SUN Magnetics

JoSIM-Pro v1.4

User Manual

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

www.sun-magnetics.com

Date: September 25, 2025

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

Introduction

JoSIM-Pro is an advanced circuit simulation tool developed for analyzing superconducting circuits and Josephson junctions. Leveraging the experience and knowledge gathered from the original JoSIM tool and other state-of-the-art software tools, JoSIM-Pro is built from the ground up to offer enhanced capabilities, improved accuracy, and increased compatibility with a range of platforms and workflows.

JoSIM-Pro focuses on providing a fast, efficient, and highly accurate simulation experience, making it the ideal choice for researchers and engineers working in the field of superconducting electronics. Whether you are exploring basic circuit behavior or complex superconducting systems, JoSIM-Pro's robust set of features can help you simulate, visualize, and understand the results in a streamlined manner.

Unlike its predecessor, JoSIM, which was built as a compact and lightweight simulation utility, JoSIM-Pro has undergone significant redevelopment, incorporating cutting-edge algorithms and techniques. This redevelopment effort ensures that JoSIM-Pro is capable of handling larger, more complex circuits while maintaining high computational performance. JoSIM-Pro has been designed to meet the needs of both experienced researchers and new users, providing a user-friendly interface combined with the power of sophisticated simulation technology.

The ongoing development of JoSIM-Pro is centered around expanding its capabilities and improving its interoperability with other tools and software commonly used in superconducting circuit design. This makes JoSIM-Pro an adaptable tool for integration into a wide range of workflows, whether for educational purposes or for industrial research and development.

Key objectives of JoSIM-Pro include:

- **Improved Performance**: JoSIM-Pro incorporates numerous performance optimizations, making it one of the fastest superconducting circuit simulators available.
- **Cross-Platform Support**: With support for Windows, Linux, and macOS, JoSIM-Pro aims to be accessible to all users, regardless of their preferred operating system.
- **Enhanced Interoperability**: The tool is designed to integrate smoothly with popular circuit design tools and environments, ensuring that users can seamlessly move from design to simulation without disruptions.
- **Ongoing Development**: JoSIM-Pro is an active development effort, with continuous research aimed at improving the capabilities, accuracy, and speed of the simulation tool.

With JoSIM-Pro, you are equipped with the tools necessary to push the boundaries of superconducting circuit design and simulation, all while benefiting from an intuitive, efficient user experience.

System Requirements

Before installing and using JoSIM-Pro, ensure that your system meets the following minimum and recommended requirements for optimal performance. JoSIM-Pro is compatible with multiple operating systems and is designed to leverage modern hardware for faster simulations, especially when working with large superconducting circuits.

Operating Systems

JoSIM-Pro is cross-platform and supports the following operating systems:

- **Windows**: Version 10 or later (64-bit)
- **Linux**: Kernel 4.15 or later (64-bit), distributions such as Ubuntu 18.04+ or CentOS 7+
- **macOS**: Version 10.13 (High Sierra) or later

Hardware Requirements

Minimum Requirements

The following hardware configuration is the minimum required to run JoSIM-Pro:

- **Processor (CPU)**: Dual-core processor (Intel or AMD) with a clock speed of 2.0 GHz or higher
- **Memory (RAM)**: 4 GB
- **Storage**: 500 MB of available disk space for installation
- **Graphics**: Basic graphics support (integrated graphics or any discrete GPU)

Recommended Requirements

For optimal performance, especially when running complex simulations, the following hardware configuration is recommended:

- **Processor (CPU)**: Quad-core processor (Intel Core i5/i7, AMD Ryzen 5/7) with a clock speed of 3.0 GHz or higher
- **Memory (RAM)**: 16 GB or more, especially for large circuit simulations
- **Storage**: SSD with at least 1 GB of available disk space for installation and temporary files
- **Graphics**: Dedicated graphics card (NVIDIA, AMD) for enhanced visualizations and performance

Software Requirements

In addition to the hardware requirements, JoSIM-Pro has specific software dependencies and requirements:

- **C++ Redistributable**: On Windows systems, the Microsoft Visual C++ Redistributable 2015-2019 is required.
- **Python (Optional)**: Python 3.x is required for executing custom scripts or integrations with third-party simulation tools.

Optional Dependencies

JoSIM-Pro can integrate with various third-party tools for enhanced functionality. These optional dependencies are recommended if you plan to extend JoSIM-Pro's capabilities:

- **SPICE Netlist Format**: JoSIM-Pro can read and simulate SPICE-format netlists. Ensure you have compatible circuit design tools that can export SPICE files.
- **Visualization Tools**: For advanced data visualization, tools such as MATLAB, Gnuplot, or Python-based plotting libraries (e.g., Matplotlib) can be used to analyze simulation results.
- **LaTeX**: If you need to generate technical documentation or reports directly from JoSIM-Pro, having a LaTeX distribution installed (e.g., TeX Live or MikTeX) can be helpful.

Network Requirements

JoSIM-Pro can be used offline, but certain features, such as software updates and external package integrations, may require an internet connection:

- **Internet Access**: A stable internet connection is required for checking for updates, downloading optional packages, and accessing cloud-based resources.

Licensing Information

JoSIM-Pro operates under a commercial license with various tiers depending on the user's role (e.g., academic, industry). Ensure that you have the appropriate license file for your usage. For licensing inquiries, please contact:

- **Support Email**: support@sun-magnetics.com
- **Phone**: +27 21 555 1234

Unsupported Systems

If the system of use does not meet the above minimum requirements support cannot be guaranteed. We do, however, endeavour to support all systems and will make special arrangements with clients on a case by case basis to support systems with non-standard hardware.

Getting Started

This chapter will guide you through the process of installing, configuring, and running JoSIM-Pro for the first time. Whether you are a seasoned professional or new to superconducting circuit simulation, these steps will help you get JoSIM-Pro up and running quickly and efficiently.

1. Installation

JoSIM-Pro can be installed on Windows, Linux, and macOS. Follow the appropriate instructions for your operating system.

Windows Installation

1. **Download the Installer**: Visit the official SUN Magnetics website and download the latest Windows installer.

2. **Run the Installer**: Double-click the installer and follow the on-screen instructions. Make sure to choose an appropriate installation directory (e.g., `C:\ProgramFiles\SUNMagnetics\JoSIM-Pro`).

    - Ensure to select the option to add the installation directory to the `PATH`.

3. **Install Dependencies**:

    - JoSIM-Pro requires the **Microsoft Visual C++ Redistributable 2015-2019**. If you do not have this installed, the installer will prompt you to install it.

