

SUN Magnetics

JoSIM®-Pro v2.7.0

User Manual

Copyright © 2026 SUN Magnetics (Pty) Ltd. All Rights Reserved for:
JoSIM®-Pro v2.7.0.

SUN Magnetics (RF) (Pty) Ltd
15 De Beer Street
Stellenbosch, 7600
Republic of South Africa

www.sun-magnetics.com

Permission is granted to anyone to make or distribute verbatim copies of this document as received, in any medium, provided that the copyright notice and the permission notice are preserved, and that the distributor grants the recipient permission for further redistribution as permitted by this notice.

JoSIM® is a registered trademark of SUN Magnetics (RF) (Pty) Ltd.
All other trademarks are the property of their respective owners.

Date: September 29, 2026

Credits

JoSIM®-Pro is the product of the combined research and development efforts of the JoSIM Development Team, including contributors from SUN Magnetics (Pty) Ltd, with additional support from industry partners and researchers.

Contents

1 Introduction **8**
About This Manual . . . 8
Who JoSIM®-Pro Is For . . . 8
What This Manual Covers . . . 8

2 System Requirements **9**
Operating Systems and Packages . . . 9
Hardware Requirements . . . 9
Minimum Requirements . . . 9
Recommended Requirements . . . 9
Software Requirements . . . 9
Installed Command and Launcher Expectations . . . 9
Network and Licensing . . . 10
Practical Guidance . . . 10
Unsupported or Non-Standard Systems . . . 10

3 Getting Started: Install, License, and Launch **11**
Choose the package for your platform . . . 11
Windows install and launch . . . 11
Linux install and launch . . . 11
macOS install and launch . . . 11
Licensing and first-run behavior . . . 12
Generate a System ID from the GUI . . . 12
Install or select the issued license . . . 13
Optional command-line validation . . . 13
Continue to the first GUI simulation . . . 13

4 GUI Quick Start: First Simulation **14**
Before starting . . . 14
Launch the GUI . . . 14
Open the installed examples directory . . . 15
Open the JTL example . . . 15
Check run settings . . . 16
Run the active netlist . . . 16
Inspect the waveform . . . 17
Export or save the result . . . 17
Continue from here . . . 17

5 GUI Workspace, Editor, and Visualizer **18**
Workspace overview . . . 18
Toolbar and context-sensitive actions . . . 18
Creating, opening, and saving files . . . 18
File browser, recent places, and pinned roots . . . 19
Drag-and-drop opening . . . 19
Session restore and persistent workspace state . . . 19
External file changes . . . 20
Editor basics . . . 20
Switching to the netlist visualizer . . . 21
Visualizer controls and layout edits . . . 21
Subcircuit navigation . . . 22

Layout persistence . . . 22
Schematic export . . . 22

6 GUI Simulation and Results 23
Simulation lifecycle . . . 23
Preconditions for running . . . 23
Run settings . . . 24
Create multiple runs . . . 24
Result states and diagnostics . . . 25
Waveform result view . . . 25
Trace Manager basics . . . 26
Visibility, order, groups, and comparison plots . . . 26
Analysis outputs . . . 26
Manual annotations . . . 27
Imported CSV and TSV result data . . . 27
Plot export studio . . . 28
Troubleshooting hooks . . . 28

7 Optimum CLI Workflows: Verify, Margin, Yield, and Optimize 29
GUI handoff model . . . 29
Installed DFF example . . . 29
CLI invocation and licensing . . . 29
Mode overview . . . 30
Runtime contract . . . 30
File paths and project layout . . . 31
Minimal configurations . . . 31
Configuration reference . . . 32
Verification inputs and tolerances . . . 34
Margin workflow . . . 37
Yield workflow . . . 38
Optimize workflow . . . 38
Worked DFF example . . . 39

8 GUI-to-CLI Equivalence 40
Simulation workflow equivalence . . . 40
Run Settings equivalence . . . 40
Output, streaming, and import equivalence . . . 40
Licensing equivalence . . . 41
Optimum handoff equivalence . . . 41

9 CLI Usage and Command Reference 43
9.1 When to use the CLI . . . 43
9.2 Command structure . . . 43
9.3 Default behavior . . . 43
9.4 Quick command patterns . . . 44
9.5 Option summary . . . 44
9.5.1 Input, licensing, and information . . . 44
9.5.2 Analysis and runtime control . . . 44
9.5.3 Output and reporting . . . 45
9.6 Operational notes . . . 45
9.6.1 Analysis and integration overrides . . . 45
9.6.2 Variable-step operation . . . 45
9.6.3 Output targets and file formats . . . 45
9.6.4 Netlist `.FILE` routing . . . 46
9.6.5 Streaming output . . . 46
9.6.6 Compression . . . 46
9.6.7 Reproducibility and randomness . . . 46
9.6.8 Licensing commands . . . 46
9.7 Common tasks . . . 46
9.7.1 Run a normal fixed-step simulation . . . 46
9.7.2 Run a voltage-domain simulation . . . 46

9.7.3 Run a variable-step `PHASE + BDF` simulation . . . 46
9.7.4 Stream a long run directly to disk . . . 46
9.7.5 Read from *STDIN* and write to *STDOUT* . . . 47
9.7.6 Generate a CSV file for GUI import . . . 47
9.8 Summary . . . 47

10 Netlist Syntax Reference **48**
Netlist anatomy . . . 48
Quick syntax index . . . 48
Elements and models . . . 48
Sources and controlled sources . . . 48
Control commands . . . 49
Detailed reference . . . 49
Element reference . . . 49
Resistor . . . 49
Inductor . . . 49
Capacitor . . . 49
Josephson junction . . . 49
Transmission Line . . . 50
Mutual Inductance . . . 53
Independent source reference . . . 53
Voltage Source . . . 53
Current Source . . . 54
Phase Source . . . 54
Waveform and source functions . . . 54
Dependent source reference . . . 56
Current Controlled Current Source . . . 56
Current Controlled Voltage Source . . . 56
Voltage Controlled Current Source . . . 56
Voltage Controlled Voltage Source . . . 56
Control-command reference . . . 56
CLI Analysis Jobs and Control Blocks . . . 56
GUI and API Analysis Jobs . . . 57
Transient Analysis . . . 58
Parameter Definition . . . 58
Subcircuits . . . 58
Include Files . . . 58
Noise Settings . . . 58
Parameter Spread . . . 59
Output Control . . . 59
IV Curve Generation . . . 60
File Output . . . 60
Runtime and .OPTION Controls . . . 61
Built-in constants . . . 62
Summary . . . 62

11 Worked Netlist Examples **63**
First transient run and result inspection . . . 63
Hierarchical design with subcircuits and included files . . . 64
Device and model characterization with .IV . . . 64
Noise, spread, and reproducibility with a fixed seed . . . 65
Variable-step and streamed output for larger runs . . . 65
Derived outputs and multiple output files . . . 65
Full logic-cell examples . . . 66
Josephson Transmission Line (JTL) . . . 66
RSFQ Splitter Cell . . . 67
RSFQ AND Gate . . . 68
Summary . . . 70

12 Simulation Feature Reference **71**

Core Simulation Capabilities . . . 71
Choosing Between Phase and Voltage Workflows . . . 71
Fixed-Step and Variable-Step Operation . . . 71
Circuit Description and Modelling . . . 71
SPICE-like Netlists, Parameters, and Hierarchy . . . 71
Sources and Circuit Elements . . . 72
Josephson Junction Models . . . 72
Noise, Spread, and Realism . . . 72
Thermal Noise . . . 72
Explicit Noise Waveforms and Junction Noise . . . 72
Parameter Spread and Reproducibility . . . 72
Characterization and Analysis Workflows . . . 73
Transient and Startup Workflows . . . 73
IV Curve Generation . . . 73
Variation, Noise, and Repeatability Studies . . . 73
Derived Outputs and Diagnostics . . . 73
Output, Post-Processing, and Data Handling . . . 73
Flexible Output Formats . . . 73
Streaming Output and Multiple Files . . . 73
Hierarchical Referencing and Expressions . . . 74
Summary . . . 74

13 Python Automation **75**
Making the module available . . . 75
Process-wide settings and licensing . . . 75
Smallest useful file-based run . . . 75
Runtime control from Python . . . 76
Explicit runtime example . . . 76
Reading results into NumPy . . . 77
Getting named traces from the print list . . . 77
Writing CSV, BIN, RAW, or normal print output . . . 77
Large runs, streaming, and variable-step restrictions . . . 77
Repeated runs and batch automation . . . 78
End-to-end plotting example . . . 78
Practical reminders . . . 79

14 Troubleshooting **80**

A Native Integration Appendix **86**
A.1 What the native interfaces are for . . . 86
A.2 When to use this instead of Python . . . 86
A.3 Minimal file-based workflow . . . 86
A.4 Path and encoding notes . . . 86
A.5 Where the authoritative reference lives . . . 86

Introduction

JoSIM®-Pro is a professional circuit simulation environment for superconducting electronics and Josephson-junction-based designs. It is intended for engineers, researchers, and technical users who need a reliable way to model circuit behaviour, study transient response, evaluate design choices, and analyse simulation results across a range of workflows.

At its core, JoSIM®-Pro is designed to make superconducting circuit simulation practical for real design work. This manual begins with graphical workflows for installation, first simulation, waveform inspection, and export, then covers command-line simulation, structured netlist support, multiple output formats, automation interfaces, and optimisation tools as complementary workflows.

JoSIM®-Pro is suitable for work ranging from small test circuits to larger and more complex superconducting systems. Whether the task is validating a basic concept, exploring timing and phase behaviour, generating output for post-processing, or preparing data for optimisation and yield studies, JoSIM®-Pro provides the tools needed to move from circuit description to useful simulation results efficiently.

JoSIM® is a registered trademark of SUN Magnetics (RF) (Pty) Ltd and is used throughout this manual to identify the simulator and associated software tools.

About This Manual

This manual is a practical guide to using JoSIM®-Pro. It is written to help users understand the software, build effective simulation workflows, and make full use of the features available in the installed package.

The emphasis throughout is on what JoSIM®-Pro does, how to use it, and how its different components fit together in normal engineering work. Where a feature has important operating limits or usage requirements, those are presented as practical operating guidance.

Who JoSIM®-Pro Is For

JoSIM®-Pro is intended for:

- engineers and researchers working with superconducting circuits and Josephson junction devices
- users who prefer a graphical environment for editing, running, and reviewing simulations
- users who need a simulator that can be driven from the command line as part of repeatable design flows
- users who want to automate simulation and post-processing from Python
- users who need design-space exploration, verification, margin, yield, or optimisation workflows through the Optimum CLI

While the manual assumes a technical audience, it is organised so that new users can begin with installation and GUI-based first-use workflows, and then move on to netlist controls, scripting, CLI automation, Python, native integration, and optimisation as needed.

What This Manual Covers

This manual covers the principal user workflows available in JoSIM®-Pro, including:

- installation, licensing, and first-time setup
- GUI-based editing, execution, and visualisation workflows
- command-line simulation with `josim-pro-cli`
- circuit netlists, control statements, and supported simulation inputs
- output files, streaming, and result handling
- Optimum CLI workflows for verification, margin, yield, and optimisation
- Python-based automation and scripting
- troubleshooting and reference material for common usage issues

The chapters that follow are arranged so that the manual can be read from front to back as an introduction to the product, or used selectively as a reference while working.

Key strengths of JoSIM®-Pro include:

- **High Performance**: JoSIM®-Pro is built to handle demanding superconducting circuit simulation workloads efficiently.
- **Cross-Platform Availability**: The software is designed for use on Windows, Linux, and macOS.
- **Flexible Workflows**: Users can work through the GUI first, then move to the command line, Python, or Optimum CLI when automation or deeper control is needed.
- **Practical Output Handling**: JoSIM®-Pro supports workflows that range from direct console output to file-based export and downstream analysis.

With JoSIM®-Pro, users have access to a capable and flexible environment for superconducting circuit simulation, from first experiments through to detailed design and analysis tasks.

System Requirements

This chapter summarizes the operating-system, hardware, software, and licensing requirements for installed use of JoSIM®-Pro. The GUI supports interactive editing, simulation, and result inspection; the CLI, Python module, Optimum CLI, and native APIs support automation, scripted operation, and integration workflows.

Operating Systems and Packages

Use the package built for your operating system and architecture.

- **Windows**: 64-bit Windows 10 or later. The installer is an MSI that installs under `Program Files/SUN Magnetics/JoSIM-Pro` and can create `JoSIM-Pro` shortcuts for launching the GUI.
- **Linux**: a recent 64-bit Linux environment compatible with the supplied DEB or RPM package. The active enterprise target is EL8/RHEL8-compatible. GUI packages require the desktop GTK/OpenGL runtime libraries declared by the package metadata.
- **macOS**: a current macOS version matching the architecture of the supplied macOS package. Current package builds are architecture-specific unless the accompanying notes explicitly state that a universal binary is supplied.

Use the installer or package supplied for your operating system rather than copying files from another platform or build layout.

Hardware Requirements

Minimum Requirements

The following configuration is suitable for installation, learning, and smaller simulation jobs:

- **Processor (CPU)**: dual-core 64-bit processor
- **Memory (RAM)**: 4 GB
- **Storage**: at least 500 MB for software plus additional space for projects, examples, output files, and temporary files
- **Display**: desktop display suitable for the GUI workspace, editor, and waveform views

Recommended Requirements

For larger circuits, longer transient runs, Monte Carlo style studies, or heavier post-processing, use:

- **Processor (CPU)**: modern quad-core or better 64-bit processor
- **Memory (RAM)**: 16 GB or more
- **Storage**: SSD storage with at least 1 GB available for installation, outputs, examples, and temporary files
- **Display**: an engineering desktop environment with enough space for netlists, logs, visualizer views, and plots

Software Requirements

- **Windows runtime**: required runtime components are supplied or expected by the MSI package. If a site image is missing Microsoft runtime components, repair the installation or install the required redistributable supplied by your administrator.
- **Linux GUI runtime**: install DEB/RPM packages with `apt install`, `dnf install`, or `yum install` so dependencies such as GTK and OpenGL runtime libraries are resolved automatically.
- **macOS package runtime**: use the supplied package layout. The application bundle contains the GUI and app-local runtime files; wrappers in `/usr/local/bin` expose the GUI and CLI tools.
- **Python (optional)**: Python is required only if you plan to use the JoSIM®-Pro Python interface or user-local Python integration helpers.
- **External plotting tools (optional)**: spreadsheet tools, Python plotting libraries, GNUPlot, MATLAB, or similar tools may be used for post-processing exported data. The GUI can inspect waveform results directly.

Installed Command and Launcher Expectations

- **Windows**: launch the GUI with the `JoSIM-Pro` shortcut; it targets `JoSIM_Pro_GUI.exe`. Command-line discovery depends on the MSI command-line PATH integration option.
- **Linux**: launch the GUI through `/usr/local/bin/josim-pro-gui` or the desktop entry named `JoSIM-Pro`. Use the wrapper rather than starting the GUI binary directly.

- **macOS**: launch the GUI from `/Applications/SUN Magnetics/JoSIM-Pro.app` or through `/usr/local/bin/josim-pro-gui` when the macOS package is installed.

The command-line simulator is `josim-pro-cli`. The optimization tool is `optimum-cli`. These tools are fully supported for automation, batch simulation, reproducibility checks, and TOML-driven optimisation workflows.

Network and Licensing

A valid JoSIM®-Pro license is required before simulations can run. Current packages also require the latest SUN Magnetics License Manager to be installed, because JoSIM-Pro delegates license checks and System ID generation to the `smlicensed` licensing daemon. Without `smlicensed`, neither the GUI nor the command-line tools can generate `SM_SysID.txt`.

- **License Manager prerequisite**: install or update the SUN Magnetics License Manager before launching JoSIM-Pro for licensing, System ID generation, or simulation work.
- **Default daemon locations**: JoSIM-Pro looks for `C:\ProgramFiles\SUNMagnetics\Licenses\bin\smlicensed.exe` on Windows, `/usr/local/share/sunmagnetics/licenses/bin/smlicensed` on Linux, and `/Applications/SUNMagnetics/Licenses/bin/smlicensed` on macOS. Site deployments can override this with `SMLICENSED_PATH` when required.
- **Offline use**: once installed and licensed correctly, normal simulation work can be performed without a permanent network connection unless your licensing arrangement requires online validation.
- **License fulfilment**: network access may be required during deployment or license administration, depending on how the software is issued and managed in your environment.
- **Licensing service**: if the GUI reports that it cannot contact the licensing service, use **Retry** only after correcting the License Manager installation, service state, network, firewall, license-file, or seat-availability problem.
- **Support**: for licensing or installation assistance, contact support@sun-magnetics.com.

Practical Guidance

For the best experience with JoSIM®-Pro:

- keep project files in a user-writeable working directory
- use the installed examples directory for manual workflows
- ensure that output locations have sufficient free space
- use SSD storage for larger projects or repeated simulation runs
- use additional memory where large netlists or extensive output data are expected

Unsupported or Non-Standard Systems

If your system falls below the minimum requirements, reliable operation cannot be guaranteed. If you need to deploy JoSIM®-Pro in an unusual environment, contact SUN Magnetics for guidance before relying on the software in production work.

Getting Started: Install, License, and Launch

This chapter covers first use after installation: install JoSIM®-Pro, launch the GUI, resolve licensing, and then continue to the GUI quick start. Terminal commands remain available for validation, automation, and scripted simulation.

1. Choose the package for your platform

Use the installer or package supplied for your operating system. Do not infer support for one operating system from files intended for another platform.

- **Windows**: use the signed `JoSIM-Pro-<version>-Windows-AMD64.msi` package.
- **Linux**: use the supplied distribution package. Install it with a dependency-resolving package manager so GTK/OpenGL GUI runtime dependencies are installed with the application.
- **macOS**: use the supplied macOS package when one is included for your platform. The macOS package installs an application bundle and command wrappers; it is not interchangeable with the Windows MSI or Linux package.

2. Windows install and launch

The Windows package is a custom MSI. It installs JoSIM®-Pro under the SUN Magnetics program directory and can create shortcuts that launch the GUI.

1. Run the `JoSIM-Pro-<version>-Windows-AMD64.msi` installer.
2. Keep **Create Start Menu shortcut** selected if you want a launcher under **SUN Magnetics** in the Start Menu.
3. Keep **Create Desktop shortcut** selected if you want a Desktop launcher.
4. Keep **Command-line PATH integration** selected if you want new terminals to discover `josim-pro-cli` and `optimum-cli` automatically.
5. After installation, launch `JoSIM-Pro` from the Start Menu or Desktop shortcut.

The shortcut target is the installed GUI executable `JoSIM_Pro_GUI.exe`. If the command-line PATH option was disabled, or if a terminal was already open before installation, run command-line tools from the installed `bin` directory or open a new terminal after installation.

3. Linux install and launch

The Linux package installs runtime files below the SUN Magnetics application prefix and exposes launch commands through `/usr/local/bin`. Launch the GUI through the wrapper so the runtime library path is prepared correctly.

For a Debian-style package, prefer a dependency-resolving install:

```
sudo cp JoSIM-Pro-*.deb /tmp/
sudo apt install /tmp/JoSIM-Pro-*.deb
```

For an RPM-style package, use `dnf` or `yum` rather than `rpm -i`:

```
sudo dnf install ./JoSIM-Pro-*.rpm
# or, on systems using yum:
sudo yum install ./JoSIM-Pro-*.rpm
```

After installation, launch from the desktop entry named `JoSIM-Pro`, or run:

```
josim-pro-gui
```

The public Linux command entries are:

- `/usr/local/bin/josim-pro-gui`
- `/usr/local/bin/josim-pro-cli`
- `/usr/local/bin/optimum-cli`

The GUI wrapper prepares the runtime library search path before starting the application. If the wrapper reports that `JoSIM_Pro_GUI` is missing or not executable, repair or reinstall the package rather than bypassing the wrapper.

4. macOS install and launch

When a macOS package is supplied, it installs JoSIM®-Pro as an application bundle:

```
/Applications/SUN Magnetics/JoSIM-Pro.app
```

Launch the GUI from the application bundle or run the wrapper:

```
josim-pro-gui
```

The macOS package also exposes `josim-pro-cli` and `optimum-cli` through `/usr/local/bin`. Current macOS packages are architecture-specific. Use the package built for your Mac architecture; do not assume a universal binary unless the package notes explicitly say one is supplied.

5. Licensing and first-run behavior

A valid JoSIM®-Pro license is required before simulations can run. The latest SUN Magnetics License Manager must also be installed, because the GUI and CLI use the `smlicensed` daemon for license checks and System ID generation. On startup, the GUI checks the configured license path and contacts `smlicensed`. A successful check opens the main workspace normally.

License renewal continues in the background while simulation matrices remain open, including matrices retained by Python or C API applications. Repeated simulations can reuse a verified authorization for up to 60 seconds. If renewal fails, a simulation already running is allowed to finish; subsequent simulations require a valid authorization. Applications that retain matrices between runs remain active for licensing purposes until those matrices are released.



Normal startup license check from the installed GUI.

If licensing is unavailable, the startup shell presents recovery choices:

- **Retry**: check again after correcting the license file, network, or licensing-service problem.
- **Continue degraded**: open the GUI for limited inspection and recovery tasks. Simulations still require a valid license.
- **Quit**: close the application.



Degraded-mode startup recovery choices when licensing is unavailable.

Use degraded mode when you need to inspect installation state before support can issue or repair a license. System ID generation still requires the License Manager: if `smlicensed` is missing, outdated, blocked, or installed outside the expected location, no `SM_SysID.txt` file will be produced.

6. Generate a System ID from the GUI

If you do not yet have a license, generate a system identifier from the GUI after installing the latest SUN Magnetics License Manager:

1. Install or update the SUN Magnetics License Manager so that `smlicensed` is available on the machine.
2. Launch `JoSIM-Pro`.
3. If the startup shell reports a license problem, choose **Continue degraded**.
4. Open `Help→About`.
5. Click **Generate System ID...**. The GUI asks `smlicensed` to create `SM_SysID.txt`; it does not generate the file independently.
6. Save the file as `SM_SysID.txt`.
7. Send `SM_SysID.txt` to support@sun-magnetics.com when requested for licensing support.

If this step fails or the selected folder does not contain `SM_SysID.txt`, repair the License Manager installation first. On non-standard deployments, confirm that `SMLICENSED_PATH` points to the installed `smlicensed` executable.

The About dialog also reports the current license state. When a license is verified, it shows the licensed use type, status, licensee, and validity date. In degraded mode it reports that verified license information is unavailable.



*The About dialog exposes current license state and the **Generate System ID...** action.*

JoSIM-Pro requires SUN Magnetics License Daemon 1.5.2 or newer. It launches the daemon with the selected license before requesting authorization; an existing daemon enrolls the supplied license.

7. Install or select the issued license

After SUN Magnetics issues a license, place the license file where your site or installation instructions expect it. If the GUI does not find it automatically, set the license path once with the command-line tool:

```
josim-pro-cli -l /path/to/sm_license.txt
```

The GUI reads the configured license path during startup. After setting the path, start `JoSIM-Pro` again and confirm that the startup shell reaches the main workspace without a licensing warning.

8. Optional command-line validation

Command-line validation is useful for administrators, existing CLI users, and automation workflows. It is not required for the first GUI simulation.

Check the installed simulator version:

```
josim-pro-cli -v
```

Generate a System ID from a terminal instead of the GUI. This command also requires the latest SUN Magnetics License Manager, because it invokes `smlicensed`:

```
josim-pro-cli -s
```

Run a direct CLI simulation only after licensing is working. From the installed examples directory, use the packaged JTL example:

```
josim-pro-cli -o JTL_Example.csv JTL_Example.cir
```

The full command reference appears in the CLI chapter. `optimum-cli` is covered separately in the Optimum chapter.

9. Continue to the first GUI simulation

After the GUI launches and licensing is resolved, continue to the **GUI Quick Start** chapter. That chapter opens an installed example netlist, runs it in the GUI, and inspects the waveform result without requiring a terminal.

If the first launch fails, use the troubleshooting chapter in this order: GUI launcher problems, licensing or degraded mode, then file/netlist/simulation issues.

GUI Quick Start: First Simulation

This chapter gives a first complete JoSIM®-Pro workflow: launch the graphical interface, open the installed `JTL_Example.cir` netlist, run it, inspect the waveform, and export a result from the GUI. The command-line tools remain available for users who automate runs from a terminal.

Before starting

This tutorial assumes that JoSIM®-Pro is installed and that the latest SUN Magnetics License Manager is installed and reachable. The GUI can be opened in degraded mode for inspection and recovery, but simulation and System ID generation still depend on the installed `smlicensed` daemon and a valid licensing setup.

Use an installed example from the examples directory. That directory includes these first-use example files:

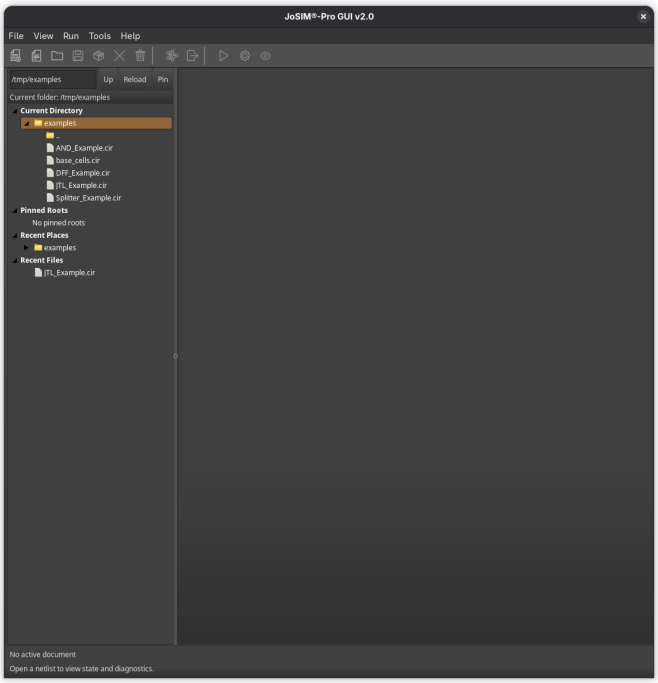
Installed file	Use in this manual
`JTL_Example.cir`	Recommended first run. It is self-contained and has explicit plot requests.
`AND_Example.cir`	Larger logic-cell example; it depends on `base_cells.cir`.
`Splitter_Example.cir`	Additional small logic-cell example for later exploration.
`DFF_Example.cir`, `DFF_Example.sp`, and `DFF_Example_config.toml`	Canonical installed Optimum handoff example; use it after the GUI first run.

1. Launch the GUI

Start the application using the installed launcher:

- On Windows, use the `JoSIM-Pro` Start Menu entry or Desktop shortcut when that installer option was selected. The launcher target is the installed `JoSIM_Pro_GUI.exe` binary.
- On Linux, use the desktop entry named `JoSIM-Pro` or run `josim-pro-gui`. The command is a wrapper that prepares the GUI runtime library search path; use it instead of starting the GUI binary directly.
- On macOS, use the application package or `josim-pro-gui` wrapper supplied for the target architecture.

If the startup license check reports a missing or invalid license, return to the install and license chapter before attempting this tutorial. `Continue degraded` is suitable for opening files and inspecting the interface, but it is not the normal simulation path.

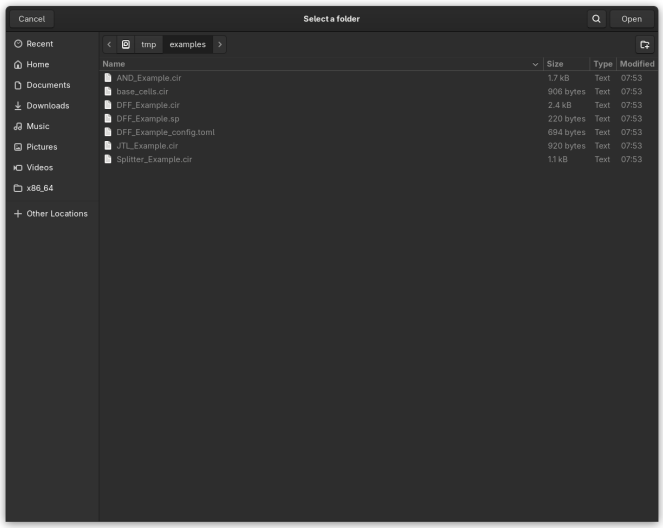


GUI workspace after opening the installed examples folder.

2. Open the installed examples directory

Use `File→Open Folder...` to point the file browser at the examples directory installed by the package. The exact parent directory is platform-dependent, but the examples are installed under an `examples` directory.

After the folder opens, the left file browser can be used to select netlists. If you already know the exact file location, use `File→Open...` instead. The open dialog accepts JoSIM netlists and result data files; for this tutorial, select `JTL_Example.cir`.



Installed examples visible from the GUI file-browser workflow.

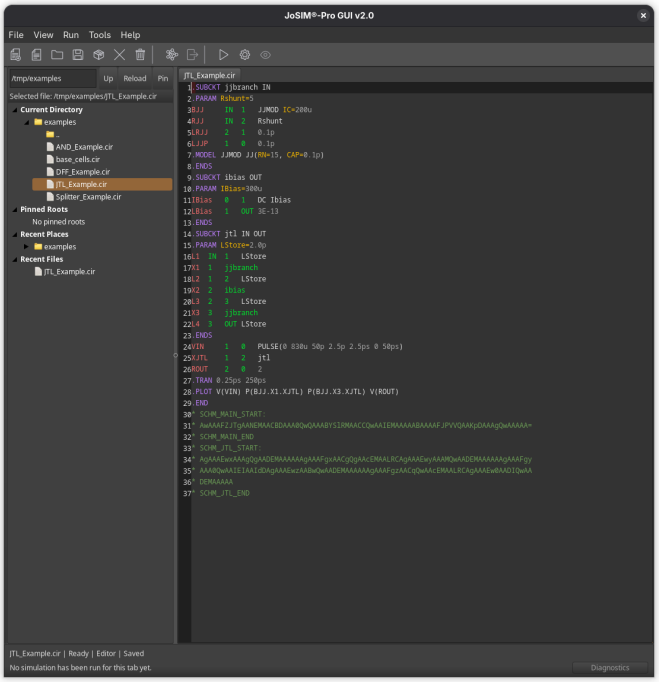
3. Open the JTL example

Open `JTL_Example.cir`. A new workspace tab appears, and the netlist is loaded into the editor. Before running it, inspect the file rather than treating it as a black-box demo.

