IDEA, Eclipse, VS Code
3. **Automated Builds**: GitHub Actions CI/CD pipeline

4. **Professional Distribution**: Native installers with bundled JRE
5. **Version Control**: Clean Git workflow with feature branches
6. **Standardization**: Industry-standard Maven conventions

Future Enhancements

Potential improvements enabled by Maven migration:

- **Unit Testing**: Add JUnit tests in `src/test/java/`
- **Code Coverage**: Integrate JaCoCo for coverage reporting
- **Dependency Updates**: Migrate to modern image processing libraries (e.g., ImageJ, OpenCV)
- **Multi-Module Structure**: Separate core, GUI, and plugins
- **Continuous Deployment**: Automatic releases to GitHub Releases or Maven Central