4. **Launch JoSIM-Pro**: Once installed, you can launch JoSIM-Pro by opening a command prompt or Windows Terminal application and running the command `josim-pro-cli`. The licensing procedure will need to be followed hereafter.

5. **(Optional) Python interface**: Included with the installation is a Python module that can be used with the with the local Python install by executing the included batch script:

    ```
    path\to\install\directory\utils\install_josimpro_python.bat
    ```

Linux Installation

1. **Download the Package**: Download the appropriate package for your Linux distribution (e.g., DEB for Ubuntu or RPM for CentOS) from the JoSIM-Pro website.
2. **Install the Package**:
   - For Debian-based systems (e.g., Ubuntu), run the following command:
     ```
     $ sudo dpkg -i josim-pro_x.x.x_amd64.deb
     ```
   - For Red Hat-based systems (e.g., CentOS), use:
     ```
     $ sudo rpm -i josim-pro_x.x.x_amd64.rpm
     ```
3. **Launch JoSIM-Pro**: After installation, JoSIM-Pro can be executed by typing `josim-pro-cli` in the terminal. The licensing procedure will need to be followed hereafter.
4. **(Optional) Python interface**: Included with the installation is a Python module that can be used with the with the local Python install by executing the included shell script:
   ```
   path/to/install/directory/utils/install_josimpro_python.sh
   ```

macOS Installation

1. **Download the Disk Image**: Download the `.dmg` file from the JoSIM-Pro website.
2. **Install the Application**:
   - Open the `.dmg` file and drag JoSIM-Pro to the Applications folder.
3. **Launch JoSIM-Pro**: You can launch JoSIM-Pro from a terminal by typing `josim-pro-cli` in a terminal window. The licensing procedure will need to be followed hereafter.
4. **(Optional) Python interface**: Included with the installation is a Python module that can be used with the with the local Python install by executing the included shell script:
   ```
   path/to/install/directory/utils/install_josimpro_python.sh
   ```

2. Initial Setup

After installation, JoSIM-Pro requires some basic configuration.

License Activation

1. **Launch JoSIM-Pro**: Upon first launch, a `SysID.txt` will be generated in the current working directory.

2. **Obtain License Key**: This `SysID.txt` needs to be mailed to support@sun-magnetics.com with the purchase order number or invoice number as subject to obtain a `jpro_license.txt`.

3. **License File**: JoSIM-Pro requires a hardware locked license to be able to execute. Once the `jpro_license.txt` has been issued it needs to be placed in the `Licenses` folder in the installed directory (e.g. for Windows, `C:\ProgramFiles\SUNMagnetics\Licenses`).

3. Running Your First Simulation

Once JoSIM-Pro is installed and configured, you're ready to run your first superconducting circuit simulation. Here's a step-by-step guide:

Step 1: Creating a Circuit File

1. **Create a Netlist**: JoSIM-Pro uses SPICE-like netlists to define circuits. Create a simple text file, `example.cir`, with the following contents:

```
* Simple Josephson Junction Circuit
B1 1 2 JJMOD
L1 1 0 1e-9
R1 2 0 1
V1 3 0 DC 0.001
.MODEL JJMOD JJ(RN=1, CAP=1e-12)
.TRAN 0.1ps 1ns
.END
```

This defines a simple Josephson junction circuit with a voltage source, a resistor, and an inductor.

Step 2: Running the Simulation

1. **Open a terminal**: Launch a terminal application and navigate to the circuit file.

2. **Start the Simulation**: Run the simulation by typing the following command into the terminal window: `josim-pro-cli -o example.csv path/to/example.cir`. JoSIM-Pro will process the netlist and simulate the circuit over the defined time interval (`1ns` in this case). *Note: The examples are stored in the installation directory which might not be user writeable. Rather execute the examples from a different user writeable directory.*

3. **View the Progress**: As the simulation progresses JoSIM-Pro will update the terminal window with the progress of each step in real time. Once the simulation is completed the total execution time will be displayed and the results will be stored in the `example.csv` file.

Step 3: Analyzing the Results

JoSIM-Pro does not officially provide a way to analyse the results yet, however, here are a few ways to do this:

- **Waveform Viewer**: Use the provided Python waveform viewer script to visualize voltages, currents, and junction phases. To install this script the following command needs to be run. *Note: This requires Python 3 with PyPI.*

```
pip install path/to/install/directory/utils/waveform_viewer-0.3.8.tar.gz
```

  The results can be viewed using the command:

```
waveform_viewer example.csv # Linux and macOS systems
waveform_viewer.bat example.csv # Windows systems
```

- **Third-party Tools**: External tools can be used to visualize the data in the produced CSV file. These tools include Microsoft Excel, MATLAB and GNUPlot.

4. Troubleshooting

If you encounter any issues during installation or while running JoSIM-Pro, here are a few common solutions:

- **License Issues**: Verify that the `jpro_license.txt` exists and is in the correct location. Additionally JoSIM-Pro can be directly pointed to the `jpro_license.txt` through the `-l` command-line switch. (e.g. `josim-pro-cli -l jpro_license.txt -o example.csv example.cir`).
- **Simulation Errors**: If the simulation does not run, check the netlist for syntax errors or missing components. Refer to the Syntax Chapter in this documentation for details on valid netlist formats and elements.
- **Performance Issues**: If simulations are running slowly, ensure that your system meets the recommended hardware requirements and close other resource-heavy applications.

5. Next Steps

Now that you have successfully installed JoSIM-Pro and run your first simulation, you are ready to explore more advanced features. Refer to the upcoming chapters for detailed tutorials on:

- **Complex Circuit Design**
- **Parameterization and Other Advanced Features**
- **Interfacing JoSIM-Pro with Other Tools**

JoSIM-Pro is continuously updated with new features and optimizations, so be sure to check for software updates regularly to take advantage of the latest improvements.

Features

JoSIM-Pro builds upon the robust foundation of the original JoSIM tool, providing a powerful, fast, and efficient simulation environment for superconducting circuits. In addition to the core functionality offered by JoSIM, JoSIM-Pro expands these advanced features to significantly enhance the user experience and broaden the tool's capabilities. These features make JoSIM-Pro an indispensable tool for both research and industrial applications.

This chapter will highlight the key features that set JoSIM-Pro apart from other simulators and explain how they can be leveraged in superconducting circuit simulations.