The important lines for the first run are:

- `VIN ... PULSE(...)` defines the input pulse source.
- `XJTL 1 2 jtl` instantiates the JTL subcircuit.
- `.TRAN 0.25ps 250ps` defines the transient run length and time step.
- `.PLOT V(VIN) P(BJJ.X1.XJTL) P(BJJ.X3.XJTL) V(ROUT)` defines the traces that should appear after a successful run.

This example is a good first test because it does not require `base_cells.cir` or any other include file.



The installed `JTL_Example.cir` netlist opened in the GUI editor.

CLI side note. The equivalent terminal run is `josim-pro-cli -o JTL_Example.csv JTL_Example.cir`. This is useful for automation, but it is not required for the GUI quick start.

Editing the netlist

Click to place the caret, drag to select text, or hold Shift while using the arrow, Home, End, Page Up or Page Down keys. Double-click selects a word. Left and Right collapse an existing selection to its beginning or end; Up and Down retain the preferred column across short lines.

Use Ctrl on Windows/Linux and Command on macOS for A (select all), C (copy), X (cut), V (paste), Z (undo), Shift+Z (redo) and F (find). Ctrl+Y also performs redo. For word movement or deletion, hold Ctrl on Windows/Linux or Option on macOS with Left, Right, Backspace or Delete. Command+Left/Right moves to the line boundary on macOS; Command+Up/Down moves to the document boundary.

The `Edit` menu and editor right-click menu provide undo, redo, clipboard and search commands. Ctrl/Command+Shift+F opens the replacement controls directly. Right-click in the file browser for open, parent folder, refresh, pin/unpin and copy-full-path actions.

The find panel offers case-sensitive and whole-word matching. Expand its replacement controls to replace the current match or every match. F3 moves to the next match; Shift+F3 moves to the previous match. A selected search match can also be replaced by typing.

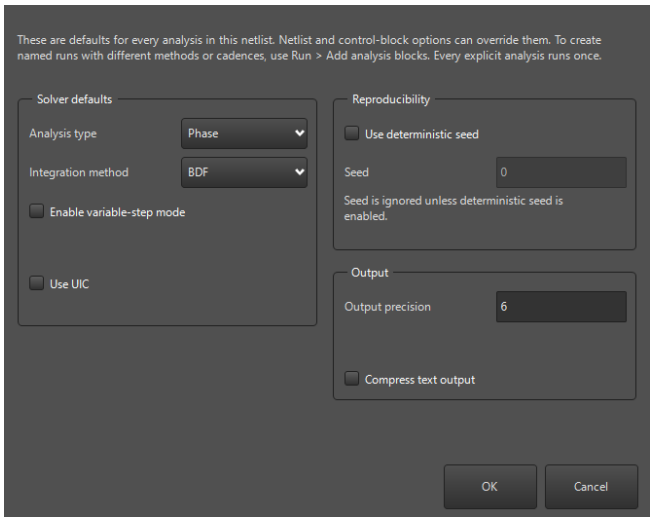
Tab indents selected lines and Shift+Tab removes one leading tab or up to four spaces. With no selection, Tab inserts a tab. Enter carries the current line's indentation. Alt+Up/Down moves the current line or selected lines. Paste, overwrite, line moves, replace-all and insertion of analysis blocks are each reversible with one undo; undo also restores the affected selection and caret.

4. Optional: check run settings

For the first run, use the default GUI settings unless you are deliberately testing a specific solver configuration. To inspect them, open `Run→Settings`. These are deck defaults; netlist and control-block settings may override them. The dialog exposes:

- analysis type: `Phase` or `Voltage`;
- integration method: `BDF` or `TRAP`;
- variable-step mode, valid only for compatible phase/BDF configurations;
- `Use UIC`;
- output precision in the supported range `0` to `17`;
- compressed text output;
- deterministic seed control for reproducible runtime behavior.

Select `Cancel` if you only inspected the settings. Select `OK` only when you intentionally changed a setting.



Default GUI run settings for the first simulation.

5. Run the active netlist

With `JTL_Example.cir` as the active tab, press the Run toolbar button. Every explicit analysis runs once; multiple analyses execute serially. To create named runs with different integration methods and output cadences, use `Run→Add analysis blocks...`, described in Create multiple runs. The GUI prepares the input file, starts the simulation session, shows progress in the active tab, and updates the result view when the run completes.

If a run is already active for the tab, the GUI prevents a second concurrent run for that tab. If the active tab contains imported CSV/TSV result data instead of a netlist, simulation is unavailable; open the source `.cir` file instead.

A failed run usually points to one of these causes:

- licensing is unavailable or degraded mode was used for inspection only;
- the netlist path or included file path is not available;
- a netlist edit introduced a syntax or model error;
- run settings were changed to an incompatible combination.

Use the diagnostics/status information in the tab before changing unrelated settings.

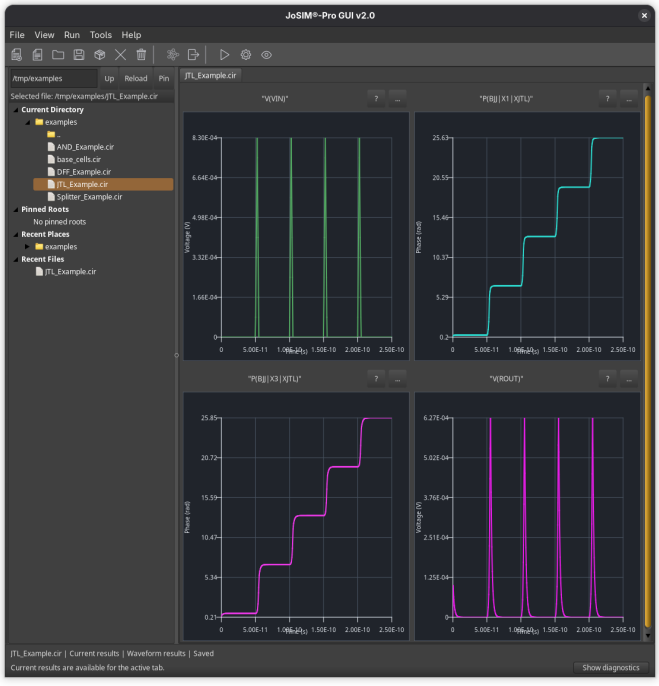
6. Inspect the waveform

After a successful run, the result view displays the traces requested by the `.PLOT` line. For `JTL_Example.cir`, expect traces for the input source, internal Josephson-junction branch phases, and the output resistor voltage.

Use the waveform area for the first inspection pass:

- verify that the expected traces appear;
- use trace visibility controls to hide or show traces;
- use `Run→Trace Manager` when result traces exist and you need a structured visibility/selection dialog;
- use `View→Grid` and `View→Sync Crosshairs` to control waveform-view behavior.

Do not switch to the CLI just to see the first waveform. CSV/TSV export and command-line processing are later workflow choices, not prerequisites for first use.



Successful GUI waveform result from the installed JTL example.

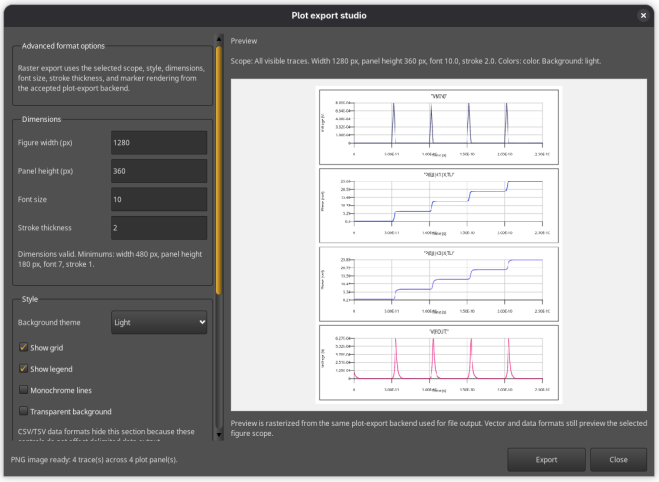
7. Export or save the result

When the waveform view is active and output is ready, use the toolbar export button to open the plot export workflow. The same export button is context-sensitive: in visualizer view it exports the schematic, while in waveform/result view it opens plot export for the current results.

For a first saved result:

1. keep the export scope at visible or all-visible traces unless you intentionally selected a subset;
2. choose a visual format such as `PNG image` for a report figure, or a data format such as `CSV data` / `TSV data` for downstream analysis;
3. save the file into your project directory rather than into the installed examples directory;
4. confirm from the tab status/diagnostics that the export succeeded.

Supported plot-export formats include `PNG`, `JPEG`, `SVG`, `PDF`, `EPS`, `TikZ/LaTeX`, `CSV`, and `TSV`. Data exports write selected trace data and x-axis values; visual style controls apply to raster/vector plot exports, not to CSV/TSV output.



Plot Export Studio configured for a visual PNG export.

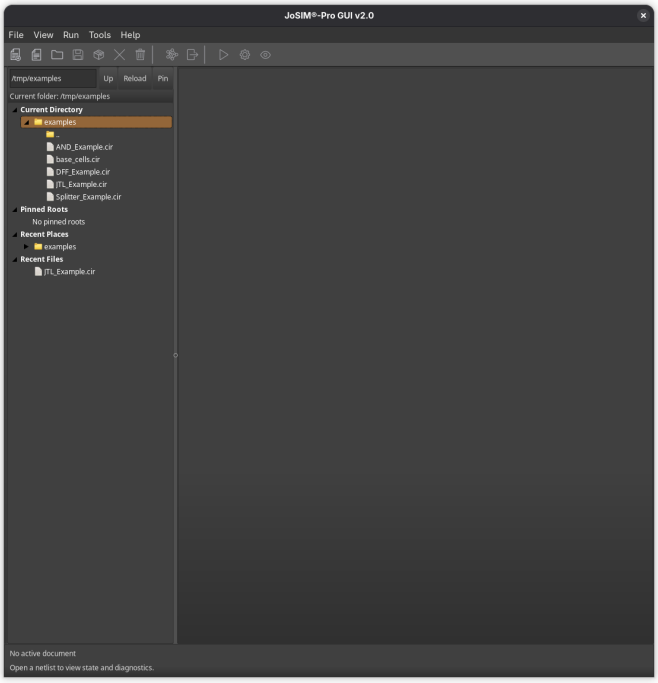
Use `File→Save As...` only when you want to save a modified copy of the netlist. It does not replace the plot/data export path for waveform results.

8. Continue from here

After the first successful run, continue with the GUI workspace chapter for file, folder, editor, and visualizer workflows. Continue with the GUI simulation/results chapter for trace placement, waveform inspection, and settings behavior. Existing terminal users can skip to the GUI-to-CLI equivalence chapter and the CLI command reference.

GUI Workspace, Editor, and Visualizer

This chapter expands the GUI workspace material into task-level documentation for files, tabs, the editor, and the schematic visualizer. Simulation execution, waveform analysis, Trace Manager, and plot export are covered in the next GUI chapter.



Workspace overview with examples folder, editor area, toolbar, and tab surface visible.

1. Workspace overview

The GUI is organized around workspace tabs. A tab can contain an editable netlist, an imported result data file, a waveform result produced by simulation, or a visualizer view built from the active netlist. The left file browser is a project navigation aid; it does not replace the normal `File→Open...` dialog.

Use the GUI workspace for interactive work:

- create or open a `.cir` netlist;
- edit and save the netlist in a workspace tab;
- open a folder into the file browser for project navigation;
- switch between editor and visualizer views for the active netlist;
- import prior `.csv` or `.tsv` result data for GUI-side waveform inspection;
- preserve recent files, recent places, pinned roots, and session state between launches.

Scope note. The visualizer is a viewing and layout tool for existing netlists. It is not documented as a full schematic-capture replacement in the current release.

2. Toolbar and context-sensitive actions

The toolbar provides shortcuts for common file, view, run, trace, and export operations. Some buttons are disabled until the active tab supports the requested action. For example, schematic export is available only while the visualizer is active, while plot export is available when waveform output is ready.

The export toolbar button is context-sensitive:

- in visualizer view, it opens the `Export schematic` workflow for PNG or SVG schematic export;
- in waveform/result view, it opens the plot export workflow for result figures or data files.

3. Creating, opening, and saving files

The `File` menu is the primary workspace entry point. Use these actions for normal file work:

Action	Behavior
`File→New`	Creates a new untitled netlist tab. The tab must be saved before the first simulation run.
`File→Open...`	Opens a JoSIM netlist or result data file. The dialog title is `Open netlist or result data`.
`File→Open Folder...`	Changes the file browser root to a selected directory. It does not automatically open every file in that directory.
`File→Save`	Saves the active editor tab. Untitled tabs prompt for a `.cir` save path.
`File→Save As...`	Saves the active netlist tab to a new `.cir` path.
`File→Save All`	Saves every modified netlist tab, prompting where needed.

The normal open path accepts these file types:

- `.cir` : opened as an editable netlist tab;
- `.csv` : opened as imported result data;
- `.tsv` : opened as imported result data.

Opening an already-open file activates the existing tab instead of creating a duplicate tab. The workspace tracks open files by normalized path so the same file is not edited from two tabs at once.

Imported `.csv` and `.tsv` result tabs are for waveform inspection, Trace Manager analysis, and plot export. They are not source netlists and cannot be simulated; open the corresponding `.cir` file to run a simulation.

4. File browser, recent places, and pinned roots

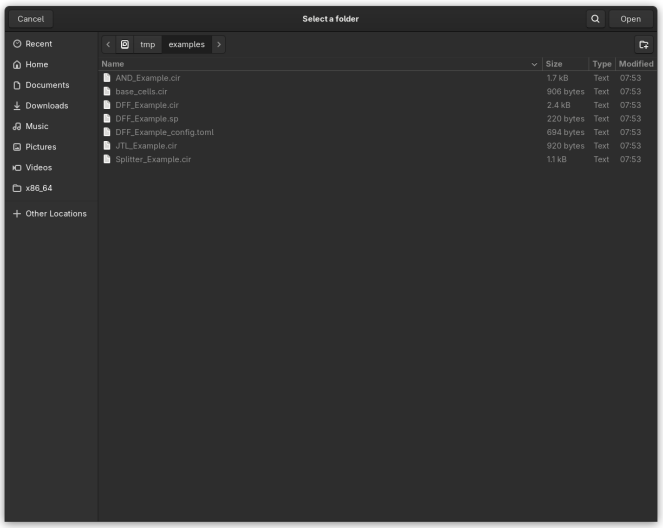
The file browser provides a collapsible project tree on the left side of the workspace. Set its root with `File→Open Folder...` or by entering a directory path in the address bar. Double-click a supported file to open it. Double-click a directory to navigate into it.

The browser lists folders, JoSIM netlists, and imported-result file types. Hidden entries and unsupported files are omitted unless the GUI preference for hidden entries is enabled. The browser recognizes the same supported open-file set as the normal open path: `.cir` , `.csv` , and `.tsv` .

The file browser also maintains navigation memory:

- **Pinned Roots**: persistent project roots selected by the user. Use the pin control or `Ctrl/Cmd+P` on a selected folder.
- **Recent Places**: directories used by recent open or save operations.
- **Recent Files**: supported files recently opened through the GUI.

Use pinned roots for stable project locations such as installed examples, active design directories, and shared benchmark folders. Use recent places for short-lived navigation.



File-browser workflow pointed at the installed examples directory.

5. Drag-and-drop opening

Files can also be dragged from the operating-system file manager into the GUI window. Dropped `.cir` , `.csv` , and `.tsv` files are routed through the normal open path. Unsupported dropped items are ignored and reported in the context/status area.

This behavior is useful for reopening CLI-produced data in the GUI. For example, a `josim-pro-cli` run can write a `.csv` file, and that file can then be dropped into the GUI for waveform inspection and export.

6. Session restore and persistent workspace state

The GUI stores user workspace configuration in the user's configuration directory. Persisted state includes the working directory, window size/state, file-tree panel width and visibility, theme, visualizer grid style and size, waveform grid and synchronized-crosshair preferences, hidden-file browser preference, and whether workspace tabs are restored on launch.

When session restore is enabled, the GUI reopens persisted tabs on startup. Session state records the active file, tab view mode, waveform layout preferences, waveform color preferences, and analysis recipes. If files are missing or fail to reload, startup continues and the GUI reports a `Session Restore Incomplete` warning rather than treating the problem as a fatal launch failure.

Session restore stores paths to files; it does not embed file contents. Keep moved or renamed project files in reachable locations if you expect them to reopen automatically.

7. External file changes

The workspace watches opened files for changes made outside the GUI. If a file changes on disk while the tab is inactive, the tab title and status can indicate an external-change state. The GUI then asks how to proceed before potentially destructive actions such as save, simulation, visualization, or close.

The important rule is simple: resolve external changes before continuing a run. If the disk copy and the in-memory tab both changed, choose deliberately between reloading from disk and overwriting the external version.

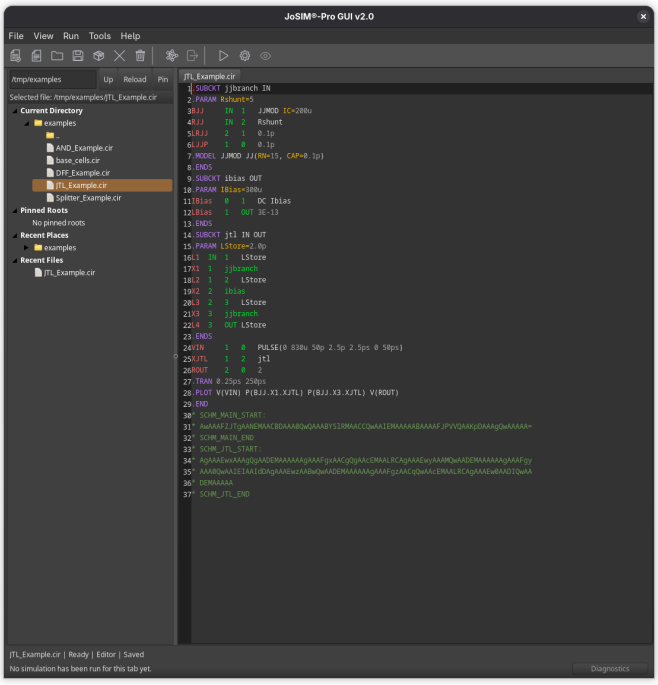
8. Editor basics

A `.cir` netlist opens in the text editor. The editor is designed for direct netlist editing and review, not for hiding the source syntax. It provides:

- line numbers and current-line indication;
- syntax highlighting for JoSIM netlist structure;
- bracket match and bracket error highlighting;
- search highlighting;
- find and replace with next/previous navigation;
- copy, cut, paste, delete, select-all, undo, and redo;
- keyboard navigation, word deletion, tab insertion, and line movement;
- modified-state tracking so unsaved tabs are visibly distinct.

Common shortcuts follow desktop-editor conventions. Use `Ctrl/Cmd+F` for find, `F3` or `Ctrl/Cmd+G` for the next match, `Shift+F3` or `Ctrl/Cmd+Shift+G` for the previous match, and `Esc` to clear search highlighting or close the search panel.

Saving an edited netlist updates the tab state and the recent-file/recent-place lists. Simulation uses a saved netlist file: if the active tab is untitled, the GUI prompts for a save path before running.



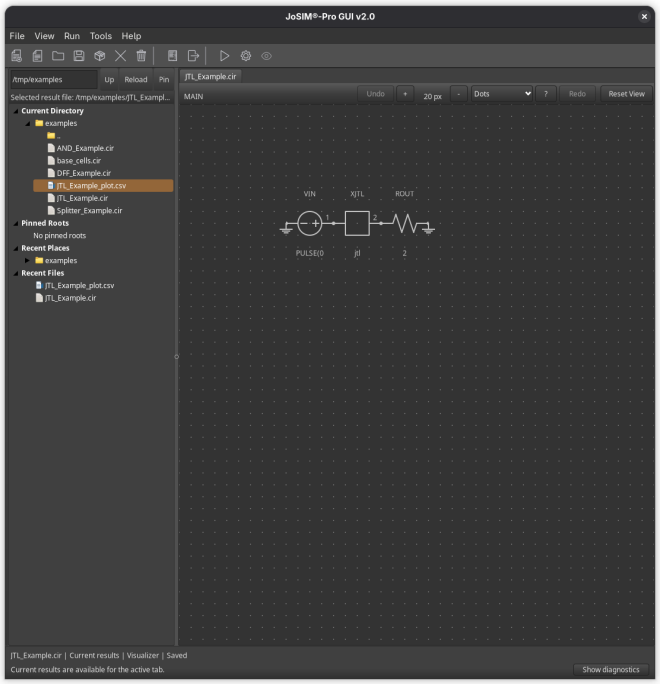
Editor view with syntax-highlighted `JTL_Example.cir` *content.*

9. Switching to the netlist visualizer

Use the visualizer toolbar button to switch the active netlist tab from text editor view to schematic visualizer view. The visualizer parses the current netlist, creates component blocks and connections, and displays the circuit as a schematic-like layout.

If parsing fails, the GUI keeps the tab in editor view and reports `Visualization Failed`. Fix the netlist syntax or include-path problem before trying again. Imported result tabs cannot be visualized because they do not contain a netlist.

When the netlist text changes, the visualizer is rebuilt from the current document revision the next time it is opened. Existing layout data is reused where possible; newly added components are placed automatically.



Visualizer view for the installed JTL example.

10. Visualizer controls and layout edits

The visualizer supports interactive inspection and cosmetic layout edits:

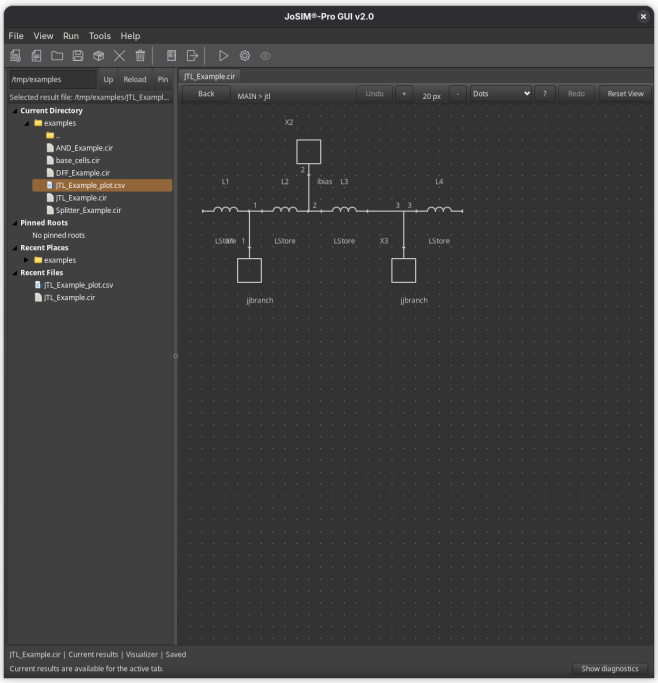
- drag a component to reposition it;
- drag empty canvas space to pan;
- use the mouse wheel to zoom around the pointer;
- use `Ctrl/Cmd+0` to reset zoom and pan;
- select a component and use arrow keys to move it;
- hold `Shift` with arrow keys for larger movement steps;
- press `R` or `Shift+R` to rotate the selected component;
- use `Ctrl/Cmd+Z` and `Ctrl/Cmd+Y` for visualizer layout undo/redo;
- press `G`, or use the grid-style selector, to cycle dots, lines, and no grid;
- use `Ctrl/Cmd+Mouse Wheel` or the grid controls to change grid spacing.

These edits are layout edits. They affect how the netlist is drawn in the GUI and exported as a schematic; they do not change the electrical meaning of the circuit unless the underlying netlist text is edited separately.

11. Subcircuit navigation

Subcircuits are drawn as blocks with pins and a subcircuit label. Double-click a subcircuit block to enter the referenced subcircuit. The visualizer then redraws that subcircuit scope. Use the visible `Back` button, `Backspace`, or `Alt+Left` to return to the parent schematic.

If a subcircuit reference cannot be resolved, the GUI reports `Open Subcircuit Failed` or `Subcircuit Not Available`. In that case, check the subcircuit name, included files, and whether the parser can see the referenced `.SUBCKT` definition.



Subcircuit navigation view with parent/child schematic context.

12. Layout persistence

Visualizer layout is persisted into the netlist as tagged comment data. When layout changes are saved, the GUI writes an encoded `SCHM_<design>` comment block for the active design scope. Subcircuit layouts can be saved independently for the subcircuit scope being viewed.

This means the layout travels with the netlist file and can be restored when the same design is opened later. It also means layout changes make the file modified even though the circuit semantics are unchanged. The GUI labels these as cosmetic layout changes so they can be distinguished from semantic netlist edits.

Do not manually edit or delete the encoded schematic comment block unless you intend to discard the stored layout. If the netlist topology changes, the visualizer reuses stored positions for matching components and places new components automatically.

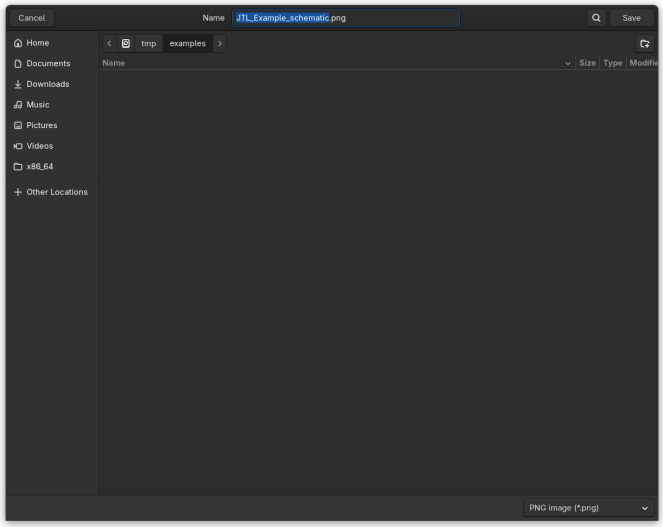
13. Schematic export

While the visualizer is active, use the toolbar export button to open `Export schematic`. The schematic export dialog supports:

- `PNG image (*.png)` for raster output;
- `SVG vector schematic (*.svg)` for scalable vector output.

If no extension is supplied, the GUI uses the selected filter to add `.png` or `.svg`. The export path defaults from the active netlist path when available. Exported schematics use the prepared visualizer scene rather than a screen capture, so the saved file is suitable for reports and presentation material.

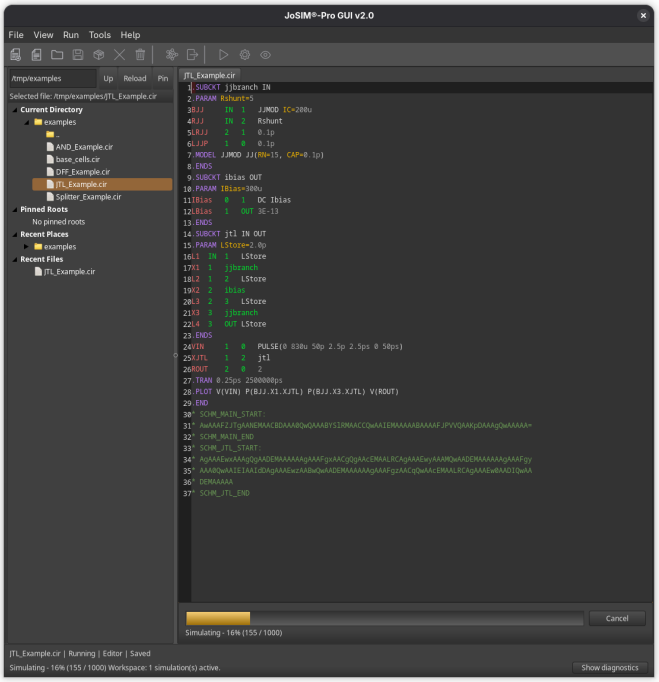
Use plot export, not schematic export, for waveform/result figures. The same toolbar button chooses the correct workflow based on whether the active tab is showing the visualizer or waveform results.



Schematic export dialog with visualizer-specific PNG/SVG output options.

GUI Simulation and Results

This chapter documents the GUI-side simulation, waveform, Trace Manager, imported-result, analysis, and plot-export workflows. The quick-start chapter gives a first run; this chapter is the task reference for day-to-day result work.



Simulation progress overlay while the active netlist is running.

1. Simulation lifecycle

Use the toolbar simulate button, or the equivalent run command in the active tab, to run the current netlist. The GUI runs the same JoSIM-Pro runtime used by `josim-pro-cli`; the difference is that progress, diagnostics, and waveform output are managed by the active GUI tab.

The GUI first reads the saved netlist and validates every analysis and output destination. It then executes analyses serially in source order, with fresh simulation state for each run. Each successful run publishes its own files and retains its requested traces. The result selector identifies runs by block name, analysis identifier and completion status.

During a run, the tab shows a progress overlay with a progress bar and `Cancel` button. If the runtime reports percent-style progress, the overlay shows the percentage and sample count; otherwise it shows the current stage text. The context/status area also reports the active stage.