1. Core Features

JoSIM-Pro inherits several core features from the original JoSIM, providing a solid foundation for simulating superconducting circuits, including:

- **SPICE-like Netlist**: JoSIM-Pro uses a SPICE-like netlist format, making it easy for users familiar with traditional circuit simulation tools to define their circuits.
- **Josephson Junction Model**: The core element of JoSIM is its ability to model Josephson junctions (JJ) using the RCSJ model, which allows for the accurate simulation of superconducting electronics.
- **Transient Analysis**: JoSIM-Pro supports transient analysis, enabling users to simulate time-dependent phenomena in superconducting circuits.
- **Export Options**: JoSIM-Pro allows simulation results to be exported to CSV format for easy post-processing and analysis in third-party tools such as MATLAB, Python, or Excel.

2. Advanced Features

JoSIM-Pro includes several features that significantly enhance the simulation capabilities. These features are designed to improve simulation performance, flexibility, and ease of use.

2.1 Parameterization of Component Values

One of the key features of JoSIM and JoSIM-Pro is the ability to parameterize component values using the `.param` command. This feature allows users to define variables for component values (such as resistances, inductances, etc.) and use these variables in their netlists. JoSIM-Pro processes these parameters using Dijkstra's shunting yard algorithm, ensuring efficient evaluation of mathematical expressions.

- Example:

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

  This example defines a resistor `R1` with a resistance of 100Ω and an inductor `L1` with an inductance of 10 nanohenry. Parameterization allows easy modification of values across large circuits by changing just the `.param` definitions.

2.2 Noise Addition

JoSIM-Pro allows users to add Johnson-Nyquist noise to their simulations using the `.temp` and `.neb` commands. Johnson-Nyquist noise, which occurs due to the thermal motion of charge carriers in resistors, is critical in accurately simulating real-world superconducting circuits.

- The `.temp` command specifies the temperature of the system, which influences the noise level.
- The `.neb` command defines the noise bandwidth, allowing fine control over the noise profile in the simulation.

```
.temp 4.2
.neb 10GHz
```

- Example:

  ```
  R1 1 2 100
  L1 2 0 10n
  .temp 4.2
  .neb 10GHz
  ```

  This example defines a resistor `R1` with a resistance of 100Ω and an inductor `L1` with an inductance of 10 nanohenry. the temperature is set to 4.2 Kelvin, and the noise bandwidth is set to 10 GHz.

By including these commands a noise current source is automatically added in parallel to every resistor in the circuit. This noise current then injects the effect of thermal noise from the resistor into the network. Noise uses a random number generator (RNG) with a seed calculated on epoch. This seed is displayed the first time an RNG is used and can be re-used via the *seed* command-line flag.

2.3 External File Inclusion

JoSIM-Pro supports external file inclusion through the `.include` command, making it easy to modularize complex designs by breaking them into smaller files. This feature allows users to maintain cleaner netlists and reuse common subcircuits or parameter definitions.

- Example:

```
.include common_components.cir
```

  This command includes the contents of the `common_components.cir` file, which might define frequently used components or parameters.

2.4 Parameter Spread

JoSIM-Pro supports the `.spread` command, which allows for the randomization of parameters with every run of the simulation. This feature is particularly useful for exploring the impact of manufacturing variations or environmental factors on circuit performance.

- Example:

```
.param R1=50
.param L1=10E-9
R1 1 2 R1
L2 2 1 L1
.spread R=0.2 0.1
```

  This example will vary the value of `R1` by ±20% and all other components by ±10% across multiple runs of the simulation, simulating real-world variations.

2.5 Subcircuits and Subcircuit Parameterization

JoSIM-Pro allows users to define subcircuits, which are reusable blocks of components that can be instantiated multiple times in a larger design. Subcircuits improve the organization of complex designs and facilitate reuse of common circuit elements.

In JoSIM-Pro, subcircuits can be parameterized by appending the component name and value to the end of the instantiation call. This adds flexibility to the instantiation of subcircuits.

- Example of subcircuit definition:

```
.subckt amp 1 2 3
R1 1 2 50
L1 2 3 10n
.ends amp
```

- Instantiation of the subcircuit with parameterization:

```
Xamp1 1 2 3 amp R1=100
```

  This instantiation of the subcircuit `amp` overrides the value of `R1` with 100Ω.

2.6 IV Curve Generation

JoSIM-Pro supports IV (current-voltage) curve generation using the `.iv` command, which allows users to generate IV curves for Josephson junctions or other circuit elements. IV curves are essential for characterizing the behavior of superconducting circuits, particularly when working with Josephson junctions.

- Example:

  ```
  .iv JJ1 300E-6 IV.csv 400
  ```

  This command will generate the IV curve for the Josephson junction model `JJ1` to a maximum current of ±300$\mu\mathrm{A}$. It will store the results in the file `IV.csv` and have a resolution of 400 steps to reach the maximum current.

2.7 Output File Compression, Binary Format Support and Streaming

JoSIM-Pro supports the compression of output files to the `tar.gz` format, which reduces the size of simulation result files. This is especially useful when working with large-scale simulations that generate large datasets.

JoSIM-Pro also supports binary output formats, providing a more compact representation of the results, which can be processed faster by certain analysis tools.

Output can also be directly streamed to file during simulation through a non-blocking thread. This greatly reduces the amount of memory required in large/long simulations since produced output is not stored in memory but instead directly written to file as simulation progresses.

2.8 Phase and Voltage Mode Simulations

JoSIM-Pro introduces the ability to perform both phase and voltage mode simulations. This dual-mode capability allows for more flexible analysis, enabling users to study both the phase dynamics of Josephson junctions and the voltage behavior of the circuit. This enables faster circuit design by setting the phase at specific nodes directly, completely removing the need to impedance match surrounding circuitry.

2.9 Differential Methods for Simulation

JoSIM-Pro supports two differential methods for simulating circuits: the **backward differential method** and the **trapezoidal differential method**.

- **Backward Differential Method**: A stable method suitable for stiff systems, ensuring convergence in cases where circuits have elements with widely varying time constants.

- **Trapezoidal Differential Method**: A method offering higher accuracy in capturing transient behavior but can introduce numerical oscillations in certain circuits.

Users can switch between these methods depending on the requirements of their simulation, balancing accuracy and stability.

2.10 Output expression manipulation

JoSIM-Pro supports expression parsing on output traces. This allows users to make compact output files by performing operations on traces before outputting to file. This is also supported in streaming mode.

- Example 1:

  ```
  .print V(R1) + V(R2)
  ```

  This command will produce an output that is the sum of the voltages across R1 and R2.

- Example 2:

  ```
  .print sin(P(B01))
  ```

  This command will take the sinusoid of the phase across B01 at each time step.

3. Summary

JoSIM-Pro extends the capabilities of the original JoSIM tool with several advanced features, including parameterization, noise modeling, subcircuit handling, and flexible output formats. These features, combined with the tool's ability to handle large-scale simulations efficiently, make JoSIM-Pro a powerful platform for researchers and engineers working with superconducting circuits.

Usage

JoSIM-Pro is a terminal-based superconducting circuit simulator designed to offer a variety of options for customizing simulations. This chapter provides an in-depth guide on how to use the JoSIM-Pro command-line interface and explains each available option in detail.

Basic Command Structure

The basic syntax for running JoSIM-Pro from the terminal is as follows:

```
josim-pro [options] input
```

input: This is the path to the input netlist file that defines the superconducting circuit to be simulated. If no input file is provided, JoSIM-Pro expects input via *STDIN*.

Where *STDIN* is the standard user provided input for the terminal application. i.e. Line-by-line input until **.end** is typed and submitted.

Available Options

Here is a detailed explanation of each available option:

-V, --verbose

Description: Sets the verbosity level of the output. Verbose output provides insights into the simulation's internal steps, useful for debugging or detailed logging.

Usage:

```
josim-pro -V 2 example.cir
```

Values:

- 0 – None (no extra output) [DEFAULT]
- 1 – Minimal (basic status messages)
- 2 – Medium (detailed progress information)
- 3 – Heavy (full diagnostic information)

-a, --analysis

Description: Specifies the type of analysis to perform: either *phase mode* or *voltage mode*.

Usage:

```
josim-pro -a 1 example.cir
```

Values:

- 0 – Phase Mode (analyzes circuit in phase mode) [DEFAULT]
- 1 – Voltage Mode (analyzes circuit in voltage mode)

-c, --compressed

Description: Stores the simulation output in a compressed *gzip* container. This option cannot be used with *binary (.bin)* output.

Usage:

```
josim-pro -c -o example.csv example.cir
```

Note: This option helps reduce the size of output files, making it ideal for large simulations.

-h, --help

Description: Displays the help menu with a summary of available options.

Usage:

```
josim-pro -h
```

Note: Use this command if you need a quick reference for the available options.

-i, --input

Description: Specifies the input file path or uses *STDIN* if not provided.

Usage:

```
josim-pro -i example.cir
```

Note: If input is provided via STDIN, JoSIM-Pro waits for the netlist data to be entered directly into the terminal.

-l, --license

Description: Provides the path to the license file (`license.txt`). JoSIM-Pro requires a valid license file to execute simulations.

Usage:

```
josim-pro -l license.txt example.cir
```

-m, --minimal

Description: Disables most of the output, allowing for *silent execution* of the simulator. Useful when running batch jobs where output is not required.

Usage:

```
josim-pro -m example.cir
```

-o, --output

Description: Specifies the output file path. JoSIM-Pro supports multiple output formats, including:

- `.csv` – Standard comma-separated values for post-processing
- `.dat` – Data format for scientific applications
- `.bin` – Binary format for compact storage
- No extension – Output in raw format

Usage:

```
josim-pro -o output.csv example.cir
```

-p, --stream

Description: Streams the output to file during simulation. File stream is opened at simulation start, writing results after each simulation step. Intermediate file is appended *.part* extension to indicate in progress. Early simulation termination still produces result up to termination. Dramatically reduces memory usage for large simulations.

Usage:

```
josim-pro -p -o example.csv example.cir
```

-r, --seed

Description: Specifies seed used in RNG. Ideal for reproducibility of noise functions. Current seed is displayed the first time a RNG is used in simulation.

Usage:

```
josim-pro -r 123456 example.cir
```

-s, --sysid

Description: Generates a *system identifier* required for licensing. When run, it outputs a `SysID.txt` file that must be sent to SUN Magnetics to obtain a license.

Usage:

```
josim-pro -s
```

-v, --version

Description: Displays the current version information of JoSIM-Pro.

Usage:

```
josim-pro -v
```

Example Output:

```
JoSIM-Pro: Professional Superconductor Circuit Simulator
Copyright (C) 2024 SUN Magnetics
v1.0.241014
```

-x, --integration

Description: Specifies the *integration method* to use during the simulation. JoSIM-Pro supports two integration methods:

- 0 – BDF (Backward Differentiation Formula) [DEFAULT]
- 1 – Trapezoidal

Usage:

```
josim-pro -x 0 example.cir
```

Note: The *BDF method* is better for stiff systems, while the *Trapezoidal method* offers improved accuracy for transient analysis but may introduce numerical oscillations in certain cases.

Example Usage

Basic Simulation Command:

```
josim-pro -V 1 -o results.csv example.cir
```

This command runs the simulation with minimal verbosity and saves the results to `results.csv`.

Using Multiple Options:

```
josim-pro -a 1 -c -o compressed_output.csv example.cir
```

This command performs a *voltage mode analysis* and saves the results in a compressed gzip file.

Specifying the License File:

```
josim-pro -l license.txt -o output.dat example.cir
```

Summary

The JoSIM-Pro command-line interface provides extensive customization options for running simulations efficiently. This chapter outlined each available option in detail, including their usage and valid parameters. By mastering these options, users can fine-tune their simulations to match specific requirements and optimize their workflow.

Syntax

This chapter provides an overview of the netlist syntax used in JoSIM-Pro. The syntax allows users to define superconducting circuits with precision. It follows a SPICE-like syntax, with additional features specific to superconducting circuit elements.

---

1. General Structure of a Netlist

A JoSIM-Pro netlist consists of various components, models, and control commands. Below is a sample netlist:

```
* Example Josephson Junction Circuit
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
V1 2 0 DC 0.001
.MODEL JJMOD JJ(RN=10, CAP=1e-12)
.TRAN 0.1ps 1ns
.END
```

The netlist consists of:

- **Components**: Resistors, inductors, capacitors, and Josephson junctions.
- **Models**: Used to define parameters for components like Josephson junctions.
- **Control Commands**: Specifies how the simulation should run.