Cancelling requests an abort of the active analysis and changes the overlay button to `Cancelling...`. Wait for the worker to finish before starting another execution. Completed analyses from that execution remain available; the interrupted analysis does not publish incomplete staged output. Diagnostics identify failed, cancelled and unstarted analyses.

2. Preconditions for running

Simulation is available only for netlist tabs, not imported result tabs. A `.csv` or `.tsv` tab can be inspected, analysed, and exported, but it cannot be simulated because it does not contain source netlist text.

The GUI runs from a saved netlist file. If the active tab is untitled, the GUI prompts to save it before the first run. If the tab has unsaved semantic netlist edits, the GUI asks whether to save and run the current text, run the last saved file, or cancel. Running the last saved file leaves the displayed result state marked stale when the unsaved text differs from the saved file.

Resolve external file changes before running. If the file on disk changed outside the GUI, the tab reports an external-change state and asks whether to reload or keep the current in-memory copy before simulation continues.

> **License requirement.** GUI simulation requires the latest SUN Magnetics License Manager and a working `smlicensed` daemon. Degraded mode is for inspection and recovery; it is not a licensed simulation mode.

3. Run settings

Open `Run→Settings` from a netlist tab to edit defaults for every analysis in the active netlist. Netlist options and block-local options can override these defaults. Run settings are disabled while that tab is simulating and are unavailable for imported result tabs. Changing run settings after a successful run marks existing output stale until the tab is re-run.

Analysis type Chooses `Phase` or `Voltage` analysis for the runtime.

Integration method Chooses `BDF` or `TRAP`.

Variable-step mode Enables variable-step mode only for the supported `Phase` + `BDF` combination. Other combinations are rejected by inline validation.

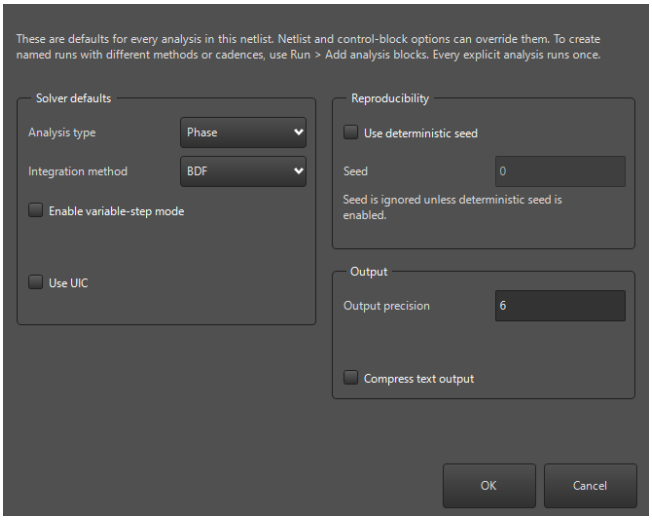
Use UIC Passes the UIC flag through to the simulation runtime.

Output precision Sets output digits in the runtime-supported range `0` to `17`. Invalid values block `OK`.

Compressed output Requests compressed text output when supported by the runtime and output path.

Deterministic seed Enables a fixed unsigned seed for reproducibility. If disabled, the seed field is ignored and the runtime receives seed `0`.

Defaults are loaded from the installed runtime configuration when the tab is created. Keep the defaults for normal first-use simulations. Change settings when you need CLI-equivalent reproducibility, a support-requested solver configuration, or a deliberate comparison between analysis/integration modes.



Run Settings supplies deck defaults; control blocks can override them.

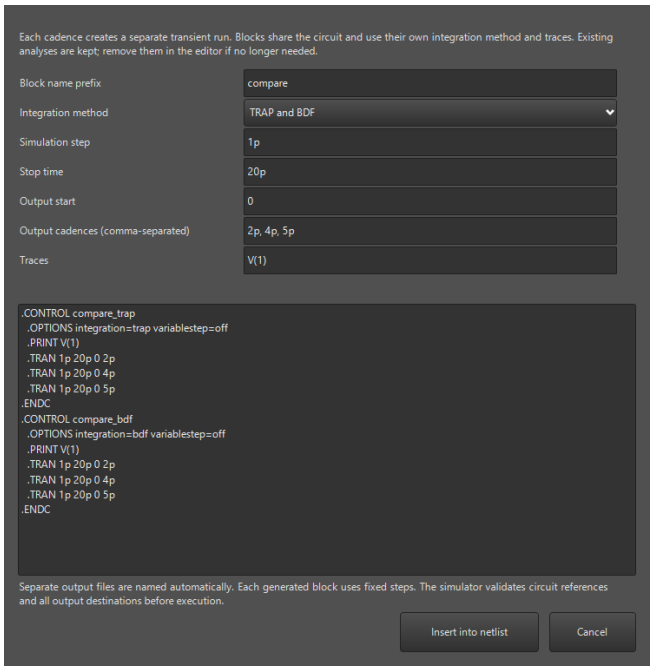
3a. Create multiple runs

Open `Run→Add analysis blocks...` from an editable netlist tab. The dialog inserts ordinary netlist commands that can be inspected and edited. It is disabled while a simulation is running and for imported result tabs.

To compare TRAP and BDF at three output cadences:

1. Enter a unique block name prefix, for example `compare`.
2. Select `TRAP and BDF`, a simulation step of `1p`, a stop time of `20p`, and an output start of `0`.
3. Enter `2p, 4p, 5p` as the output cadences and choose traces present in the circuit, for example `V(1)`.
4. Review the preview and select `Insert into netlist`. Save the netlist, then press Run.

This creates `compare_trap` and `compare_bdf`, each with three independent transients: six runs in total. Each block uses fixed simulation steps and its own integration method and traces. Separate output files receive automatic names. The simulation step controls numerical advancement; the output cadence controls the retained sample grid.



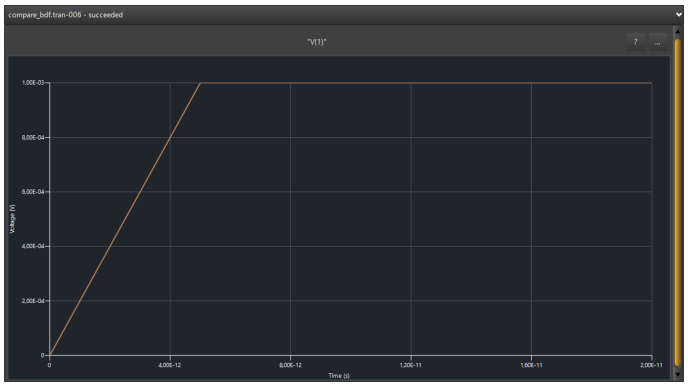
Two named blocks, each containing three transient analyses.

The dialog accepts literal times in seconds or with suffixes such as `p` and `n`. It keeps existing analyses and inserts the generated blocks before `.END`; remove unwanted existing analyses in the editor. A repeated block name is rejected. The simulator checks circuit references and output destinations before execution.

For other combinations, add more blocks with distinct pre-

fixes or edit the generated commands. Blocks may contain multiple supported analyses and their own options and output directives. Use the netlist syntax chapter for parameters, explicit `.FILE` destinations and IV analyses.

After execution, choose a run from the selector above the plots. Each selection loads that run's own traces and sampling grid; IV results use voltage as the horizontal axis. In this example the three cadences produce 11, 6 and 5 samples respectively, including zero. Changing the selection rebuilds the plot workspace: export derived traces or customised plots before switching runs.



Selecting the BDF run with a 5 ps output cadence. Other run results remain separately selectable.

4. Result states and diagnostics

The active tab reports both document state and result state. Use these states to decide whether the waveform view reflects the current netlist and run settings.

No results The tab has not produced or imported waveform data.

Running The tab has an active simulation worker. The result view updates only after the run completes.

Cancelling A cancel request was issued; wait for the worker to finish before starting another run.

Results current The waveform output matches the saved netlist and current run settings.

Results stale Semantic netlist text or run settings changed after the last successful run. Re-run before trusting the waveform as current output.

Run failed The runtime reported an error. Diagnostics include the latest failure stage and error text when available.

Run cancelled The last run was cancelled by the user. Previous successful results may still be visible.

Imported results The tab contains loaded `.csv` or `.tsv` result data rather than a source netlist.

Use the diagnostics toggle in the tab/status area when a run fails, output looks stale, an external-change warning appears, or an export reports a problem. Diagnostics include the active file, surface mode, document state, result state, recent progress stage, last run/export summary, latest issue, warnings, and recent run history.

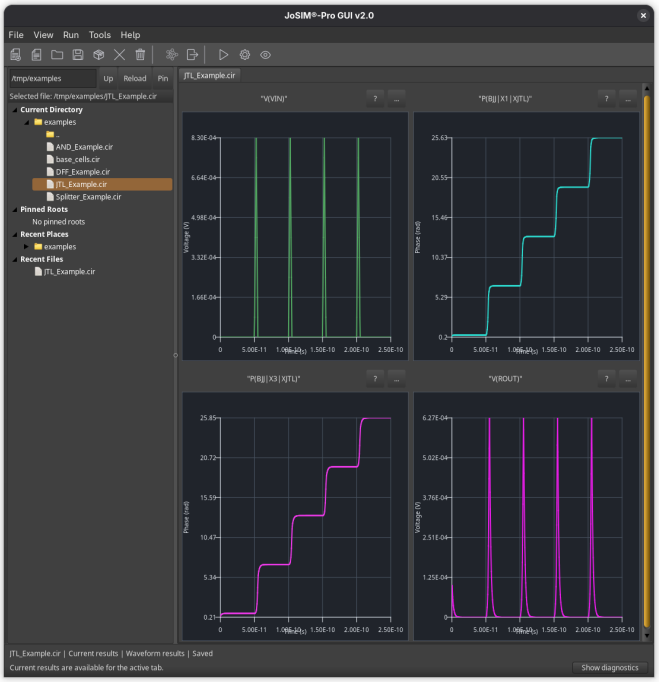
Native output-file routing is also reported in diagnostics. Each successful analysis publishes its own output plan. If `.FILE` appears only after existing `.PRINT`, `.SAVE`, or `.PLOT` commands, the GUI warns that `.FILE` routes only subsequent output requests; move `.FILE` before the output commands when file routing is intended.

5. Waveform result view

Successful simulation opens the waveform result view in the same tab. Imported result files also open directly into this waveform surface. The waveform workspace supports multiple plot panels, visible/hidden traces, trace grouping, crosshair inspection, grid display, analysis outputs, annotations, and export.

Use `View→Grid` to toggle plot-grid display for the active waveform workspace. Use `View→Sync Crosshairs` to synchronize crosshair movement across plot panels. These view preferences are stored with GUI preferences and reapplied to new result workspaces.

Traces can be inspected directly in the plot area. The screenshot below shows the waveform surface with multiple plot panels and visible traces from the installed JTL example.



Waveform result view with grid, panels, and plotted traces.

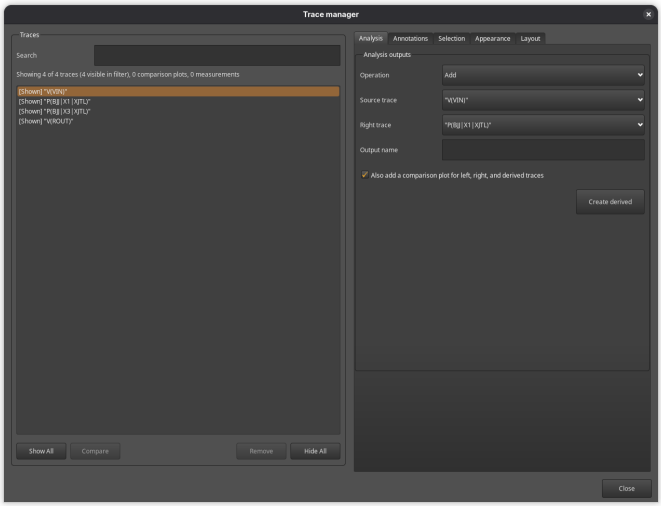
6. Trace Manager basics

Open `Run→Trace Manager` or the toolbar trace-manager button after results exist. Trace Manager is disabled while the tab is simulating and is enabled for both simulated results and imported `.csv` / `.tsv` result tabs.

The Trace Manager dialog contains a trace list on the left and task tabs on the right:

- **Analysis**: create derived traces, measurements, masks, FFT outputs, and comparison plots.
- **Annotations**: add exact vertical, horizontal, point, or interval annotations.
- **Selection**: rename selected items, toggle visibility, assign groups, reorder traces, and override trace colors.
- **Appearance**: choose plot-wide theme and color settings.
- **Layout**: choose plot-panel layout mode.

The trace list supports search, multi-selection, `Show All`, `Hide All`, removal of derived outputs, and non-destructive comparison plots. Use `Ctrl/Cmd+A` to select all items and `Ctrl/Cmd`-click for multi-selection.



Trace Manager with trace selection and analysis controls.

7. Visibility, order, groups, and comparison plots

Use the **Selection** tab for practical trace organization:

- toggle `Visible for selected trace(s)` to show or hide selected traces;
- edit `Selected item name` to rename a trace or comparison plot for display/export;
- use `Assign group` and `Reset` to move traces into named plot groups or return them to default grouping;
- use `Move Up` and `Move Down` to change plotting order;
- apply or clear selected-trace line color overrides;
- use `Compare` to create an additive comparison plot from the current selection.

Trace grouping is a display organization tool. It does not modify the source netlist or simulation output. Comparison plots are also non-destructive; they add alternate views over existing or derived traces rather than replacing the original data.

The waveform workspace remembers layout, color, and analysis recipes through session state. If a later run or import cannot restore a previous analysis recipe because a trace name or axis is missing, the tab reports restore warnings in diagnostics.

8. Analysis outputs

The **Analysis** tab creates new outputs from existing traces. Choose a source trace, optional right-hand trace, an operation, scalar parameters where required, and an output name. For trace-producing operations, `Create derived` adds a derived trace to the workspace. For measurement-producing operations, the button changes to `Measure` and records a measurement output, often with markers.

Available operation families include:

- arithmetic and scalar arithmetic such as add, subtract, multiply, divide, `x+k`, `x-k`, `x*k`, and `x/k`;
- transforms such as absolute value, negation, DC removal, subtract-initial, logarithmic forms, derivative, and cumulative integral;
- normalization and smoothing operations;
- threshold masks, range masks, trace-comparison masks, and mask logic;
- FFT magnitude, FFT dB magnitude, FFT phase, and power spectrum;
- measurements such as min, max, mean, RMS, peak-to-peak, crossing time/count, error metrics, rise/fall time, propagation delay, pulse width, duty cycle, period/frequency estimate, overshoot, and settling time.

Operations that combine traces require compatible x-axis metadata. FFT operations require a uniform time axis. The dialog exposes FFT controls for window selection, zero-padding policy, optional DC removal, and dB floor epsilon. Invalid choices are rejected in the analysis validation area rather than producing a silent output.

Derived traces and measurements are workspace outputs. They can be hidden, renamed, grouped, exported, removed, and restored as analysis recipes when session state can match their source traces.

9. Manual annotations

Use the **Annotations** tab to add exact markers from numeric coordinates. Choose a trace target, marker kind, label, and coordinates, then click `Add exact`. Supported marker kinds are:

- `Vertical marker`;
- `Horizontal marker`;
- `Point annotation`;
- `Interval marker`.

Annotations can be visible or hidden. Enable `Include in visual exports` when the marker should appear in image/vector plot exports. This flag affects visual exports only; CSV/TSV data export writes numeric trace data, not rendered annotation geometry.

10. Imported CSV and TSV result data

Use `File→Open...`, the file browser, or drag-and-drop to open existing `.csv` and `.tsv` result files. Imported result files become waveform tabs immediately; the editor surface is hidden because the tab represents data, not source text.

The import reader accepts comma-delimited and tab-delimited numeric data. Blank lines and lines beginning with `#` are ignored. A header row is optional; when present, the first column is used as the x-axis label and every remaining column becomes a trace. When no header is present, the first column is treated as `Time (s)` and trace names are generated automatically.

CSV extension Uses comma as the delimiter.

TSV extension Uses tab as the delimiter.

Unknown delimiter Uses the more frequent comma/tab character in the first data line.

Header row present Uses header text for axis and trace names. Units in parentheses are recognized where possible.

No header row Generates `Time (s)` and `Trace 1`, `Trace 2`, and so on.

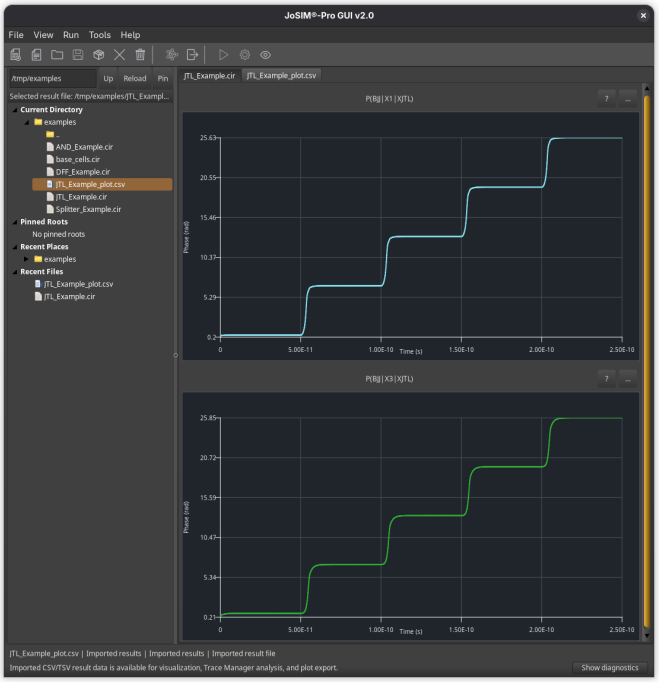
Non-numeric cell Import fails with the offending line number.

Mismatched column count Import fails with the offending line number and expected column count.

Too few samples Import requires at least two numeric samples.

Ordered axis required Time, frequency, lag, and sample axes must be non-decreasing.

Imported result tabs support Trace Manager analysis and plot export. They do not support run settings, simulation, save/save-as as netlists, or netlist visualization. If an imported file changes on disk, the GUI asks whether to reload the CSV/TSV data and rebuild the plot workspace.

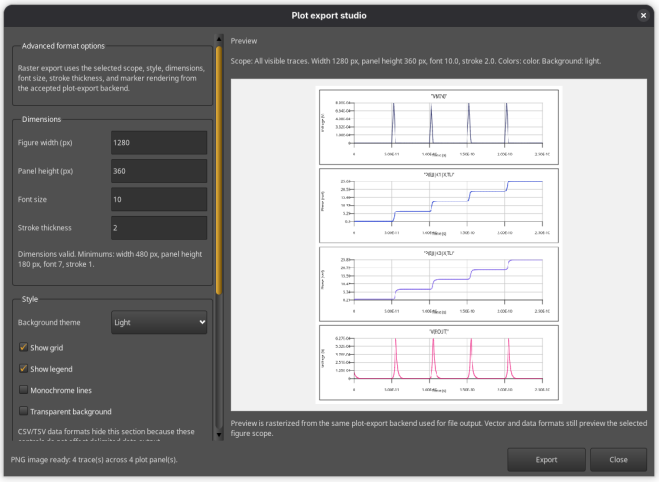


Imported CSV/TSV result data opened as a GUI waveform tab.

11. Plot export studio

When the active tab is showing waveform results, the toolbar export button opens `Plot export studio`. This is separate from schematic export: schematic export is available from the visualizer; plot export is available from the waveform/result view.

Plot export studio provides a live preview and file output using the same plot-export backend. The left side controls preset, format, scope, trace selection, style, dimensions, and advanced format disclosure. The right side shows the preview and export-readiness summary.



Plot Export Studio configured for a PNG plot export with preview.

Supported export formats are:

- raster figures: `PNG image`, `JPEG image`;
- vector figures: `SVG vector`, `PDF vector`, `EPS vector`, `TikZ/LaTeX`;
- data files: `CSV data`, `TSV data`.

Supported export scopes are:

- `Selected traces`;
- `Chosen visible group`;
- `All visible traces`;
- `All traces`.

For image/vector exports, use the presets `Publication light`, `Presentation dark`, or `Monochrome print`, then adjust grid, legend, monochrome, transparent background, figure width, panel height, font size, and stroke thickness as needed. Minimum values are enforced: width at least `480` px, panel height at least `180` px, font size at least `7`, and stroke thickness at least `1`. JPEG export does not support transparent background.

For CSV/TSV export, style and dimension controls are hidden because delimited data output writes x-axis values and selected trace series only. Delimited export requires compatible x-axis metadata across the selected groups/traces; if axes are incompatible, the export fails with an explanatory message.

The save dialog uses the selected export format to choose the default extension. If no extension is supplied, the GUI adds the format's default extension. Successful and failed exports are recorded in tab diagnostics.

12. Troubleshooting hooks

Use these checks when simulation or result handling does not behave as expected:

Simulation button is disabled Confirm the active tab is a netlist tab, not an imported `.csv` / `.tsv` result tab.

Run settings are disabled Wait for the active simulation to finish, or switch from an imported result tab to a netlist tab.

Run prompts for save The tab is untitled or has unsaved semantic edits. Save the netlist if the current text should be simulated.

Results are marked stale Re-run after changing semantic netlist text or run settings. Cosmetic visualizer layout changes do not invalidate simulation results.

No waveform appears Check diagnostics for the failing runtime stage and confirm the netlist contains output requests such as `.PLOT`, `.PRINT`, or `.SAVE`.

Native output file is missing If using `.FILE`, place it before the output requests it should route. Check diagnostics for native-output routing messages.

CSV/TSV import fails Check the reported line for non-numeric cells, mismatched column count, too few samples, or decreasing x-axis values.

Trace Manager cannot create output Check operation compatibility. Binary operations require compatible axes; FFT requires a uniform time axis; scalar operations require valid numeric parameters.

Plot export fails Check export scope and axis compatibility. Data export cannot combine traces with incompatible x-axis metadata. JPEG cannot use transparent background.

Session restore warns about analysis A saved analysis recipe referenced a trace or axis that is no longer present. Recreate the derived output or measurement from current traces.

For licensing, package launch, or System ID problems, use the installation and troubleshooting chapters first. For CLI-equivalent run settings, use the GUI-to-CLI equivalence chapter.

Optimum CLI Workflows: Verify, Margin, Yield, and Optimize

Optimum is the specification-checking and design-tuning companion to JoSIM®-Pro. Optimum is driven by `optimum-cli` and a TOML configuration file. The GUI can prepare and inspect the files used by Optimum, while the Optimum run itself is launched from the command line.

Use the GUI to edit and save netlists, inspect an optimized netlist, or open CSV/TSV result data generated by CLI workflows. Use `optimum-cli` when the task is specification verification, margin search, yield estimation, or automatic parameter optimization.

1. GUI handoff model

A normal Optimum workflow in a project that uses both the GUI and command-line tools is:

1. Open or create the design netlist in the GUI.
2. Save the netlist as a stable `.cir` file. If the design uses includes, keep the TOML file, circuit, specification file, and included files in a predictable project directory.
3. Create or edit an Optimum TOML configuration that points to the saved circuit and the verification specification. For EVENT verification, `--event-config` can list the circuit's available Josephson-junction phase signals and generate the rule block interactively.
4. Run `optimum-cli` from a terminal.
5. Reopen the resulting optimized `.cir` file or any generated CSV/TSV result data in the GUI for inspection, waveform analysis, and plot export.

This is a handoff workflow, not a hidden GUI integration. The GUI's `File -> Open...` workflow accepts `.cir`, `.csv`, and `.tsv` files, so it is suitable for reviewing Optimum inputs and downstream result data. It does not replace `optimum-cli` for the optimization run itself.

2. Installed DFF example

The package installs the canonical Optimum example files:

- `DFF_Example.cir` – circuit netlist,
- `DFF_Example.sp` – SP-style verification specification, and
- `DFF_Example_config.toml` – Optimum configuration.

These files are installed with the other examples under the package's `examples` directory. Use them for the first Optimum test because they are available after installation.

From the installed examples directory, run:

```
optimum-cli DFF_Example_config.toml
```

The provided configuration uses `mode="optimize"` by default. To test the other modes without changing file paths, edit only the top-level `mode` line:

```
#mode="verify"
#mode="margin"
#mode="yield"
mode="optimize"
```

If `optimize.output` remains set to `"DFF_Example_optimized.cir"`, an optimization run writes the optimized netlist beside the example files. `DFF_Example_optimized.cir` is generated output, not a required installed example file. Open that generated `.cir` file in the GUI to inspect the changed `.PARAM` values or to run a normal GUI simulation.

3. CLI invocation and licensing

`optimum-cli` accepts either a positional TOML file or `-i` / `--input`. A valid JoSIM-Pro license and the installed SUN Magnetics License Manager are still required for simulation-backed verification. The `--event-config` helper only parses the circuit and exits after writing an EVENT rule; it does not run a simulation or perform license validation.

Option	Meaning
`input_file`	Positional TOML input file.
`-i` , `--input`	TOML input file path. Equivalent to the positional file.
`--event-config`	Interactively list Josephson-junction phase signals available in the input circuit and write one EVENT `[[verify.rules]]` block to standard output.
`-l` , `--license`	Save or use a license-file path. If no input file is supplied, the path is saved and the tool exits after validation.
`-s` , `--sysid`	Ask the installed `smlicensed` daemon to generate `SM_SysID.txt` in the current directory.
`-r` , `--seed`	Override the random-number seed used by stochastic Optimum paths.
`-m` , `--minimal`	Reduce normal execution feedback.
`-V` , `--verbose`	Set verbose execution feedback, with `0` for off and `1` for on.
`-v` , `--version`	Print version information.
`-h` , `--help`	Print help and exit.

Examples:

```
# Positional TOML file
optimum-cli DFF_Example_config.toml

# Explicit input option
optimum-cli -i DFF_Example_config.toml

# Deterministic stochastic run
optimum-cli -r 123456 DFF_Example_config.toml

# Generate a licensing system identifier through smlicensed
optimum-cli -s

# Generate an EVENT rule from the DFF circuit
optimum-cli --event-config DFF_Example.cir
```

The EVENT helper writes prompts and the phase-signal inventory to standard error, while the generated TOML rule is written to standard output. This allows the rule fragment to be redirected without mixing it with prompts:

```
optimum-cli --event-config DFF_Example.cir > event_rule.toml
```

The helper accepts either the displayed signal number or its label. It supports `transfer` , `fanout` , `merge` , `delay` , `phase_shift` , `logic` , `clocked_logic` , and `clocked_transfer` . For logic rules it also asks for the function. It does not infer expected event times, waveform shapes, or signal roles.

4. Mode overview

At the top level, set `mode` to one of the four supported mode names. The executable lower-cases the mode value before dispatching.

Mode	Use case
`verify`	Run a nominal circuit and check it against a specification. This is the gate that all other modes must pass first.
`margin`	Search how far selected parameters can move down and up before verification fails.
`yield`	Estimate the pass fraction under randomized parameter variation.
`optimize`	Search for a better parameter center using verification and margin analysis as the score path.

All four modes use the same top-level execution sequence:

1. parse the TOML configuration,
2. load the circuit and verification specification,
3. run baseline verification on the nominal circuit, and
4. continue with the selected mode only if that nominal verification passes.

If the nominal circuit does not verify, Optimum stops before running Margin, Yield, or Optimize.

5. Runtime contract

Optimum currently enforces fixed-step transient `PHASE` analysis with `BDF` integration. Variable-step execution is intentionally disabled inside Optimum because the verification paths assume uniformly sampled time data.

In practice:

- Optimum is not the path for comparing `PHASE` against `VOLTAGE` operation.
- Optimum is not the path for comparing `BDF` against trapezoidal integration.
- Netlist or runtime settings that would move the run outside fixed-step `PHASE + BDF` are overridden or rejected by the Optimum runtime contract.

Use the GUI or `josim-pro-cli` for interactive solver-option comparisons. Use Optimum after the verification circuit is known to run correctly under the fixed-step `PHASE + BDF` contract.

6. File paths and project layout

The TOML configuration file is the anchor for Optimum inputs. Relative values in `verify.circuit`, and in `verify.file` when SP or CSV verification uses it, are resolved relative to the configuration file location. The circuit's own includes are then resolved through the normal simulator input handling from the circuit working directory.

Recommended layout:

```
my_dff_project/
  DFF_Example_config.toml
  DFF_Example.cir
  DFF_Example.sp
  base_cells.cir
```

For output files, prefer explicit names in the same project directory. Margin and Optimize write relative output paths against the circuit working directory. Yield roll-off output uses the process working directory, so run `optimum-cli` from the project directory when using relative `yield.output` paths.

7. Minimal configurations

7.1 Verify

```toml
mode = "verify"

[verify]
method = "spec_file"
file = "DFF_Example.sp"
circuit = "DFF_Example.cir"
threshold = 0.35
```

Use this when you only need to confirm that the circuit satisfies the reference behavior.

For EVENT verification, no SP or CSV reference file is required:

```toml
mode = "verify"

[verify]
method = "event"
circuit = "DFF_Example.cir"

[[verify.rules]]
type = "clocked_transfer"
input = "B1"
clock = "B6"
output = "B7"
```

Bare phase-signal names such as `B1` are accepted. Use `--event-config DFF_Example.cir` to list the circuit's available Josephson-junction phase signals and generate this rule block interactively.

7.2 Margin

```toml
mode = "margin"

[verify]
method = "spec_file"
file = "DFF_Example.sp"
circuit = "DFF_Example.cir"
threshold = 0.35

[margin]
accuracy = 0.01
binary_search_steps = 7
output = "DFF_Example_margins.txt"

[parameters]
search_string = "^(B\\d+|IB\\d+|L\\d+)$"
```

Use this when you want a per-parameter robustness summary around the nominal design.