---

2. Components

2.1 Resistor

Syntax:

```
R<name> <node1> <node2> <resistance> [spread=<spread>] [temp=<temp>] [neb=<freq>]
```

Example:

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

2.2 Inductor

Syntax:

```
L<name> <node1> <node2> <inductance> [spread=<spread>]
```

Example:

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

2.3 Capacitor

Syntax:

```
C<name> <node1> <node2> <capacitance> [spread=<spread>]
```

Example:

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

2.4 Josephson Junction (JJ)

Syntax:

```
B<name> <node1> <node2> <model> [area=<area>(default: 1)] [spread=<spread>] [ic=<
    ic>(default: model ic)] [temp=<temp>] [neb=<freq>]
```

Example:

```
B1 1 2 JJMOD area=1.2 spread=0.1 temp=4.2
.MODEL JJMOD JJ(RN=10, CAP=1e-12)
```

Model Parameters for Josephson Junctions

Parameter	Area	JJ_{IC}	
VG or VGAP	$(-\infty, \infty)$	2.8E-3	Junction gap voltage
IC or ICRIT	IC * Area		JJ_{IC}
RN	$(0, \infty)$	5	Junction normal resistance
R0	$(0, \infty)$	30	Junction subgap resistance
C or CAP	$(0, \infty)$	2.5E-12	Junction capacitance
T	$(0, \infty)$	4.2	Junction temperature in Kelvin
TC	$(0, \infty)$	9.1	Critical temperature of material
DELV	$(0, \infty)$	0.1E-3	Transition voltage from subgap to normal
D	[0.0, 1.0]	0.0	Transparency affecting phase relationship
ICFCT	[0, 1]	$\pi/4$	Ratio of critical current to step height
PHI	[0, 2π]	0	Phase offset (e.g., π-junction capability)
CPR	$(-\infty, \infty)$	1	Harmonic amplitudes for phase relationship

Effect of junction area on model parameters. If `area` is unset and the user specifies `ic`:

$$\texttt{area} = \frac{\texttt{ic}_{\text{user}}}{\texttt{model->ic}}$$

Parameter	Update	Scaling w.r.t. $area$
ic	$ic \leftarrow ic \times area$	$\propto area$
c	$c \leftarrow c \times area$	$\propto area$
r_n	$r_n \leftarrow \dfrac{r_n}{area}$	$\propto \dfrac{1}{area}$
r_0	$\begin{cases} r_0 = r_n, & rType = 0, \\ r_0 \leftarrow \dfrac{r_0}{area}, & rType \neq 0 \end{cases}$	$\propto \dfrac{1}{area}$

2.5 Transmission Line

Syntax:

```
T<name> <node1+> <node1-> <node2+> <node2-> TD=<time_delay> Z0=<impedance>
```

Example:

```
T1 1 0 2 3 TD=1ns Z0=50
```

Description: A transmission line connects two sets of nodes with a specified time delay and characteristic impedance.

2.6 Mutual Inductance

Syntax:

```
K<name> <inductor1> <inductor2> <coupling_factor>
```

Example:

```
K1 L1 L2 0.99
```

Description: The coupling factor is a value between 0 and 1, indicating how tightly two inductors are coupled.

---

3. Independent Sources

3.1 Voltage Source

Syntax:

```
V<name> <node1> <node2> <source_type>
```

Example:

```
V1 1 0 DC 1.0
```

3.2 Current Source

Syntax:

```
I<name> <node1> <node2> <source_type>
```

Example:

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

3.3 Phase Source

Syntax:

```
P<name> <node1> <node2> <source_type>
```

Example:

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

3.4 Source Types

JoSIM-Pro supports various types of functions that can be used as sources in circuits. Below we detail the syntax for the different source types.

Piece-wise Linear (PWL)

Generates a signal by linearly interpolating values between the provided time and value pairs (T & A).

Syntax:

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

> **Note:** T1 and A1 need to be 0 0 to coincide with the start of the simulation.

Pulse

Generates a single or repeating pulse train between two amplitudes with a given rise and fall time, as well as, pulse width.

Syntax:

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

- A1 - Amplitude 1
- A2 - Amplitude 2
- Td - Time delay before first pulse

  Default: 0
- Tr - Rise time between A1 and A2

  Default: Tstep
- Tf - Fall time between A2 and A1 if PW specified

  Default: Tstep
- Pw - Pulse width

  Default: Tstop
- Per - Periodicity time, repeats until end of simulation

  Default: Tstop

Sinusoid

Syntax: Creates a sinusoidal waveform with amplitude, frequency, and phase shift.

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

- A0 - Offset
- A - Amplitute
- f - frequency

  Default: 1/TStep
- Td - Time delay

  Default: 0
- θ - Amplitude modulation

  Default: 0

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

Custom Waveform

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

Syntax:

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

- file.txt - Path to plain text file containing single line of space delimited amplitudes
- Ts - Seperation time between the amplitudes in *file.txt*

  Default: Tstep
- Sf - Scaling factor which scales amplitudes in *file.txt*

  Default: 1
- Im - Interpolation method:
  - 0 - None (square wave)
  - 1 - Linear
  - 2 - Cubic

  Default: 0
- Td - Time delay before start of wave

  Default: 0
- Per - Periodicity time, repeats until end of simulation

  Default: Tstop

DC

Provides a constant DC value (A).

Syntax:

```
DC A
DC 1.0
```

Noise

Generates noise with a specified amplitude and sample time.

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 (inverse of sample frequency)

  Default: Tstep

Produces function: $f(t) = A\frac{G}{\sqrt{2T_s}}$, where G is a Gaussian random.

Exponential

Produces an exponentially increasing or decreasing waveform.

Syntax:

```
EXP(A1 A2 Td1 τ1 Td2 τ2)
EXP(0 1 1n 1p 2n 1p)
```

- A1 - Initial amplitude
- A2 - Final amplitude
- Td1 - Rise time

  Default: Tstep
- τ_1 - Rise time constant

  Default: Tstep
- Td2 - Fall time

  Default: Tstep
- τ_2 - Fall time constant

  Default: Tstep

Generates function: $f(t) = A_1 + (A_2 - A_1)(1 - e^{\frac{t-T_{d1}}{\tau_1}}) + (A_1 - A_2)(1 - e^{\frac{t-T_{d2}}{\tau_2}})$

Piece-wise Sinusoidal

Generates a sinusoidal signal for each segment defined. Similar to a PWL, but points points are splined between time value pairs.

Syntax:

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

> **Note:** T1 and A1 need to be 0 0 to coincide with the start of the simulation.

Stair

Generates a stair step function which rises by provided amplitude at set intervals with given rise time.

Syntax

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

- A - Amplitude
- Per - Period

  Default: 0.0
- Td - Delay time

  Default: Tstep
- Tr - Rise time

  Default: Tstep
- A0 - Starting amplitude

> **Note:** This type of source is ideal for phase source clock injection.