7.3 Yield

```toml
mode = "yield"

[verify]
method = "spec_file"
file = "DFF_Example.sp"
circuit = "DFF_Example.cir"
threshold = 0.35

[yield]
num_samples = 2000
spread = 0.10

[parameters]
search_string = "^(B\\d+|IB\\d+|L\\d+)$"
```

Use this when you want a Monte-Carlo pass-rate estimate for a chosen spread.

7.4 Optimize

```toml
mode = "optimize"

[verify]
method = "spec_file"
file = "DFF_Example.sp"
circuit = "DFF_Example.cir"
threshold = 0.35

[margin]
binary_search_steps = 7

[optimize]
method = "cog"
spread = 0.60
samples = 1000
max_iterations = 500
output = "DFF_Example_optimized.cir"

[parameters]
search_string = "^(B\\d+|IB\\d+|L\\d+)$"
```

Use this when you want Optimum to search automatically

for a better margin center.

8. Configuration reference

8.1 `[verify]` – common to all modes

Key	Type	Default	Meaning
`method`	string	`"spec_file"`	Verification input format: `"spec_file"`, `"csv_file"`, or `"event"`.
`file`	string	–	Specification or CSV reference file. Required for `spec_file` and `csv_file`; not used by `event`.
`circuit`	string	–	Circuit netlist to simulate. Required.
`rules`	array of tables	–	EVENT semantic rules. Required and non-empty when `method="event"`.
`threshold`	float	0.05	Global tolerance used by SP checks and as a base tolerance in CSV comparison.
`current_threshold`	float	0.05	Current tolerance factor used for `L...` columns in SP-style specifications.
`amplitude_threshold`	float	0.50	CSV pulse-detection amplitude fraction.
`slope_threshold`	float	0.02	CSV slope threshold used when identifying stable and transition regions.
`min_pulse_width`	float	2.5e-12	Minimum event width considered meaningful in CSV processing.
`min_stable_samples`	int	3	Minimum consecutive stable samples for CSV processing.
`merge_gap_threshold`	float	1e-12	Merge adjacent CSV pulses separated by less than this gap.
`edge_time_tol_ps`	float	5.0	CSV edge-pairing tolerance in picoseconds.
`edge_rise_tol_frac`	float	0.10	Accepted by the current parser, but not presently active in the comparison path.
`edge_min_amp_frac`	float	0.20	Minimum normalized edge amplitude used by the adaptive CSV edge detector.
`edge_window_size`	int	200	Window size, in samples, for adaptive CSV edge detection.

EVENT rules are configured as `[[verify.rules]]` tables. Section 9.3 gives the complete EVENT rule reference, including the required keys, supported logic functions, timing behavior, and configuration examples.

8.2 `[margin]` – robustness search

Key	Type	Default	Meaning
`subcircuit`	string	–	Limit parameter selection to this subcircuit scope. Yield and Optimize read the same scope value.
`accuracy`	float	0.01	Margin search tolerance or exhaustive-search step size.
`binary_search_steps`	int	10	Maximum number of binary-search refinement steps.
`min_search`	float	0.10	Lowest scale factor tested on the negative side.
`max_search`	float	1.90	Highest scale factor tested on the positive side.
`exhaustive`	bool	false	Scan the interval in fixed `accuracy` steps instead of using binary search.
`pocket_search`	bool	false	Emit full pass/fail pocket maps instead of only summarized margins.
`output`	string	–	Optional text file for the printed margin summary or pocket map.
`parameter_dependence`	bool	false	Re-evaluate dependent parameter expressions when a searched parameter changes.

8.3 `[yield]` – Monte-Carlo yield

Key	Type	Default	Meaning
`num_samples`	int	100	Number of Monte-Carlo trials per yield estimate.
`spread`	float	0.0	Global fractional spread. If left at zero, explicit parameter entries may provide spreads; otherwise Optimum falls back to 10% of each nominal value.
`roll_off`	bool	false	Sweep spread from `step_size` to 1.0 and report yield at each step.
`step_size`	float	0.1	Roll-off sweep step.
`output`	string	–	Optional roll-off CSV output path. Used only for roll-off output.

8.4 `[optimize]` – automatic improvement search

Key	Type	Default	Meaning
`method`	string	`"cog"`	Optimization method selector. Optimum uses the center-of-gravity workflow.
`spread`	float	0.50	Global sampling spread around the current center.
`spread_decay`	float	0.75	Multiplicative spread reduction during stalled iterations; internally clamped.
`spread_grow`	float	1.05	Multiplicative spread increase after improvement; internally clamped.
`samples`	int	10000	Number of samples per optimization iteration; raised internally to at least 100.
`samples_max`	int	20000	Maximum adaptive sample count when the search upsamples after stalled progress.
`seed`	int	1234567	Base seed for repeatable optimization clouds.
`resample_every`	int	5	Reuse a common-random-number epoch for this many iterations before reseeding.
`trim_frac`	float	0.10	Two-sided trim fraction for the trimmed mean.
`ema_alpha`	float	0.50	Exponential moving average blend between the current center and the new estimate.
`sig_digits`	int	4	Significant digits used when writing optimized numeric `.PARAM` values.
`max_iterations`	int	500	Hard iteration cap.
`patience`	int	10	Number of non-improving iterations allowed before early stopping.
`converge`	float	-1.0	Stop when the gap to the current best margin stays within this threshold. Set `>0` to enable.
`target`	float	-1.0	Stop when the best critical margin reaches this target. Values `<=1.0` are interpreted as fractions and converted to percent.
`conv_k`	int	4	Number of consecutive within-threshold iterations required before `converge`

8.5 `[parameters]` – choosing the parameter set

Margin, Yield, and Optimize need a parameter set to work on. If `[parameters]` is omitted, Optimum falls back to all parameters in the active scope. That is valid, but it is usually slower and less informative than selecting a focused set.

Use `search_string` to select a family of parameter names:

```
[parameters]
search_string = "^(IBIAS|L_.*|C_.*)$"
```

Alternatively, enumerate parameters explicitly:

```
[parameters]
B1 = {}
IB1 = {}
L1 = {}
```

Explicit entries can provide values:

```
[parameters]
B1 = { nominal = 120e-6 }
IB1 = { nominal = 180e-6, spread = 15e-6 }
```

For these settings:

- `nominal` overrides the circuit nominal value used as the baseline for Margin, Yield, and Optimize,
- `spread` is used by Yield when drawing that parameter, and
- Optimize does not currently read per-parameter `spread`; it samples from the global `optimize.spread` around the current center.

9. Verification inputs and tolerances

9.1 `method = "spec_file"`

Use an SP-style expectation file when you want to check explicit values at specific times. The first column is time. Each remaining column names a trace and gives the expected value at that time.

Optimum interprets labels as follows:

- labels starting with `L` are treated as currents and compared using `verify.current_threshold`,
- labels starting with `B` are treated as phase outputs, with SP values interpreted in cycles and converted internally to radians, and
- any other label is treated as an amplitude-style comparison.

Example:

```
time     B1  B6  B7
2e-11    0   0   0
5e-11    0   1   0
1.5e-10  0   2   0
2e-10    1   2   0
```

9.2 `method = "csv_file"`

Use a CSV reference when you want Optimum to compare waveform behavior rather than exact checkpoint values. The first column is time. The remaining columns are numeric reference traces.

For CSV verification, Optimum extracts events from the reference and simulated traces, pairs them in time, and compares their behavior using the tolerance settings in `[verify]`. This is where `threshold`, `amplitude_threshold`, `slope_threshold`, `min_pulse_width`, `min_stable_samples`, `merge_gap_threshold`, and the `edge_*` settings matter most.

9.3 `method = "event"`

Use EVENT verification when the required behavior can be described directly from Josephson-junction phase events. Unlike SP and CSV verification, EVENT does not need a reference waveform or a table of expected event times. Instead, each `[[verify.rules]]` entry states what relationship must hold between named signals.

EVENT accepts bare junction labels such as `B1` and explicit phase expressions such as `P(B1)`. A configuration must contain at least one rule.

Rule options

The keys available inside a `[[verify.rules]]` table are:

Key	Type	Meaning
`type`	string	Rule type. One of `transfer`, `fanout`, `merge`, `delay`, `phase_shift`, `logic`, `clocked_logic`, or `clocked_transfer`.
`input`	string	Single input signal used by `transfer`, `fanout`, `delay`, `phase_shift`, and `clocked_transfer`.
`inputs`	string array	Multiple input signals used by `merge`, `logic`, and `clocked_logic`.
`clock`	string	Clock signal used by `clocked_logic` and `clocked_transfer`.
`output`	string	Single output signal used by every rule except `fanout`.
`outputs`	string array	Output branches used by `fanout`. At least one output is required.
`function`	string	Boolean function used by `logic` and `clocked_logic`.

Only the keys required by the selected rule are needed. Rule types, function names, and key names are written in lower case as shown below. There are no EVENT-specific timing thresholds to enter for normal use; event grouping and timing consistency use the simulation's numerical time resolution.

Supported rules

The rule set is intentionally small. Choose the rule that describes the cell behavior rather than looking for a rule named after a particular cell.

Rule	Required keys	Behavior
`transfer`	`input`, `output`	One input event is reproduced at one output. Event order is preserved and propagation delay must remain consistent.
`fanout`	`input`, `outputs`	One input event is reproduced on every listed output branch. Branch timing must remain consistent.
`merge`	`inputs`, `output`	Events from two or more inputs converge onto one output. Inputs that arrive within the numerical time resolution are treated as one logical arrival.
`delay`	`input`, `output`	The input is reproduced at the output with a stable delay. This has the same verification behavior as `transfer`, but identifies intentional delay as the purpose of the cell.
`phase_shift`	`input`, `output`	The input is reproduced at the output across a clock or excitation phase boundary. No user-entered phase number or golden delay is required.
`logic`	`function`, `inputs`, `output`	Event-triggered combinational logic. Near-coincident input events form one logical input group.
`clocked_logic`	`function`, `inputs`, `clock`, `output`	Combinational logic evaluated at each clock event using inputs received since the previous clock. Repeated events on one input within a clock interval count as one logical input.
`clocked_transfer`	`input`, `clock`, `output`	An input event is stored until the next clock. Several input pulses before the same clock represent one stored SFQ event and require one output event.

Logic functions

`logic` supports functions whose result is determined by an arriving event group:

`buffer`, `and`, `or`, `xor`, `majority`

`clocked_logic` supports the complete function set because the clock gives an explicit evaluation point, including intervals where no input event arrived:

`buffer`, `not`, `and`, `nand`, `or`, `nor`, `xor`, `xnor`, `majority`, `minority`

Input-count requirements are:

- `buffer` and `not`: exactly one input,
- `majority` and `minority`: an odd number of at least three inputs, and
- all other logic functions: at least two inputs.

Unclocked `logic` does not support `not`, `nand`, `nor`, `xnor`, or `minority`. Those functions can require an output when no input event occurs, so use `clocked_logic` when the result depends on an empty input interval.

Rule examples

A simple transfer check is shown below. The `delay` and `phase_shift` rules use the same `input` / `output` form with only the `type` value changed:

```toml
[[verify.rules]]
type = "transfer"
input = "B1"
output = "B2"
```

A fanout check uses an output array:

```toml
[[verify.rules]]
type = "fanout"
input = "B1"
outputs = ["B2", "B3", "B4"]
```

A merge check uses an input array:

```toml
[[verify.rules]]
type = "merge"
inputs = ["B1", "B2"]
output = "B3"
```

Unclocked logic adds a Boolean function:

```toml
[[verify.rules]]
type = "logic"
function = "xor"
inputs = ["B1", "B2"]
output = "B3"
```

Clocked logic adds the clock signal:

```toml
[[verify.rules]]
type = "clocked_logic"
function = "nand"
inputs = ["B1", "B2"]
clock = "B6"
output = "B7"
```

A clocked storage/transfer rule is:

```toml
[[verify.rules]]
type = "clocked_transfer"
input = "B1"
clock = "B6"
output = "B7"
```

Timing, polarity, and use with Optimum modes

EVENT verification checks event direction as well as event count and ordering. A reverse-polarity phase transition does not satisfy a positive event. Transfer-style rules establish propagation timing from the simulated behavior and require that timing to remain consistent; they do not require a golden delay value in the TOML file.

The same EVENT rules can be used with `verify`, `margin`, `yield`, and `optimize`. For margin searches, Optimum records the nominal logical behavior before parameter variation. Rule inputs must preserve their event count and polarity. For clocked rules, the clock stimulus must preserve its nominal event count, polarity, logical event slots, and cadence, and each input event must remain associated with the same capture clock as in the nominal design. A common propagation-latency shift at the observed clock junction is allowed when it does not change the clock cadence or logical slot. Configured outputs must continue to produce the nominal event cardinality, and every candidate must still satisfy the selected rule. This allows events to move within their intended logical window without requiring golden timestamps while rejecting cycle slips or changes in the applied stimulus.

EVENT currently operates on Josephson-junction phase events. Use SP or CSV verification when acceptance must be based on ordinary voltage or current levels rather than phase events.

Use `optimum-cli --event-config circuit.cir` when the available phase-signal labels are not already known. The helper lists the Josephson-junction phase signals in the circuit, asks for the rule type and required roles, and writes the corresponding `[[verify.rules]]` block. It does not infer signal roles, expected waveform timing, or expected pulse sequences.

9.4 Choosing tolerances

Start conservatively and tighten only after the nominal design is stable.

- For SP-style specifications, start with `threshold = 0.05` and only widen it when expected timing or amplitude differences are known and acceptable.
- For CSV verification, adjust one class of control at a time: first event detection, then edge-pairing tolerance, then overall acceptance threshold.
- If the circuit passes visually but Optimum misses an event, inspect `amplitude_threshold`, `min_pulse_width`, and `edge_min_amp_frac` before loosening the final pass/fail threshold.

10. Margin workflow

Margin finds the largest verified window around a scale factor of 1.0 for each selected parameter. The negative side searches `[min_search, 1.0]`; the positive side searches `[1.0, max_search]`.

Reported values are converted to percent around nominal, so a lower scale of `0.88` becomes a negative margin of `12%`, while an upper scale of `1.14` becomes a positive margin of `14%`.

10.1 Binary, exhaustive, and pocket search

- **Binary search** is the default and is the right first choice for most parameter sets.
- **Exhaustive search** steps through the interval in `accuracy` increments. Use it when you want direct control over the tested grid.
- **Pocket search** prints the pass/fail pattern across the searched interval. Use it when a circuit is not well described by one continuous margin on each side of nominal.

If parameter expressions are coupled, enable `parameter_dependence = true` so dependent parameter expressions are re-evaluated as the searched parameter moves.

10.2 Margin output

A margin run prints one line per selected parameter, followed by the critical and average margins:

```
IBIAS :  12.50[              ************|**************
      ]  14.00
L_IN  :   5.00[                    *****|*************
      ]  12.00
Critical margin: 5.00%   ['L_IN-']
Average margin: 10.75%
```

The critical margin is the worst side of the worst parameter. That is normally the first value to inspect.

11. Yield workflow

Yield draws random parameter sets from normal distributions centered on the chosen nominal values, verifies each draw, and reports the fraction that pass. Optimum uses a Gaussian model with standard deviation $\sigma = \text{spread}/3$, so the configured spread corresponds roughly to a $\pm 3\sigma$ range. Negative sampled values are clamped to zero.

11.1 Choosing the spread model

- Set `yield.spread` when you want a single fractional spread applied to all selected parameters.
- Leave `yield.spread = 0.0` when you want explicit parameter entries to provide spreads.
- If neither global nor per-parameter spread is supplied, Optimum falls back to `10%` of each nominal parameter value.

11.2 Roll-off runs

Set `roll_off = true` to sweep the spread from `step_size` to `1.0`. If `yield.output` is set, Optimum writes a two-column CSV with `Spread,Yield`.

11.3 Yield output

A standard yield run reports the number of passing and failing samples plus the resulting percentage:

```
Passes: 1932
Fails: 68
Yield: 96.6%
```

A roll-off run produces a table or CSV such as:

```
Spread,Yield
0.10,96.0
0.20,90.5
0.30,78.0
```

Open the roll-off CSV in the GUI when you want to inspect or export it alongside other result data.

12. Optimize workflow

Optimize uses Margin as its score function. It begins from the nominal design, repeatedly samples around the current parameter center, keeps the passing samples, forms a new candidate center, and accepts that candidate only when the critical margin improves.

12.1 Current optimization method

Optimum uses a center-of-gravity style search:

1. draw a Monte-Carlo cloud around the current center,
2. keep passing samples and also test reflected versions of failing samples,
3. compute a trimmed mean of the surviving cloud,
4. smooth that candidate using `ema_alpha`,
5. re-run Margin on the new candidate, and
6. keep it only if the critical margin improves.

12.2 Stopping conditions

The run stops when one of the following occurs:

- `patience` is exhausted,
- `converge` remains satisfied for `conv_k` consecutive iterations,
- `target` is reached, or
- `max_iterations` is reached.

12.3 Optimized netlist output

When `optimize.output` is set, Optimum writes a new circuit file with numeric `.PARAM` values back-annotated into the netlist. Parameters that were directly optimized are written with the new best values. Dependent parameters that need to be locked for reproducibility are written with baseline numeric values. Other lines are left unchanged.

The optimized output is a normal `.cir` file. Open it in the GUI to inspect parameter changes, visualize the netlist, or run a standard GUI simulation. If the optimized file references include files, keep it in the same project layout expected by the original circuit.

13. Worked DFF example

The installed `DFF_Example_config.toml` file illustrates all Optimum configuration blocks. It defaults to Optimize, but the commented mode lines make it easy to switch to Verify, Margin, or Yield.

```
#mode="verify"
#mode="margin"
#mode="yield"
mode="optimize"

[optimize]
method = "cog"
spread = 0.6
converge = 0.01
samples = 1000
max_iterations = 5000
output = "DFF_Example_optimized.cir"
sig_digits = 4

[parameters]
search_string="^(?!(?:B0$|B0Rs$))(?:B\\d+|IB\\d+|L\\d+)$"

[margin]
binary_search_steps = 7
output = "DFF_Example_margins.txt"

[yield]
spread = 0.1
num_samples = 2000
roll_off = true
output = "DFF_Example_yield_roll_off.csv"
step_size = 0.05

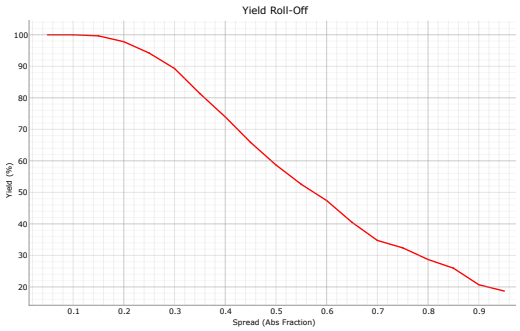
[verify]
method = "spec_file"
file = "DFF_Example.sp"
circuit = "DFF_Example.cir"
threshold=0.35
```

A margin run on this example produces output of this form:

```
Mode: Margin
B1 :  90.00[   ************************|************************
       ] 90.00
B2 :  90.00[   ************************|**********
       ] 37.27
B3 :  49.92[              **************|***********
       ] 38.67
B4 :  47.81[               *************|*******
       ] 24.61
B5 :  47.11[               *************|****
       ] 14.06
...
Critical margin: 14.06% ['B5+']
Average margin: 72.70%
```

The critical margin tells you which parameter side fails first and therefore where the design is weakest.

A yield roll-off on the same circuit produces a spread-versus-yield curve such as the one below:



An optimization run reports progress as it searches for a better critical margin:

```
Mode: Optimize
Original critical margin: B5+ 14.0625%
Iteration   20/5000 (Best critical margin:   37.27%      B4+)
      spread=1  samples=1000000
Convergence reached
Final critical margin: B4+ 37%
```

The exact numeric result is not deterministic unless the seed is controlled. Compare the final critical margin against the original nominal margin, then inspect the written optimized circuit when `optimize.output` is enabled.

GUI-to-CLI Equivalence

This chapter maps interactive GUI workflows to the command-line tools used for automation, batch runs, reproducibility checks, and Optimum. Use the GUI for interactive editing, visualization, simulation, trace analysis, and result export. Use `josim-pro-cli` for scripted simulation and native output generation. Use `optimum-cli` for TOML-driven verification, margin, yield, and optimization.

The GUI and CLI execute the same analyses and control blocks. `Run→Settings` supplies deck defaults; `Run→Add analysis blocks...` inserts `.CONTROL`, `.OPTIONS`, `.PRINT` and `.TRAN` commands. Save that netlist and pass it unchanged to `josim-pro-cli` to execute the same run plan. See Create multiple runs for the GUI workflow.

1. Simulation workflow equivalence

GUI workflow	CLI-oriented equivalent
Open a `.cir` file and run it in the active tab	`josim-pro-cli -o results.csv design.cir`
Open saved CSV/TSV result data for inspection	Generate result data with `josim-pro-cli -o results.csv design.cir`, then open the result data in the GUI
Save a netlist before running	Use a stable path as the positional input or with `-i`/`--input`
Run a netlist from standard input	`josim-pro-cli -i STDIN < design.cir`
Generate output that the GUI can import later	Prefer `CSV` or `TSV`-compatible text output; the GUI opens `.csv` and `.tsv` as result-data tabs
Use the GUI diagnostics/status area after a failed run	Read the terminal diagnostics and fix the first parse, license, runtime-option, or output-routing error before rerunning

The full command-line option reference is Chapter 9. Netlist syntax remains in Chapter 10.

2. Run Settings equivalence

GUI Run Settings control	CLI equivalent
Analysis: `Phase`	`josim-pro-cli -a 0 design.cir`
Analysis: `Voltage`	`josim-pro-cli -a 1 design.cir`
Integration: `BDF`	`josim-pro-cli -x 0 design.cir`
Integration: `TRAP`	`josim-pro-cli -x 1 design.cir`
Enable variable-step mode	`josim-pro-cli --varstep design.cir`; valid only for `PHASE + BDF`
Output precision	`josim-pro-cli -P 8 design.cir`, changing the integer value as needed
Compress text output	`josim-pro-cli -c -o results.csv design.cir`; use `CSV` or `DAT`, not `BIN`/`RAW`
Use deterministic seed	`josim-pro-cli -r 12345 design.cir`
Disable Josephson-junction Newton solving	`josim-pro-cli --no-newtonjj design.cir`

CLI overrides are run-local. If the same behavior is a design requirement, record it in the netlist with the appropriate `.OPTION` or simulation-control statement instead of relying only on a shell command.

3. Output, streaming, and import equivalence

GUI-side goal	CLI-oriented equivalent
Save result data for later GUI inspection	`josim-pro-cli -o results.csv design.cir`, then open `results.csv` in the GUI
Write primary output to the terminal	Omit `-o`/`--output`, or use `-o STDOUT`
Write a specific output file	`josim-pro-cli --output=results.csv design.cir` or `josim-pro-cli -o results.csv design.cir`
Split output groups into multiple files	Use netlist `.FILE` directives before the `.PRINT`, `.SAVE`, or `.PLOT` statements they should route
Stream a long text-output run	`josim-pro-cli -p -o results.csv design.cir`
Generate buffered binary or raw data	Keep streaming disabled: `josim-pro-cli -o results.bin design.cir` or `josim-pro-cli -o results.raw design.cir`
Compress text output	`josim-pro-cli -c -o results.csv design.cir`; the written file receives a `.gz` suffix when needed
Avoid output/input overwrite mistakes	Use distinct paths; the CLI rejects identical input and output file names

Streaming currently targets text workflows: `CSV`, `DAT`, and `STDOUT`. If a streamed target asks for `BIN`, `RAW`, an extensionless raw path, or another unsupported extension, the runtime falls back to CSV and reports the fallback.

4. Licensing equivalence

GUI licensing task	CLI-oriented equivalent
Generate a system identifier from `Help -> About -> Generate System ID...`	`josim-pro-cli -s`; this asks the installed `smlicensed` daemon to generate `SM_SysID.txt`
Set or correct a license-file path	`josim-pro-cli -l /path/to/sm_license.txt`; run this by itself to store the path
Check that the executable is present	`josim-pro-cli -v`; this prints version/build information and exits
Diagnose license-daemon problems	Confirm that the latest SUN Magnetics License Manager and `smlicensed` daemon are installed before debugging the GUI or CLI executable

Do not use `josim-pro-cli -l /path/to/sm_license.txt -v` to save a license path. The version option exits before a license-path update is performed.

5. Optimum handoff equivalence

Optimum is launched with `optimum-cli`. The GUI handoff is file based.

GUI-side step	Optimum CLI step
Create or edit the design netlist in the GUI	Save the netlist as a stable `.cir` file and reference it from `verify.circuit` in the TOML file
Keep related files together with `File -> Open Folder...`	Run `optimum-cli` from that project/examples directory when using relative output paths
Use the installed DFF example as an Optimum starting point	Run `optimum-cli DFF_Example_config.toml` from the installed examples directory
Switch between Verify, Margin, Yield, and Optimize	Edit only the TOML `mode` value: `"verify"`, `"margin"`, `"yield"`, or `"optimize"`
Inspect the optimized circuit	Open the generated `DFF_Example_optimized.cir` or other `optimize.output` file in the GUI
Inspect generated CSV/TSV data	Open the generated `.csv` or `.tsv` file in the GUI for waveform inspection, Trace Manager analysis, and plot export
Generate an Optimum-side system identifier	Use `optimum-cli -s`; this also delegates to the installed `smlicensed` daemon

For details on TOML keys, supported Optimum modes, installed DFF example files, and output handling, see Chapter 7.

CLI Usage and Command Reference

JoSIM®-Pro can be driven from the command line through `josim-pro-cli`. This chapter documents terminal workflows for automation, scripting, CI, batch simulation, reproducibility checks, and result generation for later GUI import. Interactive editing, visualization, first simulation, trace analysis, and plot export are covered in the GUI chapters.

9.1 When to use the CLI

Use `josim-pro-cli` when one of these requirements matters more than interactive inspection:

- repeatable batch runs from a shell script, build job, CI runner, or parameter sweep;
- producing `.csv` or `.tsv`-style text data for later import into the GUI or another analysis tool;
- streaming long text-output runs instead of buffering all output before writing;
- forcing a run-local analysis mode, integration method, precision, seed, or output destination without editing the netlist;
- checking CLI-only deployment behavior on systems where no desktop session is available.

For a GUI-to-terminal mapping of the same controls, see Chapter 8.

9.2 Command structure

The basic form is:

```
josim-pro-cli [options] [input_file]
```

The input netlist may be supplied in any one of the following ways:

- as a positional `input_file`;
- with `-i` or `--input`;
- as `STDIN`, usually with `-i STDIN` or shell redirection.

If both `-i` / `--input` and a positional input file are supplied, the explicit `-i` / `--input` value is the runtime input. If the program is started with no arguments at all, it prints version and usage information and exits instead of waiting on *STDIN*.

Keep durable circuit intent in the netlist itself with commands such as `.TRAN`, `.IC`, `.OPTION`, `.FILE`, and `.IV`. Use the CLI for per-run controls such as output destination, analysis-mode override, streaming, precision, verbosity, and reproducibility.

9.3 Default behavior

Unless overridden, JoSIM®-Pro behaves as follows:

- analysis mode defaults to `PHASE`;
- integration method defaults to `BDF`;
- variable-step mode is **OFF**;
- Josephson-junction Newton solving is **ON**;
- formatted output precision defaults to `6`;
- output is written to *STDOUT* unless an output target is supplied by `-o` / `--output` or by netlist `.FILE` routing;
- if `-o` / `--output` is provided without a value, the output file defaults to `out.csv`;
- the configured license path is read from `~/.josimpro/config.ini`, or from the built-in install default when no user configuration exists.

9.4 Quick command patterns

The commands below cover the most common operating patterns. Where a concrete input is useful, they use the installed `JTL_Example.cir` file from the examples directory.

Command	Purpose
`josim-pro-cli JTL_Example.cir`	Run a circuit with the default `PHASE + BDF` configuration and write results to *STDOUT*.
`josim-pro-cli -o results.csv JTL_Example.cir`	Run a circuit and write formatted CSV output to a file.
`josim-pro-cli --output=results.csv -i STDIN`	Read a netlist from *STDIN* and write CSV output to a named file.
`josim-pro-cli -a 1 -o results.dat JTL_Example.cir`	Override the analysis domain to `VOLTAGE` and write plain-text data output.
`josim-pro-cli -p -o streamed.csv JTL_Example.cir`	Stream a long text-output run directly to disk instead of buffering the full result set first.
`josim-pro-cli --varstep -o results.csv JTL_Example.cir`	Enable the supported variable-step path for `PHASE + BDF`.
`josim-pro-cli -r 123456 -o repeatable.csv JTL_Example.cir`	Reproduce spread- and noise-related random choices with a fixed seed.
`josim-pro-cli -l /path/to/sm_license.txt`	Store a license-file path in the user CLI configuration, then exit.
`josim-pro-cli -s`	Ask the installed `smlicensed` daemon to generate `SM_SysID.txt`.
`josim-pro-cli -h`	Print the built-in help summary.

9.5 Option summary

The full option surface is easier to scan when it is grouped by purpose rather than expanded one flag at a time.

9.5.1 Input, licensing, and information

Short	Long	Argument	Purpose
`-i`	`--input`	path or STDIN	Input netlist path. Use `STDIN` when piping or redirecting netlist text.
`-l`	`--license`	path	License-file path. The selected path is stored in the user configuration for future runs. When no input/output run is requested, the path is checked, saved, and the command exits.
`-s`	`--sysid`	none	Ask the installed SUN Magnetics License Manager `smlicensed` daemon to generate `SM_SysID.txt` in the current working directory for licensing support.
`-h`	`--help`	none	Print the built-in help summary.
`-v`	`--version`	none	Print product name, version, build timestamp, and git revision when available.

9.5.2 Analysis and runtime control

Short	Long	Argument	Purpose
`-a`	`--analysis`	0 or 1	Select the transient analysis domain: `0` for `PHASE`, `1` for `VOLTAGE`.
`-x`	`--integration`	0 or 1	Select the integration method: `0` for `BDF`, `1` for trapezoidal/ `TRAP`.
	`--varstep`	none	Enable the supported variable-step path. This is valid only for `PHASE + BDF`.
	`--no-newtonjj`	none	Disable Josephson-junction Newton solving for the run.
`-r`	`--seed`	unsigned integer	Set the random seed used for spread- and noise-related random choices. `0` means the runtime may use non-deterministic/time-based seeding.

9.5.3 Output and reporting

Short	Long	Argument	Purpose
`-o`	`--output`	optional target	Select the primary output target. Use a filename or `STDOUT`. If the flag is present without a value, the target is `out.csv`; if the flag is omitted, netlist `.FILE` routing or *STDOUT* is used.
`-p`	`--stream`	none	Stream text output as the run progresses instead of buffering the full result set first.
`-c`	`--compressed`	none	Store `CSV` or `DAT` output in a GZIP container. Compression is not applied to *STDOUT*, `BIN`, or `RAW` output.
`-P`	`--precision`	0 to 17	Set the number of digits used for formatted floating-point output.
`-m`	`--minimal`	none	Suppress the standard execution banner, run summaries, progress feedback, and other non-essential terminal text.
`-V`	`--verbose`	optional 0 to 3	Increase execution feedback. Use an explicit value, for example `-V 1`. A bare `-V` is accepted but resolves to the default value `0`.

9.6 Operational notes

9.6.1 Analysis and integration overrides

`PHASE` with `BDF` is the default and is the normal starting point for superconducting transient work. Use `VOLTAGE` when a voltage-domain solution is the better match for the circuit or for the quantities you intend to inspect directly.

```
josim-pro-cli -a 0 JTL_Example.cir
josim-pro-cli --analysis 1 JTL_Example.cir
josim-pro-cli -x 0 JTL_Example.cir
josim-pro-cli --integration 1 JTL_Example.cir
```

These CLI values are run-local overrides. They should not be used as a substitute for documenting durable design assumptions in the netlist.

9.6.2 Variable-step operation

Variable-step mode is useful when fast events are localized in time and a uniform fixed-step print cadence would oversample quiet regions. Keep fixed-step mode for uniformly sampled outputs, direct point-to-point comparison against earlier runs, or any run that must remain in `VOLTAGE` mode or use trapezoidal integration.

```
josim-pro-cli --varstep JTL_Example.cir
```

Constraint: variable-step mode is supported only for `PHASE + BDF`. Unsupported combinations are rejected explicitly.

9.6.3 Output targets and file formats

The output target determines the file format:

- `.csv` produces comma-separated text output;
- `.dat` produces space-separated text output;
- `.bin` produces binary output;
- `.raw` or an extensionless target produces raw output in buffered mode;
- `STDOUT` writes text output to the terminal using space-separated columns.

Unsupported extensions are normalized to CSV and the target extension is replaced with `.csv`. A `.gz` suffix is treated as a compression suffix: the logical format still comes from the preceding extension, for example `results.csv.gz` is CSV output.

Because `-o` / `--output` takes an optional value, prefer one of these unambiguous forms when piping from *STDIN*:

```
josim-pro-cli --output=results.csv -i STDIN
josim-pro-cli -i STDIN -o results.csv
```

A command such as `josim-pro-cli -o results.csv` is ambiguous to humans and is not a good pattern: use `--output=results.csv -i STDIN` when there is no positional input file.

9.6.4 Netlist `.FILE` routing

A netlist `.FILE` directive routes subsequent `.PRINT`, `.SAVE`, or `.PLOT` requests into the selected target. If no CLI `-o` / `--output` target is supplied, `.FILE` targets can define the native output plan by themselves.

Relative `.FILE` paths are resolved near the source that contains the directive, then against the input-file directory, then against the process working directory. This is important for included files: a `.FILE` inside an included file may route output relative to that included file's directory.

```
.FILE waveforms.csv
.PRINT V(q)
.FILE derived.dat
.PRINT P(j1)
```

Place `.FILE` before the output requests it is meant to route. It does not retroactively move earlier output requests.

9.6.5 Streaming output

Streaming is intended for `CSV`, `DAT`, and `STDOUT` workflows:

```
josim-pro-cli -p -o results.csv JTL_Example.cir
```

When streaming is enabled, selecting `.bin`, `.raw`, an extensionless raw target, or another unsupported extension falls back to streamed CSV output and emits a warning when fallback warnings are enabled. Use buffered output if a downstream tool specifically requires `BIN` or `RAW`.

9.6.6 Compression

Compressed output is valid for file-backed `CSV` and `DAT` output. When compression is enabled, JoSIM®-Pro ensures the written target has a `.gz` suffix.

```
josim-pro-cli -c -o results.csv JTL_Example.cir
josim-pro-cli --compressed --output=results.dat JTL_Example.cir
```

Do not request compression for `BIN` or `RAW`. Compression is ignored for *STDOUT*.

9.6.7 Reproducibility and randomness

Use a fixed seed when you need spread- and noise-related random decisions to be repeatable:

```
josim-pro-cli -r 123456 JTL_Example.cir
```

This is useful for side-by-side comparisons, regression tests, CI baselines, and Optimum-adjacent workflows where you want nominally identical random choices across repeated runs.

9.6.8 Licensing commands

Generate a system identifier from the terminal with:

```
josim-pro-cli -s
```

This delegates to the installed SUN Magnetics License Manager daemon, `smlicensed`. If the daemon is missing or outdated, `SM_SysID.txt` generation can fail even though the CLI executable itself is present.

To store a license path for future CLI runs, use `-l` / `--license` by itself:

```
josim-pro-cli -l /path/to/sm_license.txt
```

Do not combine that configuration-only form with `-v`; the version option exits before a simulation or license-path update is performed.

9.7 Common tasks

9.7.1 Run a normal fixed-step simulation

```
josim-pro-cli -o results.csv JTL_Example.cir
```

9.7.2 Run a voltage-domain simulation

```
josim-pro-cli -a 1 -P 12 -o results.dat JTL_Example.cir
```

9.7.3 Run a variable-step `PHASE + BDF` simulation

```
josim-pro-cli --varstep -o results.csv JTL_Example.cir
```

9.7.4 Stream a long run directly to disk

```
josim-pro-cli -p -o streamed.csv JTL_Example.cir
```

9.7.5 Read from *STDIN* and write to *STDOUT*

```
josim-pro-cli -i STDIN < JTL_Example.cir
```

9.7.6 Generate a CSV file for GUI import

```
josim-pro-cli -o results.csv design.cir
```

Then open `results.csv` in the GUI with `File -> Open...`, drag it into the main window, or select it from a folder opened with `File -> Open Folder...`.

9.8 Summary

Use the CLI to control how a specific run is executed and where its output goes. Keep the durable circuit intent in the netlist, then apply CLI overrides only for run-specific concerns such as output format, streaming, verbosity, and reproducibility. For interactive inspection, run setup, trace analysis, and plot export, return to the GUI workflow chapters.

Netlist Syntax Reference

This chapter is the quick-reference guide to the JoSIM®-Pro netlist language. Use it when you need the required form of an element, source, model, or control command, or when you want to confirm which optional attributes are accepted on a given line.

For workflow guidance such as when to choose phase or voltage operation, when variable-step mode is appropriate, or how streamed output behaves, see the Features chapter. For command-line overrides and concrete execution examples, see the Usage chapter.

Netlist anatomy

A typical JoSIM®-Pro netlist contains four kinds of content:

- element and source lines,
- model definitions,
- control commands, and
- the terminating `.END` statement.

A compact example is:

```
* Example Josephson junction circuit
.PARAM IBIAS=0.18m
B1 1 2 JJMOD area=1.2 spread=0.1 temp=4.2
L1 1 0 1e-9 spread=0.05
R1 2 0 50 spread=0.1 temp=4.2 neb=10GHz
I1 2 0 DC IBIAS
.MODEL JJMOD JJ(RN=10, CAP=1e-12)
.PRINT V(1) I(R1)
.TRAN 0.1ps 1ns
.END
```

Read the file from top to bottom:

1. declare any parameters and included files,
2. define the circuit elements and sources,
3. define any required models and output requests,
4. add the transient-analysis command and any runtime options, and
5. end the file with `.END`.

Quick syntax index

Use the tables below when you need to locate the right family of statement quickly before reading the detailed notes and examples later in the chapter.

Elements and models

Prefix / command	Basic form	Purpose
`R`	`Rname n+ n- value`	Resistor, with optional `spread=`, `temp=`, and `neb=` attributes.
`L`	`Lname n+ n- value`	Inductor, with optional `spread=`.
`C`	`Cname n+ n- value`	Capacitor, with optional `spread=`.
`B`	`Bname n+ n- model ...`	Josephson junction instance using a `JJ`, `RCSJ`, or `TJM` model.
`T`	`Tname n1+ n1- n2+ n2- ...`	Transmission line using the ideal `LOSSLESS`, finite `LCLADDER`, differential `RLGC`, frequency-dependent `WIDEBAND`, or two-mode `MODAL` model.
`@MTL`	`@MTL name P1=(...) P2=(...) ...`	JoSIM-Pro multiconductor RLGC matrix line with an explicit reference node.
`K`	`Kname L1 L2 k`	Mutual inductive coupling.
`.MODEL`	`.MODEL name JJ(...), TJM(...), WIDEBAND(...), or MTL(...)`	Junction or transmission-line model definition.

Sources and controlled sources

Prefix	Basic form	Purpose
`V`	`Vname n+ n- source type`	Independent voltage source.
`I`	`Iname n+ n- source type`	Independent current source.
`P`	`Pname n+ n- source type`	Independent phase source.
`F`	`Fname n+ n- cp cn gain`	Current-controlled current source.
`H`	`Hname n+ n- cp cn r`	Current-controlled voltage source.
`G`	`Gname n+ n- cp cn gm`	Voltage-controlled current source.
`E`	`Ename n+ n- cp cn gain`	Voltage-controlled voltage source.

Control commands

Command	Basic form	Purpose
.TRAN	.TRAN *tstep tstop* ...	Define the transient run window, print cadence, and startup flags.
.IC	.IC target=value	Apply explicit initial conditions to nodes or devices.
.PARAM	.PARAM *name=expr*	Define reusable parameter expressions.
.SUBCKT / X	.SUBCKT ... / X...	Define and instantiate hierarchical subcircuits.
.INCLUDE	.INCLUDE ”file”	Include another netlist file.
.TEMP / .NEB	.TEMP ... / .NEB ...	Set global thermal-noise temperature and bandwidth.
.SPREAD	.SPREAD ...	Set global or family-specific spread controls.
.PRINT / .SAVE / .PLOT	.PRINT ...	Request outputs and derived expressions.
.IV	.IV *model* [...]	Generate an IV sweep for a model.
.FILE	.FILE *path*	Route subsequent outputs to a named file.
.OPTION	.OPTION key=value ...	Set runtime, analysis, output, and adaptive-step controls in the netlist.
.END	.END	End the netlist.

Detailed reference

The remainder of this chapter provides the detailed forms, notes, and examples for each supported netlist statement.

Element reference

Resistor

Syntax:

```
Rname node1 node2 resistance [spread=spread] [temp=temp] [neb=freq]
```

Example:

```
R1 1 0 50 spread=0.1 temp=4.2 neb=10GHz
```

Inductor

Syntax:

```
Lname node1 node2 inductance [spread=spread]
```

Example:

```
L1 1 0 1e-9 spread=0.05
```

Capacitor

Syntax:

```
Cname node1 node2 capacitance [spread=spread]
```

Example:

```
C1 1 0 1e-12 spread=0.03
```

Josephson junction

Syntax:

```
Bname node1 node2 model [area=area] [spread=spread] [ic=ic] [temp=
      temp] [neb=freq] [icnoise=ICNOISE(...)]
```

Example:

```
B1 1 2 JJMOD area=1.2 spread=0.1 temp=4.2 icnoise=ICNOISE(0.01 1 0
      1p 10n 64)
.MODEL JJMOD JJ(RN=10, CAP=1e-12)
```

JoSIM®-Pro supports both `JJ` / `RCSJ` and `TJM` model definitions for Josephson junctions.

RCSJ/JJ Model Parameters

Parameter	Default	Description
VG or VGAP	2.8E-3	Junction gap voltage
IC or ICRIT	1E-3	Junction critical current before area scaling
RN	5	Junction normal resistance
R0	30	Junction subgap resistance
C or CAP	2.5E-12	Junction capacitance
T	4.2	Junction temperature in Kelvin; enables temperature-dependent RCSJ behaviour
TC	9.1	Critical temperature of the material; enables temperature-dependent RCSJ behaviour
RTYPE	1	Selects how the subgap branch is handled
DELV	0.1E-3	Transition voltage between subgap and normal regions
D	0.0	Transparency parameter affecting the phase relationship
ICFCT or ICFACT	$\pi/4$	Ratio of critical current to the transition step height
PHI	0	Phase offset, including π-junction style modelling
CPR	$\{1\}$	Harmonic amplitudes for the current-phase relationship

TJM Model Syntax

```
.MODEL TJ1 TJM(VG=2.8m IC=1m RN=5 CAP=2.5p FIT="fitparams.txt")
.MODEL TJ2 TJM(VG=2.8m IC=1m RN=5 CAP=2.5p FIT={0.1 0.2 0.3 0.4})
```

The `FIT` parameter can be supplied either inline or from an external text file. File paths are resolved relative to the including netlist location.

Area and Critical-Current Handling

If `area` is omitted but `ic=` is provided on the instance, JoSIM®-Pro derives the effective area from the requested critical current and the model's nominal `IC`. Instance area then scales the model quantities in the expected direction:

- `IC` and `CAP` scale with area,
- `RN` scales inversely with area,
- `R0` either follows `RN` or scales inversely with area depending on `RTYPE`.

Transmission line

A transmission line connects port `p1` / `n1` to port `p2` / `n2`. The `TYPE` attribute selects the model. Omitting `TYPE` selects `LOSSLESS`, preserving existing untyped `T` elements. New netlists should use the explicit `TYPE=` form. The legacy standalone `LOSSLESS` token remains accepted.

Scalar and two-mode line syntax

```
Tname p1 n1 p2 n2 TD=<time> Z0=<impedance>
Tname p1 n1 p2 n2 TD=<time> Z0=<impedance> TYPE=LOSSLESS
Tname p1 n1 p2 n2 TD=<time> Z0=<impedance> TYPE=LCLADDER NSEG=<
      integer>
Tname p1 n1 p2 n2 TYPE=RLGC RPERL=<ohm_per_length> \
       LPERL=<henry_per_length> GPERL=<siemens_per_length> \
       CPERL=<farad_per_length> LEN=<length> NSEG=<integer>
Tname p1 n1 p2 n2 TYPE=WIDEBAND MODEL=<model_name> \
       LEN=<length> NSEG=<integer>
Tname p1 n1 p2 n2 TYPE=MODAL ZDM=<impedance> TDDM=<time> \
       ZCM=<impedance> TDCM=<time> NSEG=<integer>
Tname p1 n1 p2 n2 <model-attributes> ICV=<voltage-state> ICI=<
      current-state>
```

Model	Required attributes	Use
`LOSSLESS`	`TD`, `Z0`	Ideal differential delay line. This is the default when `TYPE` is omitted.
`LCLADDER`	`TD`, `Z0`, `NSEG`	Finite lossless differential LC ladder. It models spatial dispersion and energy storage but no common-mode current path.
`RLGC`	`RPERL`, `LPERL`, `GPERL`, `CPERL`, `LEN`, `NSEG`	Uniform differential line with constant per-unit-length loss and storage values.
`WIDEBAND`	`MODEL`, `LEN`, `NSEG`	Uniform differential line with a named passive real-pole loss model.
`MODAL`	`ZDM`, `TDDM`, `ZCM`, `TDCM`, `NSEG`	Independent lossless differential and common-mode propagation for one conductor pair.

All required numeric values may be parameter expressions. Values must be finite and physically valid. `NSEG` must resolve to a positive integer and cannot change during parameter stepping because it controls generated topology.

Common attributes

Attribute	Applies to	Description
`TD`	`LOSSLESS`, `LCLADDER`	One-way propagation delay. Must be greater than zero.
`Z0`	`LOSSLESS`, `LCLADDER`	Differential characteristic impedance. Must be greater than zero.
`RPERL`	`RLGC`	Series resistance per chosen length unit. Must be greater than or equal to zero.
`LPERL`	`RLGC`	Series inductance per chosen length unit. Must be greater than zero.
`GPERL`	`RLGC`	Shunt conductance per chosen length unit. Must be greater than or equal to zero.
`CPERL`	`RLGC`	Shunt capacitance per chosen length unit. Must be greater than zero.
`LEN`	`RLGC`, `WIDEBAND`	Physical length in the same length unit used by the model values. Must be greater than zero.
`MODEL`	`WIDEBAND`	Name of a `.MODEL ... WIDEBAND(...)` definition.
`ZDM`, `TDDM`	`MODAL`	Differential-mode impedance and one-way delay.
`ZCM`, `TDCM`	`MODAL`	Common-mode impedance and one-way delay.
`NSEG`	Finite models	Number of generated sections. Required for `LCLADDER`, `RLGC`, `WIDEBAND`, and `MODAL`; invalid for `LOSSLESS`.
`ICV`, `ICI`	See initial state below	Paired explicit voltage and current state. Both must be present or both omitted.

Wideband model definition

```
.MODEL model_name WIDEBAND(RPERL=<ohm_per_length> \
      LPERL=<henry_per_length> GPERL=<siemens_per_length> \
      CPERL=<farad_per_length> RFREQ=(<frequency_hz> ...) \
      RDELTA=(<ohm_per_length> ...) GFREQ=(<frequency_hz> ...) \
      GDELTA=(<siemens_per_length> ...) FMAX=<frequency_hz>)
```

`RPERL`, `LPERL`, `GPERL`, and `CPERL` define the constant base model. `RFREQ` / `RDELTA` add frequency-dependent series-loss terms, and `GFREQ` / `GDELTA` add frequency-dependent shunt-loss terms. Each frequency list must have the same number of entries as its matching increment list. Frequencies must be strictly increasing and positive. Increments must be nonnegative. Each list may contain at most 64 entries.

`FMAX` declares the highest frequency for which the supplied fit is intended. It does not filter the simulated waveform. Results above `FMAX` are outside the declared model range. Omitting both dynamic term pairs reduces the model to the constant `RLGC` case.

JoSIM-Pro accepts an already fitted passive real-pole model. It does not import sampled network data, perform vector fitting, repair non-passive coefficients, or support complex poles, arbitrary rational multiports, or frequency-dependent multiconductor matrices.

Initial state and startup

The default `LOSSLESS` DC solution uses the zero-frequency differential relation

$$V(p1) - V(n1) = V(p2) - V(n2), \qquad I_1 = -I_2,$$

where both currents are defined as entering the line. This does not constrain common-mode voltage or create a common-mode current path. The solved state seeds transient history, so a line under constant DC bias does not begin with an artificial empty-line fill transient.

`UIC` skips the DC solve. `STARTUP` uses the existing startup procedure instead of a computed DC operating point. Use either option when an empty or externally initialized line is intended.

A scalar finite line or `LOSSLESS` line may declare a uniform state with paired `ICV` and `ICI` values. `LOSSLESS`, `LCLADDER`, and `RLGC` use one differential voltage and one current. `MODAL` uses two values in each vector:

```
T1 p1 n1 p2 n2 TYPE=MODAL ... ICV=(Vd Vc) ICI=(Id Ic)
```

Positive declared current flows from port 1 toward port 2. Explicit line state is supported with `ANALYSIS=VOLTAGE`. `WIDEBAND` does not accept `ICV` / `ICI`, because voltage and current alone do not define its internal relaxation state.

In fixed-step `LOSSLESS` runs, JoSIM-Pro inserts solve points for delayed discontinuities in addition to the nominal `.TRAN` grid. This improves edge timing but does not replace timestep-convergence checks. `LOSSLESS` remains fixed-step only.

Multiconductor RLGC matrix line

The JoSIM-Pro extension element `@MTL` represents a uniform coupled line with an explicit reference conductor.

```
@MTL MTL1 P1=(p1 p2 ... pM) P2=(q1 q2 ... qM) \
      REF=0 MODEL=MTLMODEL LEN=<length> NSEG=<integer> \
      [ICV=(v1 v2 ... vM) ICI=(i1 i2 ... iM)]

.MODEL MTLMODEL MTL(NCOND=M \
      R=(r11 r12 ... rMM) \
      L=(l11 l12 ... lMM) \
      G=(g11 g12 ... gMM) \
      C=(c11 c12 ... cMM))
```

`P1` and `P2` must contain exactly `NCOND` nodes. Their order defines the conductor order at both ports, in every matrix, and in current selectors. `REF` is the shared reference conductor. `LEN` is in metres. Matrix values are row-major and use SI units per metre:

- `R`: ohms per metre,
- `L`: henries per metre,
- `G`: siemens per metre,
- `C`: farads per metre.

Each matrix must contain exactly M^2 finite values. `R` and `G` must be symmetric positive semidefinite. `L` and `C` must be symmetric positive definite. Invalid dimensions, non-symmetric data, and non-passive data are rejected rather than repaired.

Optional paired `ICV` and `ICI` vectors define a uniform conductor-domain initial state. Each vector must contain exactly `NCOND` values. Voltages are relative to `REF`; positive currents flow from `P1` toward `P2`. The same `ANALYSIS=VOLTAGE` restriction used by scalar explicit line state applies.

Conductor currents are selected explicitly:

```
.PRINT I(MTL1[1])       * P1 conductor 1, entering the line
.PRINT I(MTL1[M])       * P1 conductor M, entering the line
.PRINT I(MTL1[M+1])     * P2 conductor 1, entering the line
.PRINT I(MTL1[2*M])     * P2 conductor M, entering the line
```

Use an integer in an actual output request. Bare `I(MTL1)` is rejected because it does not identify a conductor. Aggregate `POWER` and `ENERGY` selectors are not defined for `@MTL`.

Outputs

For a scalar or two-mode `T` line, `I(Tname)` is the current entering terminal `p1`, and `V(Tname)` is `V(p1)-V(n1)`. Either port voltage can also be requested from its external nodes, for example `V(p1,n1)` or `V(p2,n2)`. Generated ladder elements, transforms, and internal nodes are hidden from normal `.PRINT ALL` and `.SAVE ALL` output.

Solver support

Model	Fixed step	Variable step
`LOSSLESS`	Supported	Not supported.
`LCLADDER`	Supported	`PHASE + BDF`.
`RLGC`	Supported	`PHASE + BDF`.
`WIDEBAND`	`BDF` only	`PHASE + BDF`.
`MODAL`	Supported	`PHASE + BDF`.
`@MTL`	Supported	`PHASE + BDF`.

Variable-step operation also requires the normal global variable-step settings. Use a finite `STEPMAXDT`; an unbounded maximum timestep can under-resolve a finite line even when the adaptive error test passes.

Choosing `NSEG` and timestep

A finite ladder is dispersive, so delay error increases with frequency. For a lossless line with $N =$ `NSEG` and one-way delay T_D, the approximate ladder cutoff is

$$f_c = \frac{N}{\pi T_D}.$$

For angular frequency ω below cutoff, the approximate relative phase-delay and group-delay errors are

$$\epsilon_{\text{phase}} = \frac{\arcsin(x)}{x} - 1, \qquad \epsilon_{\text{group}} = \frac{1}{\sqrt{1-x^2}} - 1, \qquad x = \frac{\omega T_D}{2N}.$$

Use the larger section count required by the phase-delay and group-delay limits. For `MODAL`, apply the estimate to

both modal delays and use one `NSEG` that satisfies both modes.

For `RLGC` and `WIDEBAND`, an initial lossless transit estimate is

$$T_D \approx \texttt{LEN}\sqrt{\texttt{LPERL}\ \texttt{CPERL}}.$$

Loss changes attenuation and phase, so verify results at the frequencies of interest and repeat the simulation with a larger `NSEG`. For `@MTL`, the fastest supported mode controls spatial and timestep resolution.

Use at least ten timesteps per section as an initial limit:

$$\Delta t \leq \frac{T_{\text{section}}}{10}.$$

For `LCLADDER`, $T_{\text{section}} = T_D/NSEG$. For `RLGC` and `WIDEBAND`, use the lossless transit estimate divided by `NSEG`. For `MODAL`, use the shorter modal delay divided by `NSEG`. JoSIM-Pro warns when the configured fixed timestep or `STEPMAXDT` exceeds its estimate, but it does not change the timestep or section count.

Choose `NSEG` from the required bandwidth and delay-error target, then check timestep convergence and section-count convergence separately. Larger `NSEG` values increase matrix size, runtime, and memory use.

Model limits

- `LOSSLESS` is an ideal differential delay line and has no physical loss or common-mode path.
- `LCLADDER` is a finite lossless differential model and has no common-mode path.
- `RLGC` uses constant scalar per-unit-length values. It does not model skin effect, dielectric dispersion, geometry, or coupling to other lines.
- `WIDEBAND` supports fitted scalar real-pole loss terms only. It does not fit data or represent a frequency-dependent multiconductor matrix.
- `MODAL` represents two independent constant lossless modes. It does not model modal loss or mode conversion along the line.
- `@MTL` represents a uniform coupled line with constant RLGC matrices and an explicit reference conductor.

Examples

```
.PARAM ZLINE=4
.PARAM TLINE=2p
V1 source 0 PULSE(0 1m 1p 0.1p 0.1p 4p 20p)
RSOURCE source 1 ZLINE
T1 1 0 2 0 TD=TLINE Z0=ZLINE TYPE=LCLADDER NSEG=20
RLOAD 2 0 ZLINE
```

```
T1 1 0 2 0 TYPE=RLGC RPERL=0.8 LPERL=8p \
   GPERL=0.05 CPERL=0.5p LEN=1 NSEG=80
```

```
T1 p1 n1 p2 n2 TYPE=MODAL \
   ZDM=4 TDDM=2p ZCM=10 TDCM=6p NSEG=80
```

Mutual inductance

Syntax:

```
Kname inductor1 inductor2 coupling_factor
```

Example:

```
K1 L1 L2 0.99
```

Description: The signed coupling factor satisfies $-1 \leq k \leq 1$, with mutual inductance $M_{ij} = k_{ij}\sqrt{L_i L_j}$. Coupled inductances must be finite and nonnegative. Repeated declarations for the same pair add their mutual inductances.

Each connected group must be passive: its symmetric inductance matrix $\mathbf{L}$ must satisfy $\frac{1}{2}\mathbf{i}^T\mathbf{L}\mathbf{i} \geq 0$ for every current vector $\mathbf{i}$. Pairwise bounds alone do not guarantee this condition. Semidefinite groups, including consistent perfect coupling, are accepted with a scale-aware floating-point tolerance. Invalid groups report their inductors and coupling source locations. The same validation applies when parameters are updated.

Independent source reference

Voltage source

Syntax:

```
Vname node1 node2 source_type
```

Example:

```
V1 1 0 DC 1.0
```

Current source

Syntax:

```
Iname node1 node2 source_type
```

Example:

```
I1 1 0 PWL(0 0 1n 1)
```

Phase source

Syntax:

```
Pname node1 node2 source_type
```

Example:

```
.param TWO_PI=2*PI
P1 1 0 SIN(0 TWO_PI 1GHz)
```

Waveform and source functions

JoSIM®-Pro supports various types of functions that can be used as sources in circuits. Below are the waveform forms supported directly by the code.

Piece-wise Linear (PWL)

Generates a signal by linearly interpolating between the provided time–value pairs.

Syntax:

```
PWL(T1 A1 T2 A2 T3 A3 ... Tn An)
PWL(0 0 1n 1 2n 0)
```

Pulse

Generates a single or repeating pulse train between two amplitudes with a specified delay, rise time, fall time, pulse width, and period.

Syntax:

```
PULSE(A1 A2 [Td [Tr [Tf [Pw [Per]]]]])
PULSE(0 1 1n 0.1n 0.1n 2n 5n)
```

- A1 - Initial amplitude
- A2 - Final amplitude
- Td - Time delay before the first pulse
  Default: 0
- Tr - Rise time
  Default: Tstep
- Tf - Fall time
  Default: Tstep
- Pw - Pulse width
  Default: Tstop
- Per - Period
  Default: Tstop

Sinusoid

Creates a sinusoidal waveform with offset, amplitude, frequency, delay, and exponential damping.

Syntax:

```
SIN(A0 A [f [Td [theta]]])
SIN(0 1 1GHz 0 0)
```

- A0 - Offset
- A - Amplitude
- f - Frequency
  Default: 1/TStep
- Td - Time delay
  Default: 0
- theta - Exponential damping factor
  Default: 0

The resulting waveform is:

$$f(t) = A_0 + A\sin\left(2\pi f(t - T_d)\right)e^{-\theta(t-T_d)}$$

Custom Waveform

Loads a waveform from an external text file, with time step, scale factor, and interpolation control.

Syntax:

```
CUS(file.txt Ts Sf Im [Td [Per]])
CUS(waveform.txt 1n 1 0)
```

- file.txt - Path to a plain-text file containing amplitude samples
- Ts - Separation time between the samples in *file.txt*

Default: Tstep
- Sf - Scaling factor applied to the file samples
  Default: 1
- Im - Interpolation method:
  - 0 - None (step-like hold)
  - 1 - Linear
  - 2 - Cubic

  Default: 0
- Td - Start delay
  Default: 0
- Per - Repeat period
  Default: Tstop

Custom waveform file paths are resolved relative to the netlist or included file that references them.