---

4. Dependent Sources

4.1 Current Controlled Current Source (CCCS)

Syntax:

```
F<name> <node1> <node2> <control_pos> <control_neg> <gain>
```

Creates a `<gain>` amplified current source that is controlled by the current between `<control_pos>` and `<control_neg>`.

4.2 Current Controlled Voltage Source (CCVS)

Syntax:

```
H<name> <node1> <node2> <control_pos> <control_neg> <transresistance>
```

Creates a current source that is controlled by the voltage across `<control_pos>` and `<control_neg>` divided by the `<transresistance>`.

4.3 Voltage Controlled Current Source (VCCS)

Syntax:

```
G<name> <node1> <node2> <control_pos> <control_neg> <transconductance>
```

Creates a voltage source that is controlled by the current between `<control_pos>` and `<control_neg>` divided by the `<transconductance>`.

4.4 Voltage Controlled Voltage Source (VCVS)

Syntax:

```
E<name> <node1> <node2> <control_pos> <control_neg> <gain>
```

Creates a `<gain>` amplified voltage source that is controlled by the voltage across `<control_pos>` and `<control_neg>`.

---

4. Control Commands

4.1 Transient Analysis

Syntax:

```
.TRAN <time_step> <stop_time> [<print_start_time>] [<print_step>]
```

Example:

```
.TRAN 0.01ns 10ns
```

4.2 Parameter Definition

Syntax:

```
.PARAM <name>=<mathematical_expression>
```

Example:

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

4.3 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:

```
X<name> <io_nodes> [component_name=<parameter>]
```

Usage example:

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

4.4 Include Files

Syntax:

```
.INCLUDE "<filename>"
```

Example:

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

4.5 Noise Settings

Syntax:

```
.TEMP <temperature_in_K>
.NEB <bandwidth_in_Hz>
```

Example:

```
.TEMP 4.2
.NEB 1GHz
```

4.6 Parameter Spread

Syntax:

```
.SPREAD <percentage> [L=<inductor_spread>] [C=<capacitor_spread>] [R=<
   resistor_spread>] [B=<junction_spread>]
```

Example:

```
.SPREAD 0.2 L=0.1 C=0.05 R=0.1 B=0.2
```

4.7 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, `<device>` refers to the component inside the innermost subcircuit, and each `<subckt>` represents the subcircuit in which the device or node resides, moving 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))
```

4.8 IV Curve Generation

Syntax:

```
.IV <modelname> <max_current> <filepath> [<current_steps>]
```

Example:

```
.IV JJMOD 0.01 example_iv.csv 200
```

This command generates a current vs voltage plot for the model(`<modelname>`) to ±current(`<max_current>`) in number of steps(`<current_steps>`) and stores the results in `<filepath>` .

4.9 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.

---

5. 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

---

6. Summary

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

Examples

This chapter presents advanced examples of circuits that can be simulated using JoSIM-Pro. The focus is on practical superconducting circuits using Josephson junctions, including Josephson Transmission Lines (JTLs), Destructive Flip-Flops (DFFs), and more advanced logic circuits based on RSFQ (Rapid Single Flux Quantum) technology.

1. Josephson Transmission Line (JTL)

```
* 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`.