DC

Provides a constant value. A bare scalar is also accepted anywhere a source function is expected.

Syntax:

```
DC A
DC 1.0
```

Noise

Generates sampled white-noise style excitation with a specified spectral amplitude and sample interval.

Syntax:

```
NOISE(A [Td [Ts]])
NOISE(1 1n 1p)
```

- A - Noise amplitude (spectral amplitude)
- Td - Time delay before noise starts
  Default: 0
- Ts - Noise sample time
  Default: Tstep

Power-Law Noise

Generates $1/f^{\alpha}$-style stochastic excitation.

Syntax:

```
PNOISE(A1 alpha [Td [dt [tcorr [N]]]])
PNOISE(1u 1 0 1p 10n 64)
```

- A1 - Amplitude spectral density at 1 Hz
- alpha - Power-law exponent
- Td - Start delay
- dt - Noise update interval
- tcorr - Correlation time used to set the low-frequency band edge
- N - Number of internal processes used to synthesize the spectrum

Flux-Noise Style Source

Generates $1/f^{\alpha}$-style source excitation referenced to Φ_0. This is primarily useful in phase or flux-oriented workflows.

Syntax:

```
FNOISE(Aphi0 alpha PHIDC [Td [dt [tcorr [N]]]])
FNOISE(1e-6 1 0 0 1p 10n 64)
```

- Aphi0 - Amplitude spectral density at 1 Hz, referenced to Φ_0
- alpha - Power-law exponent
- PHIDC - DC phase or flux offset, referenced to Φ_0

Exponential

Produces an exponentially rising and optionally falling waveform.

Syntax:

```
EXP(A1 A2 [Td1 [tau1 [Td2 [tau2]]]])
EXP(0 1 1n 1p 2n 1p)
```

The resulting waveform uses the standard decaying exponent form for both the rising and falling segments.

Piece-wise Sinusoidal

Generates a sinusoidal transition between each pair of successive time–value points.

Syntax:

```
PWS(T1 A1 T2 A2 T3 A3 ... Tn An)
PWS(0 0 10p 0 15p 1)
```

Stair

Generates a staircase waveform that increases by a fixed increment at a fixed interval, with optional delay, rise time, and initial offset.

Syntax

```
STAIR(A Per [Td [Tr [A0]]])
```

- A - Step increment
- Per - Step period
  Default: Tstep
- Td - Delay time
  Default: 0
- Tr - Rise time
  Default: Tstep
- A0 - Starting amplitude

Junction Critical-Current Noise Attribute

Josephson junction instances also support a dedicated `ICNOISE(...)` attribute for fractional critical-current noise:

```
B1 1 2 JJMOD ICNOISE=ICNOISE(0.01 1 0 1p 10n 64)
```

The argument structure matches `PNOISE(...)`.

Dependent source reference

Current-controlled current source (CCCS)

Syntax:

```
Fname node1 node2 control_pos control_neg gain
```

Creates a current source scaled by the specified gain and controlled by the current between the named control nodes.

Current-controlled voltage source (CCVS)

Syntax:

```
Hname node1 node2 control_pos control_neg transresistance
```

Creates a voltage source whose output is controlled by the current flowing between the named control nodes and scaled by the specified transresistance.

Voltage-controlled current source (VCCS)

Syntax:

```
Gname node1 node2 control_pos control_neg transconductance
```

Creates a current source whose output is controlled by the voltage across the named control nodes and scaled by the specified transconductance.

Voltage-controlled voltage source (VCVS)

Syntax:

```
Ename node1 node2 control_pos control_neg gain
```

Creates a voltage source scaled by the specified gain and controlled by the voltage across the named control nodes.

Control-command reference

Command names are case-insensitive and must match a supported command exactly. Inside a `.CONTROL` block, commands may omit the leading dot. Unsupported analysis or control commands stop the run with a source-location diagnostic before simulation output is written.

CLI analysis jobs and control blocks

The CLI executes each explicit `.TRAN` and `.IV` as a separate job, serially in source order after includes are expanded. Each job starts with fresh simulation state. An IV-only deck does not run a transient; its requested model can be defined without a main circuit. With no explicit analysis, an unscoped deck or a single unlabeled control block uses one default transient.

For several runs of one circuit with shared settings and traces, place several explicit transient commands at top level. Each command supplies its own timing:

```
I1 0 1 DC 1m
R1 1 0 1
.PRINT V(1)
.TRAN 1p 20p 0 2p
.TRAN 2p 20p 0 4p
.END
```

For a deck named `compare.cir`, the default files are `compare.tran-001.csv` and `compare.tran-002.csv`. Settings at top level are shared defaults, not sequential changes between runs.

For a named run with its own settings and outputs, put one analysis in each control block. The following complete deck runs the same circuit twice:

```
I1 0 1 DC 1m
R1 1 0 1
.CONTROL fine
  .OPTIONS integration=trap
  .TRAN 1p 20p 0 2p
  .FILE fine.csv
  .PRINT V(1)
.ENDC
.CONTROL coarse
  .OPTIONS integration=bdf
  .TRAN 2p 20p 0 4p
  .FILE coarse.csv
  .PRINT V(1)
.ENDC
.END
```

Here `fine` requests a 1 ps simulation step and 2 ps output cadence; `coarse` requests a 2 ps simulation step and 4 ps output cadence. The fourth `.TRAN` argument sets output cadence independently of the simulation step. Local `.PRINT` directives select each run's columns; local `.FILE` directives select its destinations.

Use one explicit analysis per block for this named-run pattern. The CLI also accepts several analyses in a block, producing one job per analysis with shared block settings, and settings-only blocks, which produce no job when the deck contains explicit analyses. A block is declarative: it is not an executable control script, and there are no loops, conditionals or sequential assignments.

Block labels start with an ASCII letter and contain only letters, digits, underscores or hyphens. Labels are unique ignoring case. Blocks must balance and cannot nest. Named or multiple blocks require an explicit analysis. Top-level settings apply to all jobs; block settings override them only in that block, regardless of their position relative to the analysis. Parameters, models and circuit declarations belong outside named or multi-analysis blocks. A legacy single-analysis unlabeled block retains its declaration handling.

A block inherits the complete top-level output-directive sequence unless it provides its own output directives, which replace that sequence. IV jobs use their own destination and voltage/current columns. Transient output directives in an IV-only scope are rejected. An explicit nonzero seed starts equivalent fresh random streams for each job. Automatic seeding selects one seed for the deck and reports the effective seed for each job.

A single transient retains its output name; a single IV defaults to `base.iv.csv`. Multiple transient jobs derive names such as `base.fine.tran-001.csv`; multiple IV jobs without paths use names such as `base.iv-002.csv`. The number is the position in the complete job sequence. Unlabeled blocks use ordinal names such as `block-1.tran-001`. For repeated transients, CLI `-o` supplies the base and format; without it, the input filename supplies the base and CSV is used. Explicit `.FILE` and IV paths are never automatically renamed. Only one job may write STDOUT.

The CLI checks all resolved file destinations before simulation, including format/compression changes, input files and temporary-file names. Collisions are errors. Files are staged until their job succeeds. A failure or cancellation stops the sequence and preserves earlier successful jobs. Existing files for a failed job remain intact unless some renames have already completed during a multi-file commit; the summary lists each file's outcome. Already-emitted STDOUT data cannot be recalled. Ctrl-C and termination requests cancel active CLI jobs cooperatively. Abrupt process termination and filesystem changes by other processes are outside this cleanup guarantee.

These deck-job rules apply to CLI execution, the explicit deck-job APIs and GUI execution. The low-level C++, C API and Python simulation operations retain their single-analysis behavior.

GUI and API analysis jobs

The GUI runs every analysis in the deck and provides a result selector showing each job ID and outcome. Several analyses in one block share that block's settings and traces, while each retains its own timing and results. Select a successful run to view its plots. Transient plots use time; IV plots use voltage on the horizontal axis and current on the vertical axis. Switching runs opens that run's source plots; plot edits and derived traces are not retained across run selection. Failed or cancelled runs have no selectable result, but earlier successful results remain available. Run details list file destinations and whether they were committed. Cancellation also stops an active IV analysis.

C++ clients use `AnalysisJobs(input, retainResults)`, Python clients use `AnalysisJobs(input, retain_results)`, and C clients use the `JPRO_AnalysisJobs_*` functions. Creation validates the complete plan; `run()` (C: `JPRO_AnalysisJobs_Run`) executes it once. Create a fresh plan for another execution. Existing low-level simulation calls do not execute a deck's other analyses.

Results expose job identity, block label, source, analysis kind, effective seed, integration method, transient timing, state and file publication outcomes. Inspect results before or after execution, including after an execution error; only cancellation may be requested concurrently. Planning errors do not return a plan. A plan snapshots its input settings and shares the supplied runtime's cancellation token. Clear a reused runtime's cancellation request before creating a new plan, or use a fresh runtime.

Retained results are optional and require buffered output. The GUI retains results for selection; clients can disable retention and use output files for streamed execution. Retention stores every successful run's visible series and therefore requires memory for their combined results. C++ and Python result snapshots and C output handles obtained from a plan remain valid after that plan is destroyed. For

IV, the retained axis (the C output “time vector”) is voltage in volts and the series is current in amperes; the analysis kind must be checked before interpreting the axis.

Transient analysis

Syntax:

```
.TRAN time_step stop_time [print_start_time] [print_step] [UIC] [
     STARTUP|DST]
```

Examples:

```
.TRAN 0.01ns 10ns
.TRAN 0.01ns 10ns 0 0.02ns UIC
.TRAN 0.01ns 10ns STARTUP
```

Each explicit command starts an independent CLI run; see the named-run examples on page 56.

The optional trailing flags may appear in any order:

- `UIC` — use the user-specified initial conditions directly instead of the default consistent start,
- `STARTUP` — enable legacy startup priming,
- `DST` — explicitly disable legacy startup priming.

If `print_step` is omitted, output is emitted at the accepted simulation steps. If it is provided, it must be positive.

Initial conditions are supplied separately with `.IC`. Supported forms include node-oriented aliases and explicit typed targets:

```
.IC n1=0
.IC V(n2)=0
.IC P(n3)=PI/2
.IC B1[2]=0
```

Parameter definition

Syntax:

```
.PARAM name=mathematical_expression
```

Example:

```
.PARAM Rval=50*cos(2*PI)
R1 1 0 Rval
```

Subcircuits

Syntax:

```
.SUBCKT name io_nodes
elements
.ENDS name
```

Example:

```
.SUBCKT amplifier 1 2 3
R1 1 2 100
L1 2 3 1e-9
.ENDS amplifier
```

Usage syntax:

```
Xname io_nodes [component_name=parameter]
```

Usage example:

```
X1 1 2 3 amplifier R1=400
```

Include files

Syntax:

```
.INCLUDE "filename"
```

Example:

```
.INCLUDE "common_components.cir"
```

Noise settings

Syntax:

```
.TEMP temperature_in_K
.NEB bandwidth_in_Hz
```

Example:

```
.TEMP 4.2
.NEB 1GHz
R1 1 0 50 temp=4.2 neb=1GHz
B1 1 2 JJMOD temp=4.2 neb=1GHz
```

These commands set the global temperature and effective noise bandwidth used when thermal noise is enabled on supported resistive devices and junctions. Individual resistors and Josephson junctions can override the global values with `temp=` and `neb=` attributes.

Parameter spread

Syntax:

```
.SPREAD [global_spread] [B=junction_spread] [C=capacitor_spread] [L=inductor_spread] [R=resistor_spread]
```

Examples:

```
.SPREAD 0.02
.SPREAD 0.02 L=0.10 C=0.05 R=0.10 B=0.20
R1 1 0 50 spread=0.05
```

The optional leading scalar applies a global spread. Family-specific overrides apply to Josephson junctions, capacitors, inductors, and resistors respectively. Individual devices can also specify `spread=` directly on the instance.

Output control

Syntax:

```
.PRINT type(device)
.SAVE type(node)
.PLOT type(node/device)
```

Example:

```
.PRINT V(1) I(R1)
.SAVE P(B1)
.PLOT NODEP B1
```

`.PRINT`, `.PLOT` and `.SAVE` perform the exact same function and can be used interchangeably.

Output Types

The following output types can be used within the **.PRINT**, **.SAVE** or **.PLOT** commands:

- **NODEV**: Nodal voltage between a specified node and ground, or between two nodes.
- **NODEP**: Nodal phase between a node and ground, or between two nodes.
- **DEVV**: Device voltage across a specific element, such as a resistor or Josephson junction.
- **DEVI**: Device current through an element.
- **DEVP**: Device phase associated with a particular element.
- **V()**: Voltage at a node (or between two nodes).
- **I()**: Current through a device.
- **P()**: Phase of a node or element.
- **W()**: Instantaneous apparent power of element. Only works on elements and not nodes.
- **E()**: Energy consumed/produced by element. Only works on elements and not nodes.

Example:

```
.PRINT V(1) I(R1) P(J1)
.PRINT NODEV(1) NODEP(2, 3)
.SAVE DEVV(R1) DEVI(L1) DEVP(J1)
.PRINT E(B01) W(IB01)
```

These commands allow users to monitor and analyze specific quantities in their circuit simulations, providing detailed insight into the behavior of both nodes and devices.

Subcircuit Output Referencing

Referencing the nodes or devices within a subcircuit requires unrolling the hierarchy of the subcircuit instantiations to the top level. In JoSIM®-Pro, subcircuits can be nested within each other, meaning that devices or nodes within a deeply nested subcircuit must be referenced using their complete hierarchical path.

Syntax for Subcircuit Referencing:

```
device.subckt1.subckt2.subcktN
```

Here, the device name refers to the component inside the innermost subcircuit, and each subcircuit name identifies the containing hierarchy as you move outward to the top-level netlist.

Example:

```
R1.X1.X2.X3
```

In this example:

- `R1` is a resistor inside the subcircuit `X1`.
- `X1` is instantiated inside another subcircuit `X2`.
- `X2` is instantiated within the subcircuit `X3`.
- `X3` is part of the top-level netlist.

To output the voltage across `R1`, you would use the following command:

```
.PRINT V(R1.X1.X2.X3)
```

Explanation: When JoSIM®-Pro unrolls the subcircuits during simulation, it treats each nested subcircuit as a hierarchical block. In order to refer to a specific device or node within such a hierarchy, you must fully specify the path from the innermost device to the top level. This ensures that the simulator correctly identifies the element within the nested structure.

More Examples: If you want to print the phase of a Josephson junction `B1` inside a deeply nested subcircuit, you would reference it like this:

```
.PRINT P(B1.X1.X2)
```

To save the current through an inductor `L2` inside the subcircuit `X4`:

```
.SAVE I(L2.X4)
```

Why This is Important: Proper subcircuit referencing ensures that the simulator can correctly locate the elements, especially when dealing with large and complex designs. Nested subcircuits allow modular design, but without clear referencing, it would be impossible to track specific elements within the hierarchy during the simulation. JoSIM®-Pro's ability to handle these references also makes it easier to debug and analyze simulations that involve reused or replicated components.

Expanded Nested Example:

```
.SUBCKT inner 1 2
R1 1 2 50
.ENDS inner

.SUBCKT middle 3 4
X1 3 4 inner
L1 3 4 1e-9
.ENDS middle

.SUBCKT outer 5 6
X2 5 6 middle
V1 5 0 DC 1
.ENDS outer

X3 1 2 outer
.TRAN 0.01ns 1ns
.END
```

In this example:

- `R1` is located inside the `inner` subcircuit.
- The `inner` subcircuit is instantiated as `X1` within the `middle` subcircuit.
- The `middle` subcircuit is instantiated as `X2` within the `outer` subcircuit.
- Finally, `outer` is instantiated as `X3` in the top-level netlist.

To print the voltage across `R1`, the correct reference would be:

```
.PRINT V(R1.X1.X2.X3)
```

This ensures that JoSIM®-Pro traces through the entire hierarchy to correctly locate `R1`.

Output Operations: Mathematical operations can now be performed on output traces with the same level of capability seen in `.PARAM` expression parsing. This enables users to extract complex outputs directly from the simulation without the need for post processing.

```
.PRINT V(R01) + V(R02)
.PLOT SIN(P(B01))
```

IV curve generation

Syntax:

```
.IV model_name [max_current] [filepath] [current_steps]
```

Example:

```
.IV JJMOD 300u example_iv.csv 400
```

This command generates a current-versus-voltage sweep for the named model to $\pm$ the requested maximum current, using the requested number of steps and storing the result at the requested file path. When omitted, the code supplies defaults for the maximum current, output path, and step count.

File output

Syntax:

```
.FILE filepath
```

Example:

```
.FILE results1.csv
.PRINT V(1) I(R1)
.FILE results2.csv
.PRINT P(J1)
```

The **.FILE** command allows for multiple output files. Each output command following a **.FILE** line stores results in the specified file.

The chosen file extension still determines the intended format for that target. When streaming output is enabled from the CLI, text targets are the intended path; requests for `BIN` or `RAW` output normalize to streamed `CSV` instead. See the Usage chapter for the CLI-side streaming controls.

Runtime and `.OPTION` controls

The `.OPTION` command is used to control simulation behavior directly from the netlist. This section documents the settings most relevant to everyday use, including analysis mode, integration method, output formatting, and variable-step controls.

General syntax:

```
.OPTION key=value [key=value ...]
```

General Runtime Controls

The following `.OPTION` controls are commonly used when configuring a run:

- `analysis=` phase, voltage, 0, or 1
- `integration=` BDF, TRAP, 0, or 1
- `seed=` *integer*
- `outputprecision=` *integer* or `precision=` *integer*
- `compressed=` ON, OFF, 1, or 0; alternatively `nocompress`
- `newtonjj=` ON, OFF, 1, or 0
- `newtonmaxiters=` *integer*
- `newtonreltolphase=` *value*
- `newtonabstolphase=` *value*
- `newtondamping=` *value*
- `variablestep=` ON, OFF, 1, or 0

Examples:

```
.OPTION analysis=PHASE integration=BDF
.OPTION seed=123456 outputprecision=12 compressed=ON
.OPTION newtonjj=ON newtonmaxiters=20 newtondamping=0.5
.OPTION variablestep=ON
```

Practical selection: `PHASE` is the default and is the mode to use when a workflow depends on Josephson phase evolution or variable-step operation. `VOLTAGE` remains available for voltage-domain workflows. `BDF` is the default integration method and is also the required method for variable-step runs, while `TRAP` remains available for fixed-step use.

Curated Adaptive-Step Surface

The following adaptive-step controls are the main settings users are most likely to adjust when tuning variable-step behaviour:

- `stepreltolphase`, `stepabstolphase`
- `stepreltoldphi`, `stepabstoldphi`
- `stepreltoldphidt`, `stepabstoldphidt`, `stepltedphidtweight`
- `stepreltoljjvolt`, `stepabstoljjvolt`, `stepltejjvoltweight`
- `stepsafetyfactor`, `stepmaxgrow`, `stepminshrink`
- `stepgrowththreshold`, `stepgoodstepsbeforegrow`
- `stepmindt`, `stepmaxdt`, `stepstartupdt`
- `stepcapstartupbreakpoint`, `stepstartupbreakpointfrac`
- `stepcapjjphaseadvance`, `stepjjetaphi`
- `stepcapjjvgapproach`, `stepjjvgapproachfrac`, `stepjjvgapproachsafety`

Example:

```
.OPTION variablestep=ON integration=BDF analysis=PHASE
.OPTION stepreltolphase=1e-3 stepabstolphase=1e-6
.OPTION stepmindt=1e-15 stepmaxdt=1e-12 stepstartupdt=5e-15
.OPTION stepcapjjphaseadvance=ON stepjjvgapproachfrac=0.85
```

Constraints and Invalid Combinations

Variable-step mode is only supported for `PHASE + BDF`. If `variablestep=ON` is combined with a different analysis type or integration method, JoSIM®-Pro rejects the run explicitly.

CLI overrides still take precedence when the corresponding CLI override was set for the run. For example, `--varstep` or `--no-newtonjj` on the command line overrides the same setting in the netlist.

Additional advanced variable-step controls are also available for breakpoint fencing, local growth clamps, BDF1 restart behaviour, oscillation recovery, JJ-activity gating, and startup holdoffs. These controls are intended for specialised tuning rather than everyday use, but they are part of the supported `.OPTION` surface.

Recommended Practice

Use `.OPTION` to keep simulation intent close to the netlist, and use the CLI for per-run overrides such as output location, verbosity, and licensing. In most cases, it is best to start with the default adaptive-step settings and only tune them when a specific circuit requires it. For exact command-line forms of `--varstep`, output streaming, and analysis selec-

tion, see the Usage chapter.

Built-in constants

Constant	Symbol	Value
PI	π	3.141592653589793
PHI_ZERO	Φ_0	2.067833831170082E-15
BOLTZMANN	k_B	1.38064852E-23
EV	e	1.6021766208E-19
HBAR	$\hbar$	1.0545718001391127E-34
C	c	299792458
MU0	μ_0	12.566370614E-7
EPS0	ϵ_0	8.854187817E-12
RFQ	$\frac{\Phi_0}{2\pi}$	3.291059757E-16

Summary

This chapter provided a detailed explanation of the JoSIM®-Pro netlist syntax, including component definitions, dependent sources, model parameters, and built-in constants.

Worked Netlist Examples

This chapter presents practical example workflows for JoSIM®-Pro. Each example is organized around a task rather than around a standalone netlist dump. For every workflow, the goal is the same: show what problem the example solves, what command to run, what output to inspect, and what behavior to expect from the results.

Example paths in this chapter have two explicit scopes:

- **Installed examples** are files that the package installs under its `examples` directory. Use these for first-use GUI work, quick CLI validation, and Optimum handoff.
- **Manual-only snippets** are small self-contained netlists printed in this chapter. They are not installed as named files; copy them into your own working directory when you want to reproduce the workflow.

The installed examples directory contains the following files:

Installed file	Primary use
`JTL_Example.cir`	First simulation and quick CLI validation.
`AND_Example.cir`	Installed RSFQ AND-gate example; requires the installed `base_cells.cir` in the same directory.
`Splitter_Example.cir`	Installed RSFQ splitter example.
`DFF_Example.cir`	Installed DFF circuit used by the Optimum example.
`DFF_Example.sp`	Installed DFF verification specification used by Optimum.
`DFF_Example_config.toml`	Installed Optimum configuration for the DFF example.
`base_cells.cir`	Installed reusable cell library used by `AND_Example.cir`.

Additional example files outside this installed list are not assumed to be present after installation. Some are large stress designs, generated netlists, or environment-specific examples. They are therefore not used by this manual's guided workflows.

The examples below are intentionally small enough to adapt into your own netlists. For larger superconducting logic cells and system-level designs, the same workflow patterns apply: build the hierarchy, choose the analysis mode and stepping method, select the outputs you need, and run the simulation in the format that best fits the downstream toolchain. The chapter closes with installed full logic-cell examples.

1. First transient run and result inspection

Problem this solves: You want to confirm that a simple transient simulation runs correctly and that you understand how to read the resulting waveform file.

Example netlist (save as `first_run.cir`):

```
* first_run.cir
.MODEL JJMOD JJ(IC=200u RN=15 R0=30 CAP=0.2p)
VIN in 0 PULSE(0 830u 20ps 2ps 2ps 8ps 40ps)
L1 in j 2p
B1 j 0 JJMOD
R1 j 0 2

.TRAN 0.25ps 120ps
.PRINT V(in) V(j) I(B1) P(B1)
.END
```

Command to run:

```
josim-pro-cli -o first_run.csv first_run.cir
```

What to inspect:

- the time column in `first_run.csv`
- the input drive `V(in)`
- the junction node response `V(j)`
- the junction current and phase: `I(B1)` and `P(B1)`

What to expect:

- The CSV file will contain the requested output traces together with the simulation time points.
- The input pulse at `V(in)` drives the junction branch through `L1`.
- The phase and current traces provide a quick way to see whether the junction remains in a quiet operating region or switches dynamically during the transient.

Why this example matters: This is the smallest useful end-to-end workflow in JoSIM®-Pro: define a model, drive a circuit, choose outputs, run the simulator, and inspect the output file.

---

2. Hierarchical design with subcircuits and included files

Problem this solves: You want to organize reusable building blocks in separate files and still reference internal devices cleanly in your output requests.

Library file (save as `base_cells.cir`):

```
* base_cells.cir
.SUBCKT jjbranch IN OUT
.PARAM RSH=5
BJJ IN N1 JJMOD IC=200u
RJJ IN N2 RSH
LRJJ N2 N1 0.1p
LRET N1 OUT 0.1p
.MODEL JJMOD JJ(RN=15 R0=30 CAP=0.1p)
.ENDS
```

Top-level netlist (save as `hierarchy_run.cir`):

```
* hierarchy_run.cir
.include base_cells.cir

.SUBCKT delay_stage IN OUT
L1 IN N0 2p
X1 N0 OUT jjbranch
.ENDS

VIN in 0 PULSE(0 830u 20ps 2ps 2ps 8ps 50ps)
XSTAGE1 in mid delay_stage
XSTAGE2 mid out delay_stage
RLOAD out 0 2

.TRAN 0.25ps 150ps
.PRINT V(in) V(mid) V(out)
.PRINT P(BJJ.X1.XSTAGE1) P(BJJ.X1.XSTAGE2)
.END
```

Command to run:

```
josim-pro-cli -o hierarchy_run.csv hierarchy_run.cir
```

What to inspect:

- the stage-to-stage propagation from `V(in)` to `V(mid)` to `V(out)`
- the hierarchical phase traces `P(BJJ.X1.XSTAGE1)` and `P(BJJ.X1.XSTAGE2)`

What to expect:

- The design is built from a reusable cell file and a higher-level subcircuit wrapper.
- The output file shows the delay through each stage.
- The fully qualified names demonstrate how internal devices are referenced once the hierarchy is unrolled.

Why this example matters: Real JoSIM®-Pro designs are rarely single flat netlists. This workflow shows how to split designs into reusable files while still inspecting internal activity when debugging or characterizing a larger circuit.

---

3. Device and model characterization with `.IV`

Problem this solves: You want to characterize a junction model directly instead of embedding it first inside a larger circuit.

Example netlist (save as `iv_run.cir`):

```
* iv_run.cir
.MODEL JJMOD JJ(IC=200u RN=15 R0=30 CAP=0.2p)
.IV JJMOD 1m jjmod_iv.csv 200
.END
```

Command to run:

```
josim-pro-cli iv_run.cir
```

What to inspect:

- the generated file `jjmod_iv.csv`
- the voltage-current sweep shape for the model under test

What to expect:

- JoSIM®-Pro generates a helper circuit internally and writes the IV data to the requested file.
- The sweep covers positive and negative current excursions and returns through zero, which is useful when checking symmetry and transition behavior.
- The `steps` argument controls the resolution of the generated curve.

Why this example matters: `.IV` is the quickest way to validate that a model has the electrical behavior you expect before using it in a larger circuit or optimization flow.

---

4. Noise, spread, and reproducibility with a fixed seed

Problem this solves: You want to introduce randomized variation and noise into a run while still being able to reproduce the same realization later.

Example netlist (save as `noise_seeded.cir`):

```
* noise_seeded.cir
.OPTION seed=123456
.SPREAD 0.05 R=0.02 B=0.03
.MODEL JJMOD JJ(IC=180u RN=12 R0=25 CAP=0.15p)

IBIAS j 0 DC 250u
B1 j 0 JJMOD icnoise=ICNOISE(0.02 1 20ps 1ps 200ps 64)
R1 j 0 2 temp=4.2 neb=20GHz

.TRAN 0.25ps 150ps
.PRINT V(j) I(B1) I(IBIAS)
.END
```

Command to run:

```
josim-pro-cli -o noise_seeded.csv noise_seeded.cir
```

What to inspect:

- the waveform file from one run to the next
- the bias point and current fluctuations in `V(j)` and `I(B1)`

What to expect:

- With the same seed, the same randomized spread realization and noise sequence are reproduced.
- Changing the seed produces a different realization while keeping the same nominal circuit.
- The resistor contributes thermal-noise behavior through `temp` and `neb`, while the junction uses an explicit `ICNOISE(...)` model for critical-current variation in time.

Practical note: When you want deterministic reruns from the command line, `-r` can be used to override the netlist seed for a particular run. The same workflow also applies to explicit source-side noise functions such as `NOISE(...)`, `PNOISE(...)`, or `FNOISE(...)` when those are more appropriate for the quantity being injected.

---

5. Variable-step and streamed output for larger runs

Problem this solves: You want to run a longer transient efficiently and avoid buffering a large result set entirely in memory.

Example netlist (save as `long_run.cir`):

```
* long_run.cir
.OPTION analysis=PHASE integration=BDF variablestep=ON
.MODEL JJMOD JJ(IC=200u RN=15 R0=30 CAP=0.2p)

VIN in 0 PULSE(0 830u 20ps 2ps 2ps 10ps 200ps)
L1 in j 2p
B1 j 0 JJMOD
R1 j 0 2

.TRAN 0.25ps 2ns
.PRINT V(in) V(j) I(B1)
.END
```

Command to run:

```
josim-pro-cli --varstep -p -o long_run.csv long_run.cir
```

What to inspect:

- the spacing of the time points in `long_run.csv`
- file generation during the run rather than only at the end

What to expect:

- Variable-step mode uses non-uniform time steps, so adjacent rows in the output file do not need to be evenly spaced in time.
- Streaming writes text output progressively, which is useful for longer runs and large waveform sets.
- This workflow is only valid for the supported `PHASE + BDF` combination.

Practical note: For streamed output, a text format such as CSV is the safest choice. When you need binary or raw output, use the normal buffered output path instead of streaming.

---

6. Derived outputs and multiple output files

Problem this solves: You want JoSIM®-Pro to write both direct waveforms and derived quantities without relying on a separate post-processing script.

Example netlist (save as `derived_outputs.cir`):

```
* derived_outputs.cir
.MODEL JJMOD JJ(IC=200u RN=15 R0=30 CAP=0.2p)
VIN in 0 PULSE(0 830u 20ps 2ps 2ps 8ps 40ps)
L1 in j 2p
B1 j 0 JJMOD
R1 j 0 2

.TRAN 0.25ps 120ps

.FILE waveforms.csv
.PRINT V(in) V(j) I(B1)

.FILE derived.csv
.PRINT V(j)-V(in) SIN(P(B1)) E(B1) W(R1)
.END
```

Command to run:

```
josim-pro-cli derived_outputs.cir
```

What to inspect:

- `waveforms.csv` for the direct simulation traces
- `derived.csv` for computed expressions and device-derived quantities

What to expect:

- The `.FILE` directive routes subsequent output requests into the selected file.
- Derived expressions such as `V(j)-V(in)` and `SIN(P(B1))` are evaluated during output generation.
- Energy and power style outputs such as `E(B1)` and `W(R1)` make it easier to inspect device behavior without exporting every raw trace first.

Why this example matters: Output expressions and multiple output files reduce friction in downstream analysis. They are especially useful when different consumers need different result sets, such as plotting tools, scripting workflows, or spreadsheet review.

Full logic-cell examples

The workflow examples above are compact and task-oriented. The three full logic-cell examples below are installed with the package. Open them from the installed `examples` directory in the GUI, or run the listed CLI commands from that same directory. Together they show both focused simulation tasks and complete cell-level designs.

---

7. Josephson Transmission Line (JTL)

Installed file: `JTL_Example.cir`. Open it from the installed examples directory with `File -> Open...`, or run it from that directory with:

```
josim-pro-cli -o JTL_Example.csv JTL_Example.cir
```

The listing below shows the simulation-relevant content. The installed file may also contain `SCHM_<design>` comment blocks that preserve GUI visualizer layout; those comments do not change the circuit semantics.

```
* Josephson Transmission Line (JTL)
.SUBCKT jjbranch IN
.PARAM RSHUNT=5
BJJ     IN  1   JJMOD IC=200u
RJJ     IN  2   RSHUNT
LRJJ    2   1   0.1p
LJJP    1   0   0.1p
.MODEL JJMOD JJ(RN=15, CAP=0.1p)
.ENDS

.SUBCKT ibias OUT
.PARAM IBIAS=300u
IBIAS   0   1   DC IBIAS
LBIAS   1   OUT 3E-13
.ENDS

.SUBCKT jtl IN OUT
.PARAM LSTORE=2.0p
L1      IN  1   LSTORE
X1      1       jjbranch
L2      1   2   LSTORE
X2      2       ibias
L3      2   3   LSTORE
X3      3       jjbranch
L4      3   OUT LSTORE
.ENDS

VIN     1   0   PULSE(0 830u 50p 2.5p 2.5ps 0 50ps)
XJTL    1   2   jtl
ROUT    2   0   2

.TRAN 0.25ps 250ps
.PLOT V(VIN) P(BJJ.X1.XJTL) P(BJJ.X3.XJTL) V(ROUT)
.END
```

Description: The Josephson Transmission Line (JTL) is a key building block in superconducting circuits, allowing the efficient propagation of flux quanta. This example demonstrates a more advanced JTL design with shunt resistors, inductors, bias currents, and multiple Josephson junctions.

Explanation: This advanced JTL design is composed of the following key elements:

- **Josephson Branch (`jjbranch`):**

  - A Josephson junction (`BJJ`) modeled by `JJMOD` .
  - A shunt resistor (`RJJ`) to maintain stability.
  - Two inductors, `LRJJ` and `LJJP` , forming part of the branch.
- **Bias Current Subcircuit (`ibias`)**:
  - Provides a steady bias current to the circuit using an ideal current source and an inductor.
- **JTL Subcircuit (`jtl`)**:
  - Multiple `jjbranch` and `ibias` subcircuits are connected with inductors (`L1` , `L2` , `L3` , `L4`) to propagate flux quanta through the line.
- **Top-Level Netlist**:
  - A pulse source (`VIN`) drives the input.
  - The output voltage is monitored across the resistor `ROUT` .

Key Simulation Insights:

- The `.TRAN` command defines a transient simulation with a step size of `0.25ps` and a total duration of `250ps` .
- The `.PLOT` command outputs:
- The input voltage at `VIN` .
- The phase across the Josephson junctions `BJJ` in the first and third branches of the JTL.
- The output voltage across the load resistor `ROUT` .

Why This Example is Useful:

- This example illustrates the construction of a superconducting transmission line using multiple Josephson junctions and bias currents.
- It demonstrates how to organize complex circuits into hierarchical subcircuits.
- It shows how to reference devices within nested subcircuits using full paths such as `BJJ.X1.XJTL` .

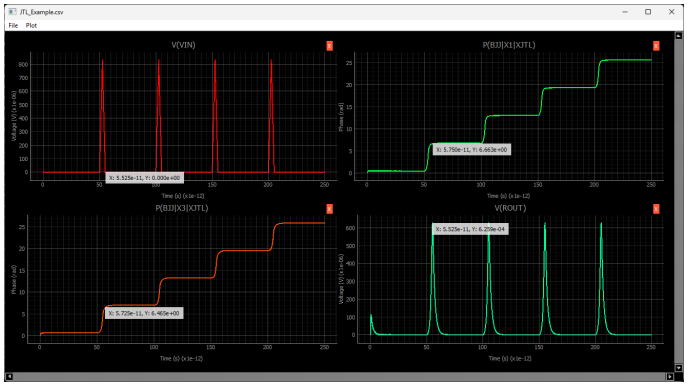


Figure 11.1: JTL example output traces from the installed example

---

8. RSFQ Splitter Cell

Installed file: `Splitter_Example.cir` . Open it from the installed examples directory with `File -> Open...` , or run it from that directory with:

```
josim-pro-cli -o Splitter_Example.csv Splitter_Example.cir
```

```
* RSFQ Splitter Cell
.SUBCKT jjbranch IN
.PARAM Rshunt=5
BJJ     IN  1   JJMOD IC=200u
RJJ     IN  2   Rshunt
LRJJ    2   1   0.1p
LJJP    1   0   0.1p
.MODEL JJMOD JJ(RN=15, CAP=0.1p)
.ENDS

.SUBCKT ibias OUT
.PARAM IBias=300u
IBias   0   1   DC Ibias
LBias   1   OUT 3E-13
.ENDS

.SUBCKT splitter a Q0 Q1
LA      a   6   0.3E-12
XJJ1    6       jjbranch    BJJ=100u
XBIAS1  6       ibias       IBIAS=125u
L1      6   7   1.5E-12
XJJ2    7       jjbranch    BJJ=175u
L2      7   18  3E-12
XBIAS2  18      ibias       IBIAS=350u
L3      18  19  0.5E-12
L4      4   19  1.3E-12
XJJ3    4       jjbranch    BJJ=125u
XBIAS3  5       ibias       IBIAS=75u
L5      4   5   1.3E-12
XJJ4    5       jjbranch    BJJ=175u
LQ0     5   q0  2.2E-12
L6      19  8   1.3E-12
XJJ5    8       jjbranch    BJJ=125u
XBIAS4  9       ibias       IBIAS=75u
L7      8   9   1.3E-12
XJJ6    9       jjbranch    BJJ=175u
LQ1     9   q1  2.2E-12
```

```
.ENDS

VIN     a 0  PULSE(0 830u 50p 2.5p 2.5ps 0 50ps)
Xsplit  a Q0 Q1 splitter
R1      Q0 0 2
R2      Q1 0 2

.TRAN 0.25ps 250ps
.PLOT V(VIN) V(R1) V(R2)
.END
```

Description: This example demonstrates a splitter cell based on RSFQ technology, which splits the input signal into two outputs. The circuit uses the same **Josephson Junction branch** (`jjbranch`) and **bias current subcircuit** (`ibias`) in multiple parts of the design, allowing for efficient parameterization and reusability.

Explanation: This RSFQ splitter design is composed of the following key elements:

- **Josephson Branch (`jjbranch`):**
  - A Josephson junction (`BJJ`) modeled by `JJMOD` .
  - A shunt resistor (`RJJ`) for stability.
  - Two inductors, `LRJJ` and `LJJP` , forming the Josephson branch.
- **Bias Current Subcircuit (`ibias`):**
  - Provides steady bias current to the circuit using an ideal current source and an inductor.
- **Splitter Subcircuit (`splitter`):**
  - This subcircuit uses multiple instances of `jjbranch` and `ibias` subcircuits to build a splitter that routes the input signal `a` into two outputs `Q0` and `Q1` .
  - The inductors `LA` , `L1` , `L2` , `L3` , and `LQ0` , `LQ1` form the core of the splitting operation.
- **Top-Level Netlist**:
  - A pulse source (`VIN`) drives the input.
  - The output voltages are monitored across resistors `R1` and `R2` .

Key Simulation Insights:

- The `.TRAN` command defines a transient simulation with a step size of `0.25ps` and a total duration of `250ps` .
- The `.PLOT` command outputs:
  - The input voltage at `VIN` .
  - The output voltage at `R1` and `R2` , corresponding to `Q0` and `Q1` .

Why This Example is Useful:

- This example illustrates how to efficiently reuse subcircuits by parameterizing them (e.g., modifying Josephson junction critical current `BJJ` and bias current `IBIAS` for different instances).
- It demonstrates how to organize complex circuits into hierarchical subcircuits, simplifying the design.
- It shows how to split a signal in RSFQ logic, a fundamental operation in superconducting digital circuits.

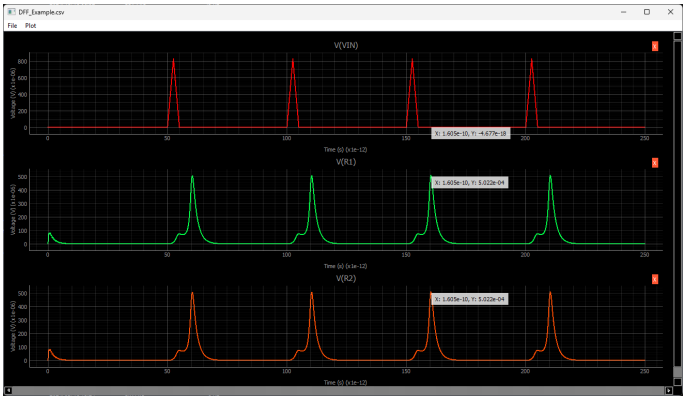


Figure 11.2: Splitter example output traces from the installed example

---

9. RSFQ AND Gate

Installed files: `AND_Example.cir` and `base_cells.cir` . Keep both files in the same directory, because the AND netlist includes `base_cells.cir` . Open `AND_Example.cir` from the installed examples directory with `File -> Open...` , or run it from that directory with:

```
josim-pro-cli -o AND_Example.csv AND_Example.cir
```

```
* RSFQ AND Cell
.include base_cells.cir

.subckt and a b clk q
LA      a   1   2p
XJJ1    1       jjbranch
XBIAS1  1       ibias       IBIAS=175u
L1      1   2   3p
B1      2   4   JJMOD       IC=180u
RB1     2   3   4
LRB1    3   4   2.2p
XJJ2    4       jjbranch    IC=250u
XBIAS2  4       ibias       IBIAS=175u
L2      4   5   10p
XJJ3    5       jjbranch    IC=250u
B2      5   7   JJMOD       IC=180u
RB2     5   6   4
LRB2    6   7   2.2p
L3      7   8   1p
L4      5   9   3p
B3      9   11  JJMOD       IC=180u
RB3     9   10  4
LRB3    10  11  2.2p
LB      b   12   2p
XBIAS3  12      ibias       IBIAS=175u
XJJ4    12      jjbranch    IC=250u
L5      12  13  3p
B4      13  15  JJMOD       IC=180u
RB4     13  14  4
LRB4    14  15  2.2p
XJJ5    15      jjbranch    IC=250u
XBIAS4  15      ibias       IBIAS=175u
L6      15  16  10p
B5      16  18  JJMOD       IC=180u
RB5     16  17  4
LRB5    17  18  2.2p
L7      18  8   1p
XJJ6    16      jjbranch    IC=250u
L8      16  19  3p
B6      19  11  JJMOD       IC=180u
RB6     19  20  4
LRB6    20  11  2.2p
LCLK    clk 21  2p
XJJ7    21      jjbranch    IC=250u
XBIAS5  21      ibias       IBIAS=175u
L9      21  22  3p
XJJ8    22      jjbranch    IC=250u
XBIAS7  22      ibias       IBIAS=175u
L10     22  8   1p
L11     11  23  1p
XJJ9    23      jjbranch    IC=250u
XBIAS6  23      ibias       IBIAS=175u
LQ      23  q   2p
.ends

VCLK    clk 0   PULSE(0 830u 50p 2.5p 2.5ps 0 50ps)
VA      a   0   PWL(0 0 80p 0 82.5p 830u 85p 0 180p 0 182.5p 830u
     185p 0)
VB      b   0   PWL(0 0 130p 0 132.5p 830u 135p 0 180p 0 182.5p 830
     u 185p 0)
XAND    a   b   clk     q     and
RQ      q   0   2

.TRAN 0.125ps 250ps
.PLOT V(VA) V(VB) V(VCLK) V(RQ)
.END
```

Description: This example demonstrates a basic RSFQ AND gate. The AND logic is achieved by using Josephson junctions and bias currents to combine two inputs. The design showcases the reuse of subcircuits like `jjbranch` and `ibias` from the included base file (`base_cells.cir`).

Explanation: This RSFQ AND gate design includes the following key elements:

- **Josephson Branch (`jjbranch`):**
  - A Josephson junction modeled by `JJMOD` .
  - A shunt resistor and inductors forming the Josephson branch.
- **Bias Current Subcircuit (`ibias`):**
  - Provides a steady bias current using an ideal current source.
- **AND Gate Subcircuit (`and`):**
  - This subcircuit combines the two inputs, `a` and `b` , and processes them through Josephson junctions and bias currents to output a logic AND at node `q` .
  - The inputs are routed through the inductors (`L1` , `L2` , `LA` , etc.) and the Josephson junction branches.

Key Simulation Insights:

- The `.TRAN` command defines a transient simulation with a step size of `0.125ps` and a total duration of `250ps` .
- The `.PLOT` command outputs:
  - The input voltages at `VA` and `VB` .
  - The clock signal at `VCLK` .
  - The output signal at `RQ` .

Why This Example is Useful:

- This example demonstrates the construction of a rudimentary RSFQ AND gate.
- It showcases the reuse of subcircuits and the inclusion of external files using the `.include` command.
- It provides a basic example of how logic gates are implemented in RSFQ technology, demonstrating the potential for building more complex logic circuits.

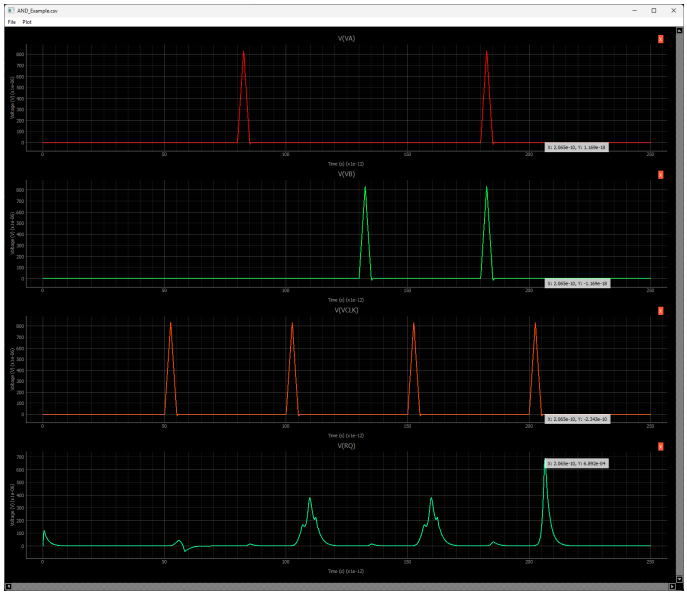


Figure 11.3: RSFQ AND gate output traces from the installed example

---

Summary

This chapter combines two kinds of examples. The workflow-oriented snippets at the beginning are manual-only starting points that you can copy into your own working directory. The full logic-cell examples at the end are installed files and should be used when the manual asks for a packaged example. Additional examples outside the installed list are not assumed to be present after installation.

Simulation Feature Reference

JoSIM®-Pro is a professional superconducting circuit simulator designed for practical circuit design, analysis, and characterization. The software combines a SPICE-like netlist flow with device models and analysis features tailored to superconducting electronics. This chapter summarizes the main capabilities of JoSIM®-Pro and explains how they fit into typical simulation workflows.

1. Core Simulation Capabilities

At its core, JoSIM®-Pro is built for transient simulation of superconducting circuits. Users describe a circuit in a SPICE-like netlist, choose the desired analysis settings, and generate output traces for inspection, plotting, or post-processing.

Key simulation capabilities include:

- **Transient simulation**: Simulate switching, biasing, clock propagation, pulse interaction, and waveform evolution in the time domain.
- **Phase and voltage workflows**: Run in either phase mode or voltage mode depending on the circuit and the quantity of interest.
- **Multiple integration methods**: Choose between backward differentiation and trapezoidal integration.
- **Fixed-step and variable-step operation**: Run with a fixed print cadence or enable adaptive stepping for the currently supported subset of problems. In the current JoSIM®-Pro v2.7.0 release, variable-step mode is supported for `PHASE + BDF`.
- **Practical startup control**: Use the default consistent start at $t = 0$, preserve user-specified initial conditions with `UIC`, or enable legacy startup priming when a workflow benefits from it.
- **Large-circuit support**: Work with hierarchical designs, reusable subcircuits, multiple output targets, and streamed text output for long or memory-intensive runs.

These capabilities make JoSIM®-Pro suitable for both early-stage design exploration and detailed verification of larger superconducting systems. The Usage chapter explains how to select these modes from the CLI, while the Syntax chapter documents the netlist commands that capture the same intent inside a circuit file.

1.1 Choosing Between Phase and Voltage Workflows

`PHASE` mode is the default operating mode in JoSIM®-Pro and is the natural choice for most superconducting transient studies, especially when the workflow depends on Josephson-junction phase evolution, SFQ-style pulse propagation, or variable-step operation. `VOLTAGE` mode remains available when a voltage-domain representation is the better fit for the circuit under study or for the way results need to be reported.

In practice, a good rule is to start in `PHASE` mode unless there is a specific reason to prefer voltage-domain solving. The selected analysis mode affects both the solved state representation and the default meaning of output requests. Users can choose the mode from the CLI with `-a` / `--analysis` or from the netlist with `.OPTION analysis=...`.

1.2 Fixed-Step and Variable-Step Operation

Fixed-step runs are the straightforward choice when a workflow needs uniformly spaced samples, easy run-to-run comparison, or a known print cadence for downstream tools. Variable-step runs are more appropriate when switching events are localized in time, long quiet intervals would otherwise be oversampled, or the goal is to reduce point count while still refining around fast dynamics.

The current JoSIM®-Pro v2.7.0 release supports variable-step mode only for `PHASE + BDF`. That means variable-step is not the right choice for voltage-mode runs or trapezoidal integration. For most circuits, it is best to begin with the default adaptive settings and only tune the detailed `.OPTION` controls if a specific circuit requires it.

2. Circuit Description and Modelling

2.1 SPICE-like Netlists, Parameters, and Hierarchy

JoSIM®-Pro uses a SPICE-like netlist syntax so that circuit definitions remain compact, readable, and easy to version-control. Users can define scalar parameters with `.PARAM`, reuse common content with `.INCLUDE`, and structure larger designs with subcircuits.

```
.param scalar=2
.param R1=50*scalar
.param L1=10E-9*0.25*scalar
R1 1 2 R1
L1 2 0 L1
```

This style of netlisting is useful for design scaling, design-space exploration, and maintaining repeated constants across larger projects.

2.2 Sources and Circuit Elements

The simulator supports the component and source models needed to build practical superconducting circuits, including Josephson junctions, resistors, inductors, capacitors, transmission lines, mutual inductive coupling, independent voltage/current/phase sources, and the standard dependent source families.

Independent sources can be driven with a broad set of built-in waveform definitions, including:

- **DC** values,
- **PWL** and **PWS** piecewise waveforms,
- **PULSE**, **SIN**, **EXP**, and **STAIR** excitations,
- **CUS** waveforms loaded from external text files,
- **NOISE**, **PNOISE**, and **FNOISE** for stochastic excitation.

This allows the same simulator to cover straightforward bias setups, pulse-driven logic experiments, external waveform replay, and noise-driven studies.

2.3 Josephson Junction Models

JoSIM®-Pro supports more than one Josephson-junction model family. Standard `JJ` and `RCSJ` models cover the usual resistively and capacitively shunted workflow, while `TJM` models support tunnel-junction modelling with fitted parameter sets.

For practical modelling work, the code supports:

- standard junction parameters such as critical current, normal resistance, subgap resistance, capacitance, and gap voltage,
- temperature-dependent RCSJ behaviour through `T` and `TC`,
- transition shaping with `DELV` and `RTYPE`,
- phase-offset and non-sinusoidal current-phase relationships through `PHI` and `CPR`,
- TJM fit data supplied inline or loaded from an external file.

This gives users enough modelling range to move from simple circuit studies to more specialised device characterisation work without changing tools.

3. Noise, Spread, and Realism

3.1 Thermal Noise

JoSIM®-Pro can include Johnson–Nyquist noise through global `.TEMP` and `.NEB` commands, and resistive devices and junctions can also override those settings locally with `temp=` and `neb=` attributes when needed.

```
.temp 4.2
.neb 10GHz
R1 1 2 50 temp=4.2 neb=10GHz
B1 2 3 JJMOD temp=4.2 neb=10GHz
```

This makes it possible to move from idealized switching studies to temperature- and bandwidth-aware transient simulations.

3.2 Explicit Noise Waveforms and Junction Noise

In addition to thermal noise, JoSIM®-Pro supports explicit stochastic waveform functions for source-driven studies and device-level perturbation:

- `NOISE(...)` for sampled white-noise style excitation,
- `PNOISE(...)` for $1/f^{\alpha}$-style amplitude noise,
- `FNOISE(...)` for flux-noise style excitation referenced to Φ_0,
- `ICNOISE(...)` on Josephson junctions for fractional critical-current noise.

```
I1 1 0 PNOISE(1u 1 0 1p 10n 64)
P1 2 0 FNOISE(1e-6 1 0 0 1p 10n 64)
B1 2 3 JJMOD ICNOISE=ICNOISE(0.01 1 0 1p 10n 64)
```

These capabilities are useful when characterising jitter sensitivity, low-frequency drift, or noise-limited switching behaviour.

3.3 Parameter Spread and Reproducibility

Manufacturing variation and device mismatch are often just as important as nominal behaviour. JoSIM®-Pro supports both global spread controls and per-component spread attributes.

```
.param R1=50
.param L1=10E-9
R1 1 2 R1 spread=0.05
L1 2 0 L1
.spread 0.02 R=0.05 B=0.10
```

This is useful when evaluating robustness, estimating yield, or understanding sensitivity to fabrication tolerances. When reproducibility matters, the random seed can be fixed from the CLI or from `.OPTION seed=...`.

4. Characterization and Analysis Workflows

4.1 Transient and Startup Workflows

The normal workflow in JoSIM®-Pro is transient analysis driven by `.TRAN`. Within that flow, users can choose how the run is initialized:

- use the default consistent start at $t = 0$,
- apply explicit initial conditions with `.IC`,
- use `UIC` when user-specified initial conditions should be taken directly,
- enable `STARTUP` when a legacy startup-priming flow is desired.

This makes the transient engine practical for both clean nominal runs and more specialised startup or state-initialisation studies.

4.2 IV Curve Generation

JoSIM®-Pro can generate IV data directly from a model definition. This is useful for characterising junction behaviour, validating model parameters, and checking expected operating ranges before a device is placed into a larger circuit.

```
.iv JJMOD 300E-6 IV.csv 400
```

This command sweeps the named model to $\pm 300\mu\mathrm{A}$ and stores the resulting current–voltage data in `IV.csv`. Because the workflow is model-based, it is useful both as a first-pass device check and as a way to compare fitted model behaviour before running a larger transient simulation.

4.3 Variation, Noise, and Repeatability Studies

Characterisation in JoSIM®-Pro is not limited to a single nominal transient run. Users can combine `.SPREAD`, explicit source or device noise, fixed random seeds, and expression-based outputs to study robustness, sensitivity, and repeatability.

A typical workflow is to begin with a nominal run, then introduce spread or noise one step at a time while keeping the seed fixed so that results remain comparable across revisions. This is particularly useful when evaluating switching margins, timing sensitivity, or the stability of a design under realistic perturbations.

4.4 Derived Outputs and Diagnostics

Output requests are not limited to raw node traces. Users can request voltages, currents, phases, apparent power, and accumulated energy, and may also build expressions directly in output statements.

```
.print V(R1) + V(R2)
.print sin(P(B01))
.print W(R1) E(B01)
```

This is useful when a workflow depends on derived traces, compact diagnostics, or direct reporting of power- and energy-related quantities without extra post-processing.

5. Output, Post-Processing, and Data Handling

5.1 Flexible Output Formats

Simulation results can be written in multiple formats depending on the downstream workflow:

- **CSV and DAT**: convenient for plotting and importing into analysis tools such as Python, MATLAB, or spreadsheets,
- **BIN**: compact binary output for offline workflows,
- **RAW**: extensionless raw output for minimal-overhead file generation.

Text formats can also be compressed in a GZIP container when file size matters.

5.2 Streaming Output and Multiple Files

For long simulations or large output sets, JoSIM®-Pro can stream text output during the run rather than waiting until the end of the simulation. In practice, streaming is intended for `CSV`, `DAT`, and `STDOUT` workflows. Offline generation remains the practical choice when `BIN` or `RAW` output is required.

When streaming is enabled, file-format choice becomes part of the workflow design. If a downstream tool requires exact `BIN` or `RAW` output, keep streaming disabled and let the run complete in buffered mode. If the priority is lower peak memory usage or immediate text output during a long run, choose streamed `CSV`, `DAT`, or `STDOUT` instead.

The `.FILE` command allows output to be split across multiple files, which is useful when separating waveform groups or organizing results by function. The same format rules apply per target, so a multi-file workflow can mix logical output groupings while still following the runtime rules for streaming versus buffered output.

5.3 Hierarchical Referencing and Expressions

When working with hierarchical circuits, JoSIM®-Pro can reference signals inside nested subcircuits using fully qualified hierarchical names. Output statements can also include mathematical expressions built from supported signal references.

This allows large designs to remain modular while still exposing the exact internal traces needed for debugging, verification, and reporting.

Summary

JoSIM®-Pro combines superconducting-device simulation, hierarchical netlist support, multiple Josephson-junction model paths, thermal and explicit noise modelling, spread analysis, IV characterization, and flexible output handling in a single workflow. Together, these capabilities make it suitable for research, design exploration, and engineering use where accurate time-domain simulation of superconducting circuits is required.

For day-to-day execution choices such as analysis mode selection, output destinations, streaming, and reproducibility controls, see the Usage chapter. For the exact netlist forms of `.TRAN`, `.IC`, `.OPTION`, `.FILE`, output expressions, and `.IV`, see the Syntax chapter.

Python Automation

The `josimpro` module exposes the JoSIM®-Pro simulation engine for scripted and repeatable workflows. In practice, the Python layer is most useful when you want to run the same circuit repeatedly, inspect results in NumPy, or export traces directly from a script.

This chapter is organized around normal user tasks:

1. make the module importable,
2. configure process-wide settings such as the license path,
3. load a circuit file,
4. adjust runtime settings for a run,
5. execute the simulation,
6. inspect results in Python, and
7. write output files for later analysis.

The netlist syntax, analysis modes, print directives, and output behavior are the same ones described in the Usage, Syntax, Features, and Examples chapters. The Python layer does not define a separate circuit language.

1. Making the module available

The Python extension module is named `josimpro`. Product installs place the compiled module in the JoSIM®-Pro binary location. The helper scripts in `utils/install_josimpro_python.sh` and `utils/install_josimpro_python.bat` add that binary directory to the Python user site-packages path by writing a `josimpro.pth` file.

If you are using a custom installation layout, make sure the directory containing the compiled `josimpro` module is on `PYTHONPATH` before importing it.

A quick import and version check looks like this:

```python
import josimpro as jp

print("Version:", jp.VERSION)
print("Built:", jp.BUILD_TIMESTAMP)
print("Git:", jp.GIT_HASH)
```

2. Process-wide settings and licensing

`jp.Settings` holds process-wide settings. These are not the per-run simulation controls. In the current Python binding, the main process-wide settings are:

- `Settings.verbosity`
- `Settings.minOutput`
- `Settings.license`
- `Settings.licensee` (read-only)
- `Settings.validity` (read-only)
- `Settings.endUse` (read-only)

A typical setup block is:

```python
import josimpro as jp

jp.Settings.license = "/path/to/sm_license.txt"
jp.Settings.verbosity = 1
jp.Settings.minOutput = False
```

Use `Settings` for process-wide presentation and licensing state. Use `Runtime` for the settings that control a specific simulation run.

3. Smallest useful file-based run

The simplest Python workflow is file-based. Construct an `Input` from a circuit file, build a `Matrix`, then run a `Simulation`:

```python
import josimpro as jp

jp.Settings.license = "/path/to/sm_license.txt"

inp = jp.Input("JTL_Example.cir")
mat = jp.Matrix(inp)
mat.validate_simulation_ready()

sim = jp.Simulation(mat)
sim.simulate(mat)

print("Simulation OK:", sim.isSimOK)
print("Accepted steps:", sim.stepCount)
```

The `Input("file")` constructor reads and parses the circuit immediately. This is the most convenient entry point when the netlist already contains the `.TRAN`, `.OPTION`, and `.PRINT` directives you want to use.