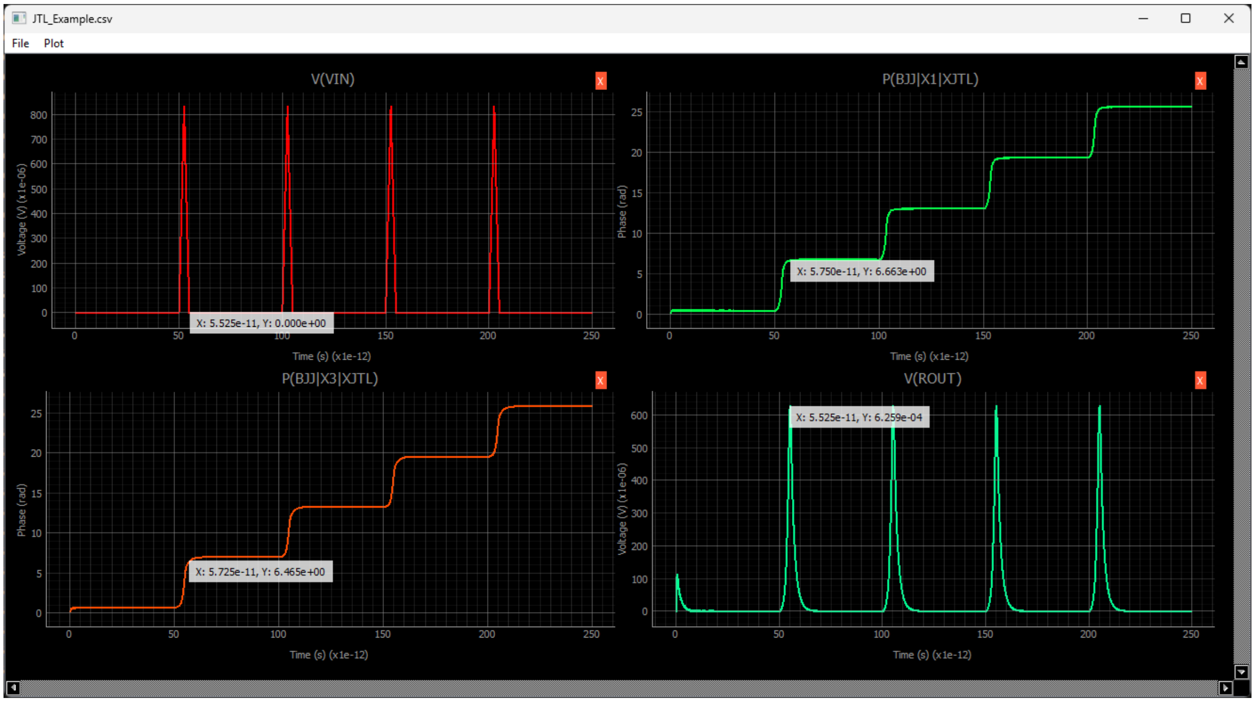


Figure 1: JTL Example output traces visualized using included script

---

2. RSFQ Splitter Cell

```
* 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.

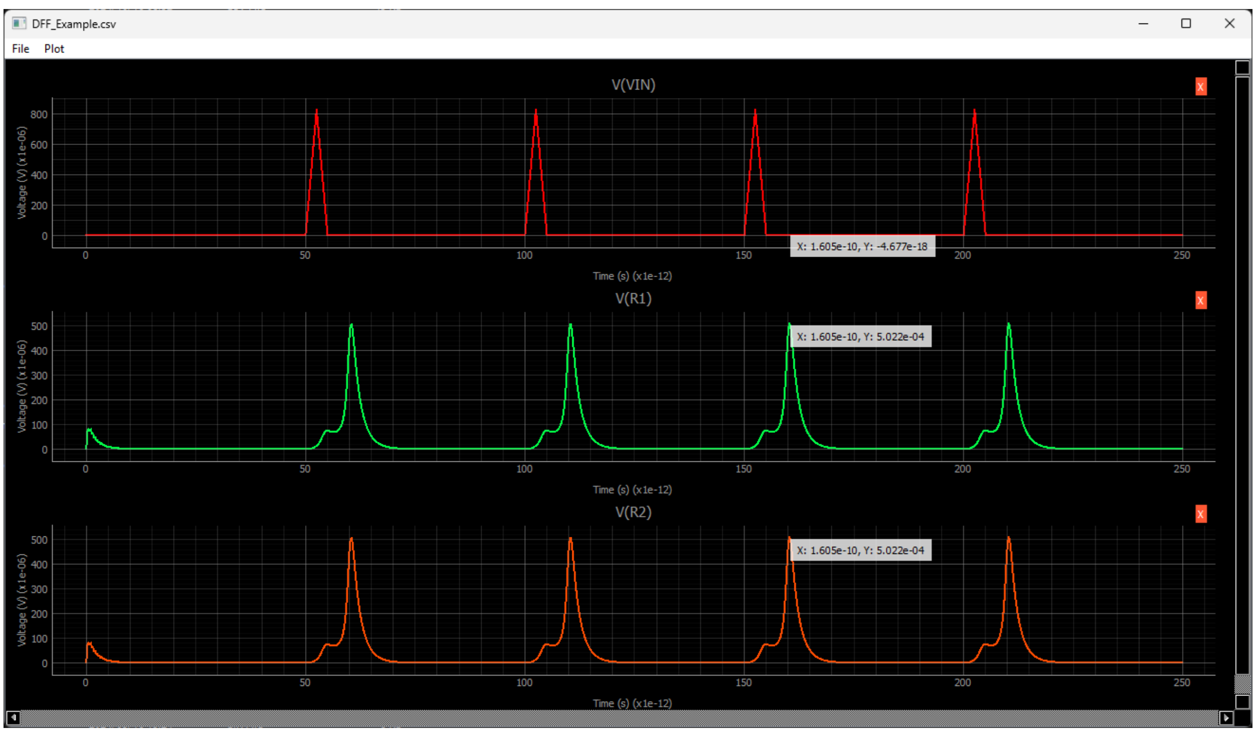


Figure 2: Splitter Circuit output traces visualized using included script

3. RSFQ AND Gate

```
* 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 830u 185p 0)
XAND     a    b    clk      q      and
RQ       q    0    2

.TRAN 0.25ps 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 `0.25ps` 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.

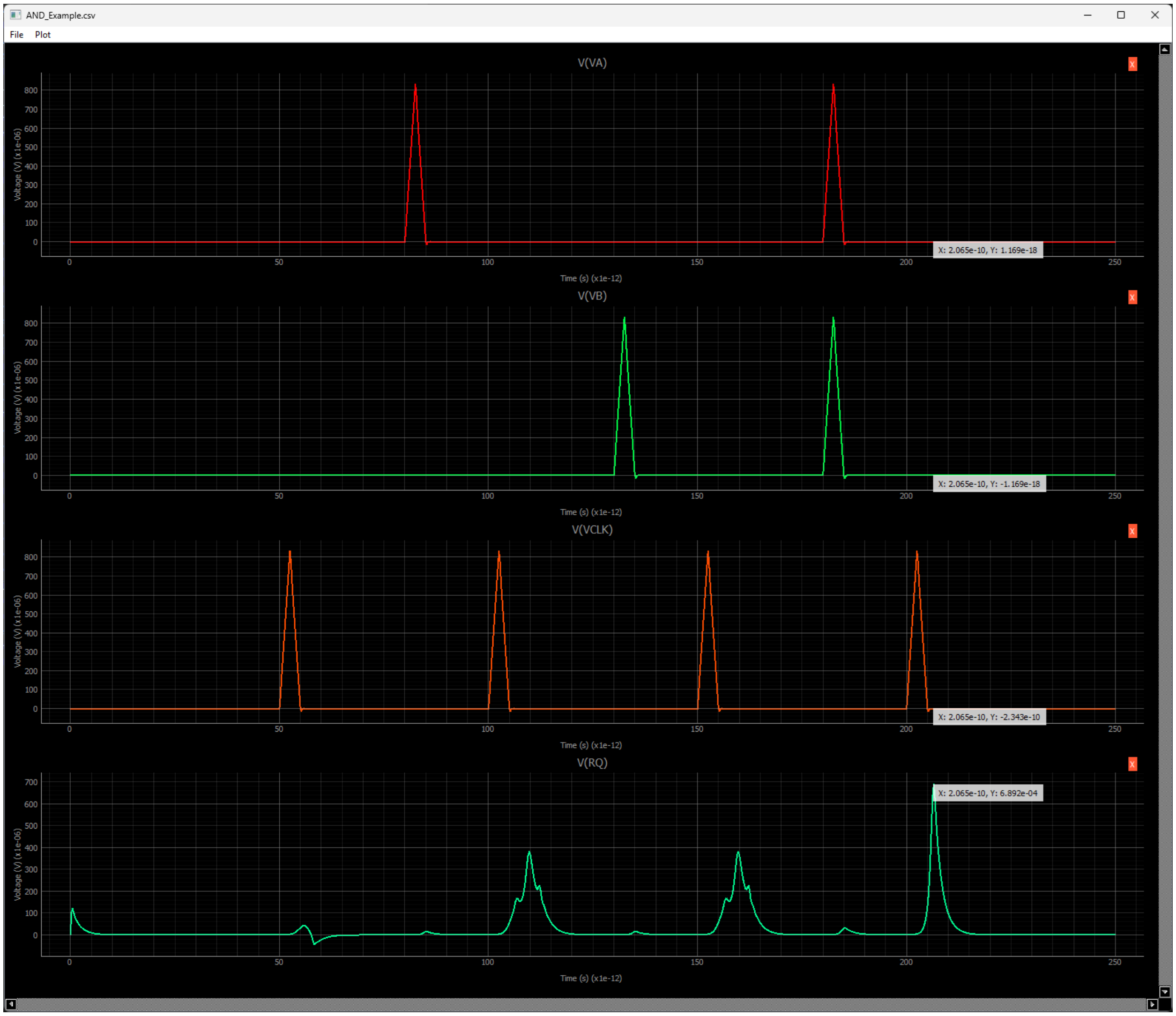


Figure 3: RSFQ AND Gate output traces visualized using included script

Python Module

Version: `josimpro.VERSION` (see below for how to display version)

JoSIM Pro is a simulation engine for circuit simulations. This Python API provides object-oriented access to the core functionality of JoSIM Pro. You can configure simulation settings, read and parse circuit netlists, build the simulation matrix, run simulations, and access the results. The module also exposes licensing settings so you can inspect (and in some cases adjust) global parameters.