4. Runtime control from Python

Use `jp.Runtime` when you want to set or inspect run-specific controls from Python.

```python
import josimpro as jp

rt = jp.Runtime()
# also available:
rt2 = jp.Runtime.fromGlobalSettings()
```

In the current release, `Runtime.fromGlobalSettings()` is effectively a convenience constructor for the default runtime object. It does not copy the process-wide `Settings` fields such as license location or verbosity.

Two details matter immediately:

1. the Python binding currently exposes analysis and integration selection as integer fields, not Python enums;
2. unlike the CLI, the Python layer does not expose separate override masks for `.TRAN` and `.OPTION`. If the netlist sets the same field during parse, the netlist value remains authoritative.

The integer mappings are:

- `analysisType`: `0 = PHASE`, `1 = VOLTAGE`
- `integrationMethod`: `0 = BDF`, `1 = TRAP`
- `exprDivZeroPolicy`: `0 = ReturnZero`, `1 = ClampToMax`, `2 = NaN`, `3 = Error`

The runtime fields most users adjust from Python are:

- `inputFile`, `workingDir`
- `tStep`, `tStop`, `pStart`, `pStep`, `useUIC`
- `analysisType`, `integrationMethod`, `variableStepEnabled`
- `newtonJJEnabled`, `newtonMaxIters`, `newtonRelTolPhase`, `newtonAbsTolPhase`, `newtonDamping`
- `outputPrecision`, `compressedOutput`, `streamOutput`
- `globalTemp`, `globalNeb`, `globalSpread`, `globalRSpread`, `globalCSpread`, `globalLSpread`, `globalBSpread`
- `seed`

The module also exposes the full supported runtime field list through `jp.PUBLIC_RUNTIME_FIELDS`.

Important: variable-step mode is currently supported only for `PHASE + BDF`. In Python, that means `analysisType = 0`, `integrationMethod = 0`, and `variableStepEnabled = True`. Other combinations are rejected by the core runtime checks.

5. Explicit runtime example

When you want to set the run configuration from Python before reading the file, create an `Input` manually and attach the runtime first:

```python
import josimpro as jp

rt = jp.Runtime()
rt.inputFile = "JTL_Example.cir"
rt.workingDir = "."
rt.analysisType = 0          # PHASE
rt.integrationMethod = 0     # BDF
rt.variableStepEnabled = False
rt.outputPrecision = 12
rt.seed = 12345

inp = jp.Input()
inp.runtime = rt
inp.read_input()
inp.parse_input()

mat = jp.Matrix(inp)
mat.validate_simulation_ready()

sim = jp.Simulation(mat)
sim.simulate(mat)
```

This pattern is the safest starting point for scripted automation because the runtime is explicit and the input file path is tied to that runtime object.

6. Reading results into NumPy

After a run, `sim.results` provides NumPy-friendly access to the stored solution.

The two most useful helpers are:

- `results.get_time()` — returns the stored time vector as a NumPy array
- `results.get_x_full(matrix)` — returns the stored solver state as a 2D NumPy array

Example:

```
import josimpro as jp
import numpy as np

inp = jp.Input("JTL_Example.cir")
mat = jp.Matrix(inp)
sim = jp.Simulation(mat)
sim.simulate(mat)

time = sim.results.get_time()
x = sim.results.get_x_full(mat)

print(type(time), time.shape)
print(type(x), x.shape)
```

The array returned by `get_x_full(matrix)` is the stored solver state. Its rows follow the matrix column ordering used internally for the simulation, or the relevant filtered subset when output pruning is active. When you want named traces that follow the circuit's `.PRINT` plan, build an `Output` object and evaluate the print list instead.

7. Getting named traces from the print list

`Output(inp, mat)` parses the same `.PRINT` and `.FILE` directives used by the command-line flow. This is the easiest way to obtain named traces in Python.

```
import josimpro as jp
import numpy as np

inp = jp.Input("JTL_Example.cir")
mat = jp.Matrix(inp)
sim = jp.Simulation(mat)
sim.simulate(mat)

out = jp.Output(inp, mat)
out.format_time_vector(sim.results, mat)
out.compute_all_print_values(mat)

traces = {
    p.name: np.asarray(p.values, dtype=float)
    for p in out.plist
    if not p.hidden
}

for name, values in traces.items():
    print(name, values.shape)
```

Use this route when you want the same signal naming and derived-expression behavior that JoSIM®-Pro would produce from the netlist itself.

8. Writing CSV, BIN, RAW, or normal print output

The `Output` helpers let a Python script write the same kinds of files discussed in the output chapters.

The simplest choice is:

```
out = jp.Output(inp, mat)
out.print_output(sim, mat)
```

That uses the parsed print plan and whatever output routing follows from the netlist and runtime state.

When you want explicit file names from Python, first format the output time vector, then call the file writer you need:

```
import josimpro as jp

inp = jp.Input("JTL_Example.cir")
mat = jp.Matrix(inp)
sim = jp.Simulation(mat)
sim.simulate(mat)

out = jp.Output(inp, mat)
out.format_time_vector(sim.results, mat)

out.print_CSV_DAT(',', "jtl_run.csv", mat)
out.print_BIN("jtl_run.bin", mat)
out.print_RAW("jtl_run.raw", mat)
```

Use `print_CSV_DAT(',', ...)` for CSV and `print_CSV_DAT(' ', ...)` for DAT-style text output.

Practical note: `Runtime.outputFiles` is exposed read-only in Python as a record of the active parsed output targets. If you want to choose file names directly from Python, the explicit `print_*` helpers shown above are the straightforward option.

9. Large runs, streaming, and variable-step restrictions

For runs where you want to use the streaming-capable output path, pass an `Output` object to `simulate()`:

```python
import josimpro as jp

rt = jp.Runtime()
rt.inputFile = "JTL_Example.cir"
rt.analysisType = 0          # PHASE
rt.integrationMethod = 0     # BDF
rt.variableStepEnabled = True
rt.streamOutput = True

inp = jp.Input()
inp.runtime = rt
inp.read_input()
inp.parse_input()

mat = jp.Matrix(inp)
out = jp.Output(inp, mat)
sim = jp.Simulation(mat)
sim.simulate(mat, out)
```

This is the Python-side route into the streaming-aware output path.

Keep these restrictions in mind:

- variable-step mode currently supports `PHASE + BDF` only,
- the same `.PRINT`, `.FILE`, and output-format rules from the Usage and Syntax chapters still apply,
- if the netlist sets conflicting `.TRAN` or `.OPTION` values, those directives still govern the parsed run.

10. Repeated runs and batch automation

Python is most valuable when you need to run the same circuit repeatedly with controlled changes. A simple example is exporting one output file per random seed:

```python
import josimpro as jp

for seed in (1001, 1002, 1003):
    rt = jp.Runtime()
    rt.inputFile = "JTL_Example.cir"
    rt.seed = seed
    rt.outputPrecision = 12

    inp = jp.Input()
    inp.runtime = rt
    inp.read_input()
    inp.parse_input()

    mat = jp.Matrix(inp)
    sim = jp.Simulation(mat)
    sim.simulate(mat)

    out = jp.Output(inp, mat)
    out.format_time_vector(sim.results, mat)
    out.print_CSV_DAT(',', f"jtl_seed_{seed}.csv", mat)
```

The same structure works for sweeps over output precision, analysis mode, spread settings, temperature, Newton settings, or other runtime fields that are already available on `Runtime`.

11. End-to-end plotting example

The following script runs a circuit, pulls the stored solver state into NumPy, and plots every stored row. It is intentionally simple; for publication-quality plots you will usually select a smaller set of named traces from `Output.plist` instead.

```python
import josimpro as jp
import matplotlib.pyplot as plt

rt = jp.Runtime()
rt.inputFile = "JTL_Example.cir"
rt.analysisType = 0          # PHASE
rt.integrationMethod = 0     # BDF
rt.variableStepEnabled = True

inp = jp.Input()
inp.runtime = rt
inp.read_input()
inp.parse_input()

mat = jp.Matrix(inp)
sim = jp.Simulation(mat)
sim.simulate(mat)

time = sim.results.get_time()
y = sim.results.get_x_full(mat)

fig, axes = plt.subplots(y.shape[0], 1, sharex=True)
for i, ax in enumerate(axes):
    ax.plot(time, y[i])
plt.show()
```

Which produces the resulting plot:

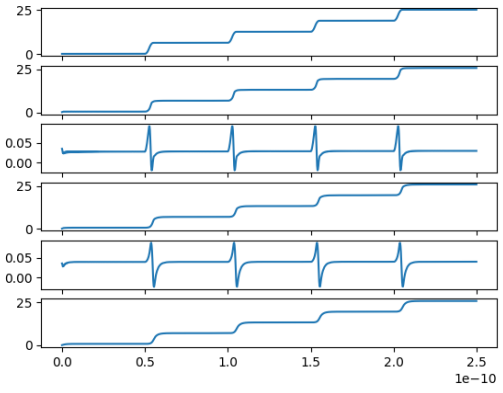


Figure 13.1: JTL example simulated and plotted using the JoSIM®-Pro Python interface

12. Practical reminders

For day-to-day Python use, the main points are:

- use `Settings` for licensing and process-wide presentation state,
- use `Runtime` for run-specific controls,
- prefer `Input("file") -> Matrix -> Simulation` for the simplest scripts,
- use `Output` when you want named traces or exported files,
- treat `analysisType` and `integrationMethod` as integer fields in the current binding,
- remember that variable-step mode requires `PHASE + BDF`, and
- remember that netlist directives still remain authoritative when they set the same fields during parse.

Troubleshooting

This chapter is organized by what the user sees when launch, licensing, simulation, or result generation fails. Start with GUI launch and licensing issues, then move to file, netlist, runtime-option, and CLI-specific problems.

1. The GUI launcher does not start

Typical symptoms

- The `JoSIM-Pro` Start Menu or Desktop shortcut is missing after Windows installation.
- Running `josim-pro-gui` on Linux prints `missing or non-executable GUI binary`.
- The Linux desktop entry appears, but the application does not open.
- A terminal cannot find `josim-pro-cli` or `optimum-cli` even though the GUI shortcut exists.

What this usually means

- The optional Windows shortcut or PATH feature was not selected, or the terminal was opened before installation finished.
- The Linux package is incomplete, was installed without dependency resolution, or the installed GUI executable is missing or not executable.
- The Linux desktop entry is present, but the wrapper cannot reach the installed GUI files.
- Command-line discovery and GUI launcher discovery are separate installation concerns.

Most likely fixes first

- On Windows, repair or rerun the MSI and select the Start Menu/Desktop shortcut options if you want launchers.
- If a Windows terminal cannot find CLI tools, open a new terminal after installation, enable command-line PATH integration, or run the tools from the installed `bin` directory.
- On Linux, install the package with `apt install`, `dnf install`, or `yum install`; avoid `dpkg -i` or `rpm -i` because they do not resolve dependencies automatically.
- On Linux, launch the GUI with `/usr/local/bin/josim-pro-gui`; use the wrapper rather than starting the GUI binary directly.
- If the wrapper still reports a missing executable, reinstall or repair the package. That error indicates a packaging or staging problem, not a netlist problem.

2. The GUI reports a licensing problem or opens in degraded mode

Typical symptoms

- The startup shell stays at `Checking license...` or reports `Licensing unavailable`.
- The startup shell offers **Retry**, **Continue degraded**, and **Quit**.
- The main workspace opens with a degraded-mode status message.
- `Help→About` reports `Status: Degraded mode / licensing unavailable`
- **Generate System ID...** fails or no `SM_SysID.txt` file appears in the selected folder.

What this usually means

- The SUN Magnetics License Manager is not installed, is outdated, or its `smlicensed` daemon cannot be started.
- The configured license path is empty, incorrect, or points to a file the application cannot read.
- The licensing service cannot be contacted.
- The license is expired, inactive, tied to different hardware, or not valid for this installation.
- All available concurrent seats are already in use.

Most likely fixes first

- Install or update the SUN Magnetics License Manager first. System ID generation depends on `smlicensed`; JoSIM-Pro does not generate `SM_SysID.txt` by itself.
- Choose **Retry** after correcting the License Manager installation, license file, licensing service, network, firewall, or seat-availability issue.
- Choose **Continue degraded** when you need to inspect the GUI. Simulations still require a valid license, and System ID generation still requires `smlicensed`.
- Open `Help→About` and use **Generate System ID...** to ask `smlicensed` to save `SM_SysID.txt` when support requests a system identifier.
- On non-standard deployments, set `SMLICENSED_PATH` to the installed `smlicensed` executable and restart JoSIM-Pro.
- If the license file was issued but is not detected automatically, set the license path once with `josim-pro-cli -l /path/to/sm_license.txt` by itself, then restart the GUI. Do not add `-v`; version display

exits before the license path is saved.
- Contact support@sun-magnetics.com for expired, deactivated, hardware-mismatched, or account-side license issues.

3. Command-line launch exits immediately

Typical symptoms

- `Unknown switch: ...`
- `Input file and output file names are the same.`
- The program prints usage/version information and exits.

What this usually means

- A command-line option was misspelled or is not supported.
- The same file was accidentally used for both input and output.
- The command was run without a valid input file or without the intended options.

Most likely fixes first

- Check the switch spelling against Chapter 9.
- Make sure the input and output paths are different.
- If you intended to pipe a netlist through standard input, provide input explicitly with `-i STDIN` instead of running the executable with no arguments.

Example

```
# Wrong: same path used for both input and output
josim-pro-cli circuit.cir -o circuit.cir

# Correct
josim-pro-cli -o results.csv circuit.cir
```

4. Input files, `.include` paths, and subcircuit boundaries

Typical symptoms

- `The file ... could not be found.`
- `Invalid .include statement: ...`
- `Missing end of subcircuit ...`
- `Mismatched .ends name: ...`
- `Unknown subcircuit specified in line: ...`
- `Incorrect number of I/O nodes for subckt ...`
- A cyclic include warning or cyclic subcircuit expansion error.
- The GUI opens a netlist, but simulation fails immediately because an included file is missing.

What this usually means

- A referenced file does not exist at the expected path.
- An included file is being resolved from the wrong working directory.
- A `.subckt` block was not closed properly, or the `.ends` name does not match.
- A subcircuit call uses the wrong subcircuit name or the wrong number of connection nodes.
- Two files include each other, or a subcircuit expands into itself recursively.

Most likely fixes first

- In the GUI, prefer `File→Open Folder...` for projects that depend on nearby include files.
- Check the path exactly as written in the netlist.
- Keep included files near the main netlist or use explicit relative paths that are stable.
- Verify every `.subckt` has a matching `.ends`.
- Check that each `X...` line provides exactly the number of I/O nodes expected by the target subcircuit.
- Remove cyclic include or cyclic subcircuit relationships.

Example

```
.include models/junctions.inc

.subckt JJ_STAGE in out bias
B1 in out JJMOD
R1 out bias 2
.ends JJ_STAGE

X1 n1 n2 nbias JJ_STAGE
```

If the included file is not found, or if `X1` does not provide the correct node count, JoSIM®-Pro will stop before simulation begins.

5. Netlist syntax, component definitions, and model lines

Typical symptoms

- `Unknown parameter ... found in component ...`
- `Duplicate component ... detected.`
- `Missing model for ...`
- `Invalid model line specified: Missing parameters.`
- `Unknown model type: ...`
- `Unknown model parameter in ...`
- `Invalid mutual inductance line specified: ...`

What this usually means

- A component instance uses an attribute that the component does not support.
- Two devices share the same identifier.
- A Josephson junction or other model-backed element references a model that does not exist.
- A `.MODEL` line is missing the type, parentheses, or one or more name=value pairs.
- A mutual inductance entry references bad syntax or invalid parameters.

Most likely fixes first

- Compare the line against the syntax reference rather than editing it by guesswork.
- Make all component names unique.
- Confirm that the model name on the component line exactly matches a defined `.MODEL`.
- For model definitions, check the type first: `JJ`, `RCSJ`, or `TJM` as appropriate.
- For mutual inductance, verify that both inductors exist and that the coupling value is parseable.

Example

```
.MODEL JJMOD JJ(RN=10 CAP=2.5p IC=1m)
B1 1 2 JJMOD area=1.2
```

A line such as `B1 1 2 MISSING_MODEL` will fail immediately because the model does not exist.

6. Parameters, expressions, and control commands fail to parse

Typical symptoms

- `String (...) cannot be converted to double.`
- `Mismatched parentheses in expression.`
- `Invalid parameter definition found: ...`
- `Unparseable parameters found.`
- `Invalid spread command defined.`
- `Invalid .IC syntax ...`
- `Invalid .option ... value: ...`

What this usually means

- A value that should evaluate to a number does not.
- A parameter expression contains missing parentheses or an unsupported form.
- A control statement such as `.SPREAD`, `.IC`, or `.OPTION` is malformed.

Most likely fixes first

- Reduce the failing expression to the smallest possible form and rebuild it.
- Check every `name=value` pair for missing delimiters.
- For `.OPTION analysis=` and `.OPTION integration=`, use only supported values.
- For `.OPTION outputprecision=`, keep the value in the documented range.
- For `.IC`, make sure the left-hand side targets a valid node or typed quantity and the right-hand side evaluates to a finite number.

Examples

```
.PARAM scale = 2
.PARAM Rbias = 50*scale
.SPREAD 0.02 R=0.05 B=0.10
.OPTION analysis=PHASE integration=BDF outputprecision=12
.IC V(n1)=0
```

If a parameter expression cannot be evaluated, fix that expression first before debugging downstream errors.

7. Source functions and noise expressions are rejected

Typical symptoms

- `Unsupported function type: ...`
- `Too few values to form ... function.`
- `Too many values to form ... function.`
- `The file ... could not be found.`
- `Invalid DC function value.`

What this usually means

- The waveform name is misspelled.
- The argument count does not match the selected function.
- A custom waveform file referenced by `CUS(...)` cannot be opened.
- A numeric argument in `NOISE(...)`, `PNOISE(...)`, `FNOISE(...)`, `ICNOISE(...)`, `PULSE(...)`, or another function is missing or malformed.

Most likely fixes first

- Start from the documented function signature and re-enter the arguments in order.
- Check commas, spaces, and parentheses before changing the function itself.
- For `CUS(...)`, verify the file path and that the file contains numeric sample values.
- For noise functions, check every numeric argument, especially the sample/update interval and correlation-time style terms.

Examples

```
I1 1 0 NOISE(1u 0 1p)
I2 2 0 PNOISE(1u 1 0 1p 10n 64)
P1 3 0 FNOISE(1e-6 1 0 0 1p 10n 64)
B1 4 5 JJMOD ICNOISE=ICNOISE(0.01 1 0 1p 10n 64)
```

If the parser reports a function error, correct that line first. Later simulation errors often disappear once the source expression is valid.

8. Output requests are rejected or produce the wrong signals

Typical symptoms

- `Invalid .print directive: ...`
- `The component ... requested for output does not exist in the circuit.`
- Output appears empty or does not contain the signals you expected.
- The GUI waveform area opens with fewer traces than expected.

What this usually means

- The `.PRINT` or `.PLOT` form is invalid.
- A node or component name was mistyped.
- The request uses the wrong output form for the quantity being asked for.

Most likely fixes first

- Check the exact print form rather than editing the line incrementally.
- Make sure the requested node or device actually exists in the flattened circuit.
- Use device-oriented forms such as `W(R1)` or `E(R1)` for power and energy on supported two-terminal devices.
- In the GUI, check run diagnostics before assuming the plot view is at fault.

Examples

```
.PRINT V(n1)
.PRINT I(R1)
.PRINT P(B1)
.PRINT W(R1) E(R1)
.PRINT @R1[I]
```

If a request references a name that is not present in the circuit, fix the name first before investigating the output format.

9. Variable-step mode or runtime options are incompatible with the netlist

Typical symptoms

- `Variable-step mode currently supports PHASE + BDF only.`
- `Variable-step mode is not supported for LOSSLESS transmission lines.`
- `Blocking LOSSLESS lines: T1, T2.`
- `Invalid .option analysis value: ...`
- `Invalid .option integration value: ...`

What this usually means

- Variable-step mode was enabled for an unsupported analysis or integration method.
- The circuit contains one or more `LOSSLESS` transmission lines. The diagnostic lists the blocking line names.
- A runtime option was given an unsupported value.

Most likely fixes first

- If you want variable-step mode, use `PHASE` analysis with `BDF` integration.
- Use fixed-step mode for `LOSSLESS` transmission lines, including untyped `T` elements because `LOSSLESS` is their default model.
- Where the finite LC-ladder model is physically appropriate, select `TYPE=LCLADDER NSEG=...`; it supports variable-step `PHASE + BDF` through its generated inductors and capacitors.
- Re-check boolean and enumerated `.OPTION` values instead of trying nearby spellings.

Example

```
.OPTION analysis=PHASE integration=BDF variablestep=ON
```

If that run contains a `LOSSLESS` transmission line, disable variable-step mode or use a validated fixed-step setup. Do not change to `LCLADDER` only to avoid the restriction: the two models have different numerical and physical approximations.

10. Transmission-line initial conditions or delayed breakpoints are rejected

Typical symptoms

- `ICV and ICI must be specified together.`
- `Invalid initial-condition dimensions ...`
- `Explicit transmission line initial conditions require ANALYSIS=VOLTAGE.`
- `Explicit initial conditions are not supported for WIDEBAND ...`

Most likely fixes first

- Supply both `ICV` and `ICI`; do not specify only voltage or only current.
- Use one value for scalar lines, two differential/common values for `MODAL`, and exactly `NCOND` values for `@MTL`.
- Use `ANALYSIS=VOLTAGE`. Remove the explicit line state from `PHASE` runs.
- Remove `ICV`/`ICI` from `WIDEBAND` lines. Their recursive relaxation state is not defined by voltage and current alone.
- Delayed events and required interpolation history have no fixed application count cap. JoSIM-Pro retains the history needed by the line delay; sufficiently large runs still require adequate system memory.

Delayed-event breakpoints improve edge placement but do not prove timestep convergence. Repeat the run with a smaller nominal `.TRAN` step when validating edge shape or amplitude.

11. The simulation starts but fails, diverges, or produces no useful result

Typical symptoms

- `Simulation failed. Matrix has no solution.`
- `Simulation failed. Matrix has no solution during Newton iteration.`
- The run completes, but the output is clearly not what you expected.

What this usually means

- The circuit description is internally inconsistent.
- A model, bias condition, or initial condition is driving the solver into an impossible or unstable state.
- The requested analysis settings are not suitable for the circuit as written.

Most likely fixes first

- Check for missing grounds, missing models, or impossible parameter values.
- Reduce the problem to a smaller reproducible netlist.
- Remove optional spread/noise terms and verify the nominal circuit first.
- Try a simpler startup path before adding `UIC`, `STARTUP`, or advanced runtime tuning.
- For difficult circuits, confirm that the requested output and analysis modes are appropriate.

Practical debugging approach

1. Get the circuit running without spread or explicit noise.
2. Verify that all models and included files resolve correctly.
3. Add complexity back one feature at a time.

If a very small, valid-looking reproducer still fails in the same way after these checks, that is the point where it is reasonable to treat the behavior as a likely defect and report it.

12. Large runs, output files, compression, and streaming issues

Stored-result memory grows with the number of retained samples and traces. Advance reservation is bounded, but there is no fixed total trajectory memory budget. Use `--stream` with CSV or DAT to avoid retaining the complete trajectory; solver state, delay history and output buffers still require memory. If allocation fails, the CLI and C API report insufficient memory. Failed file streams discard temporary output and preserve an existing destination. Data already written to standard output cannot be retracted.

Typical symptoms

- `Compressed output is not supported for RAW output.`
- `Compressed output is not supported for BIN output.`
- A warning that streaming does not support the requested extension and will write CSV instead.
- Streaming output stops unexpectedly or the produced files are not in the format you expected.

What this usually means

- Compression was requested for a format that does not support it.
- Streaming mode was used with `.raw`, `.bin`, or another unsupported extension.
- The output target fell back to CSV because the requested format is not available in streaming mode.

Most likely fixes first

- Use `CSV` or `DAT` if you want compressed text output.
- Use buffered output instead of streaming if you need `RAW` or `BIN` output.
- Read warnings about fallback formats carefully; they are often telling you exactly what file was written instead.
- For very large runs, keep output requests focused so you are not writing unnecessary traces.

Examples

```
# Good streaming target
josim-pro-cli -p -o results.csv design.cir

# Better for buffered binary/raw workflows
josim-pro-cli -o results.bin design.cir
josim-pro-cli -o results.raw design.cir
```

13. Optimum exits before Margin, Yield, or Optimize runs

Typical symptoms

- `Configuration file missing mode value.`
- `Configuration file missing verify table.`
- `Verify table missing file value.`
- `Verify table missing file circuit.`
- `... is not a valid mode of operation.`
- Optimum prints `Mode: ...`, then stops before Margin, Yield, or Optimize output appears.
- An optimized netlist is not written even though `optimize.output` is set.

What this usually means

- The TOML file is missing the top-level `mode` value or the required `[verify]` table.
- `verify.file` or `verify.circuit` does not point to a readable file from the TOML file location.
- The nominal verification failed. Optimum intentionally stops before Margin, Yield, or Optimize when the baseline circuit does not verify.
- A selected parameter does not exist in the active circuit or subcircuit scope.
- `optimize.output` points to a directory that does not exist or cannot be written.

Most likely fixes first

- Start from the installed `DFF_Example_config.toml`, `DFF_Example.cir`, and `DFF_Example.sp` files before adapting a larger project.
- Keep the TOML, circuit, specification file, and include files in one project directory while validating relative paths.
- Confirm that `mode` is exactly one of `"verify"`, `"margin"`, `"yield"`, or `"optimize"`.
- Run `mode="verify"` first. Only move to Margin, Yield, or Optimize after nominal verification passes.
- Check `parameters.search_string`, explicit `[parameters]` entries, and `margin.subcircuit` against the actual parameter names in the netlist.
- Remember that Optimum runs through `optimum-cli` and a TOML configuration file. Use the GUI to save netlists or inspect generated files, then run Optimum from a terminal.

14. A practical order for debugging difficult runs

When several errors appear at once, fix them in this order:

1. GUI launcher and package-installation problems
2. licensing, degraded mode, and System ID generation problems
3. missing files and `.include` errors
4. syntax, model, and parameter errors
5. source-function and output-request errors
6. runtime-option compatibility issues
7. solver failures and result-quality problems
8. CLI-specific invocation, output, compression, or streaming problems
9. Optimum TOML, verification, parameter-selection, or output-path problems

That order matters. Many later-stage failures are simply downstream effects of an earlier launch, license, parse, or setup error.

Native Integration Appendix

This appendix keeps the native integration surfaces in view without turning them into a second automation manual. For most scripting and batch workflows, the Python chapter should remain the first choice. Use the native interfaces when you need to embed JoSIM®-Pro into an existing compiled application, when Python is not a suitable dependency, or when you are integrating from C, C++, Delphi, or Free Pascal.

A.1 What the native interfaces are for

The native interfaces are provided in two forms:

- a **C API** based on opaque integer handles for input, matrix, simulation, output, and runtime objects;
- a **Pascal import unit** that exposes the same exported functions for Delphi- and Free Pascal-style consumers.

These interfaces are useful when you want to call the simulator directly from another compiled program. They complement the Python chapter, which covers script-level automation.

A.2 When to use this instead of Python

Prefer the native interfaces when one or more of the following is true:

- your application is already written in C, C++, Delphi, or Free Pascal;
- you want to embed simulation into a GUI, service, or larger native tool;
- you need a C-callable boundary for another language or framework;
- Python is not available or not desirable in the deployment environment.

If your goal is mainly repeatable simulation runs, result inspection, and file export from scripts, the Python interface is simpler and should usually be preferred.

A.3 Minimal file-based workflow

The smallest practical native workflow is file-based and closely follows the normal simulator pipeline:

1. create an input handle from a circuit file with `JPRO_Input_CreateFromFileA(...)` or, on Windows, `JPRO_Input_CreateFromFileW(...)`;
2. optionally create a runtime handle when you need per-run overrides, then build the matrix with `JPRO_Matrix_CreateFromInputEx(...)`; otherwise use `JPRO_Matrix_CreateFromInput(...)`;
3. create a simulation handle with `JPRO_Simulation_CreateFromMatrix(...)`;
4. run the solve with `JPRO_Simulation_Run(...)` or `JPRO_Simulation_RunWithOutput(...)`;
5. either:
   - retrieve raw result arrays with `JPRO_Simulation_GetResults(...)` and the `JPRO_Results_*` accessors, or
   - create an output handle with `JPRO_Output_CreateFromInputMatrix(...)` and write CSV, binary, RAW, or console output with the `JPRO_Output_*` functions.
6. destroy all handles when complete.

All exported functions return a status code. When a call fails, retrieve the message with `JPRO_GetLastError()` before continuing.

A.4 Path and encoding notes

The file-based entry points are the recommended way to create inputs.

- Functions with an `A` suffix take UTF-8 paths.
- `JPRO_Input_CreateFromFileW(...)` is available on Windows for UTF-16 paths.
- The current runtime path setters exposed in the native API use UTF-8 `A`-style entry points.

In practice, this means UTF-8 paths are the portable default for native integrations, while Windows callers can use the wide-character input constructor when needed.

A.5 Where the authoritative reference lives

This appendix is intentionally brief. For exact signatures, argument types, calling conventions, and exported names, use the installed API files or the matching distribution files as the authoritative reference:

- `CAPI/jpro_capi.hpp` for the C API;
- `CAPI/JoSIM_CAPI.pas` for the Pascal import unit.

If this appendix and those files ever disagree, follow the header or Pascal unit for the package version you are using.