1. Module Overview

When you import the module:

```
import josimpro
```

You gain access to the following classes and members:

- **Settings**: Global static settings that control simulation parameters and license location.
- **Input**: Represents input from a netlist file or string.
- **Netlist, SubCircuit**: Data structures representing the parsed netlist.
- **Model and Param**: Classes for specifying device models and parameters.
- **Matrix**: Represents the circuit matrix (built from an Input) and stores component data.
- **Simulation**: Runs the circuit simulation on a given Matrix.
- **Results**: Contains raw simulation results (with helper methods to return NumPy arrays).
- **Print, Output**: Handle output formatting and printing of simulation results.

2. Settings

The global settings are stored in the `josimpro.Settings` class. These are static variables that control various aspects of the simulation and license location. Some values are meant to be read-only, while others can be adjusted by the user.

For example:

```
import josimpro

# View the current global temperature
print("Global Temperature:", josimpro.Settings.globalTemp)

# Adjust some user-configurable settings:
josimpro.Settings.minOutput = True
josimpro.Settings.verbose = 2

# The license location should be set before simulation if different from default:
josimpro.Settings.license = "/path/to/my/license.txt"

# Display license validity (formatted as a human-readable date):
print("License Validity:", josimpro.Settings.validity)
```

3. Input and Netlist

3.1 Input

The `Input` class is used to load and parse a netlist. You can create an Input object by either using a default constructor or by passing a file name. Example:

```
# Create an Input object from a netlist file.
inp = josimpro.Input("path/to/my_netlist.cir")

# Or specify the lines individually
inp = josimpro.Input()
inp.fileLines = ["L01 0 1 2H", "R01 1 2 2", "C01 2 0 2"]
inp.read_input()     # Reads the netlist lines
inp.parse_input()    # Parses the input and builds the Netlist
```

The parsed netlist is stored in the netlist member, which is an instance of the Netlist class.

3.2 Netlist

Holds the raw lines from the netlist file and various parsed data:

```
net = inp.netlist
print(net.fileLines)  # All lines of the netlist
```

The module automatically converts STL containers (like lists and dictionaries) to their Python equivalents.

4. Model and Param

4.1 Model

The `Model` class represents a circuit model with various parameters (e.g. gap voltage, critical current). For example:

```
model = josimpro.Model()
model.modname = "MyModel"
model.vg = 0.003
print("Model type:", model.type)
```

4.2 Param

A `Param` object represents a parameter with an expression, a value, and a flag indicating whether it has been parsed.

```
param = josimpro.Param("Vth", 0.7, True)
print("Parameter:", param.exp, param.value, param.parsed)
```

5. Matrix

The `Matrix` class represents the simulation matrix built from the input netlist. It is a key object that encapsulates the parsed netlist, component data, and other simulation-related parameters.

Create a Matrix object using an Input object:

```
mat = josimpro.Matrix(inp)
```

You can also access public members for inspection:

```
print("Matrix parameters:", mat.params)
print("Number of components:", len(mat.components))
```

Note: The components property is exposed as a read-only list of component pointers.

6. Simulation and Results

6.1 Simulation

The `Simulation` class runs the simulation on a given Matrix. It stores solution vectors and simulation progress.

Example:

```python
sim = josimpro.Simulation(mat)
sim.simulate(mat)
print("Simulation OK:", sim.isSimOK)
```

6.2 Results

The simulation results are stored in `sim.results`. To convert raw pointer data into Python-friendly NumPy arrays, helper methods are provided.

For example, to get the time vector as a NumPy array:

```python
time_arr = sim.results.get_time()
print("Time steps:", time_arr.shape)
```

To get the entire simulation data as a 2D NumPy array (the helper requires the Matrix object to determine the number of variable rows):

```python
x_full = sim.results.get_x_full(mat)
print("Simulation data shape:", x_full.shape)
```

Under the hood, `get_x_full()` infers the number of variables (either from the matrix's `relevant` vector or its total number of columns) and constructs a NumPy array of shape (`num_vars`, `number_of_time_steps`).

7. Output

The `Output` class formats and prints the simulation results. It uses a list of Print objects to store different output commands.

Example usage:

```python
# Create an Output object using the Input and Matrix.
out = josimpro.Output(inp, mat)
# Format the output results (pass the simulation results and simulation size)
out.format_output(sim.results, mat, sim.simSize)
# Print output (for example, to the console)
out.print_output(sim, mat)
```

Additional functions allow you to export results in various formats:

- `print_CSV_DAT(del, fname, printIndex=0)`
- `print_BIN(fname, printIndex=0)`
- `print_RAW(fname, printIndex=0)`
- `print_COUT(printIndex=0)`

The `plist` (list of Print objects) and the output `time` vector are available as attributes if you need to further process or inspect the printed results.

8. Example

Putting this all together in an example:

```python
# This imports the library and gives it a shortened name
import josimpro as jp
# Import some libraries to show results
import matplotlib.pyplot as plt
# Create an input object from a circuit netlist
inp = jp.Input("JTL_Example.cir")
# Create a matrix from the parsed input
mat = jp.Matrix(inp)
# Create an output object to inform the matrix object what it needs to store
out = jp.Output(inp, mat)
# Create a simulation object using the matrix object
sim = jp.Simulation(mat)
# Perform the simulation
sim.simulate(mat)
# Extract the time axis from the results
time = sim.results.get_time()
# Extract the relevant results using the matrix object
y = sim.results.get_x_full(mat)
# Create some plots with a common time axis
fig, axes = plt.subplots(y.shape[0], 1, sharex=True)
# Plot each of the extracted results
for i, ax in enumerate(axes):
    ax.plot(time, y[i])
```

Which produces the resulting plot:

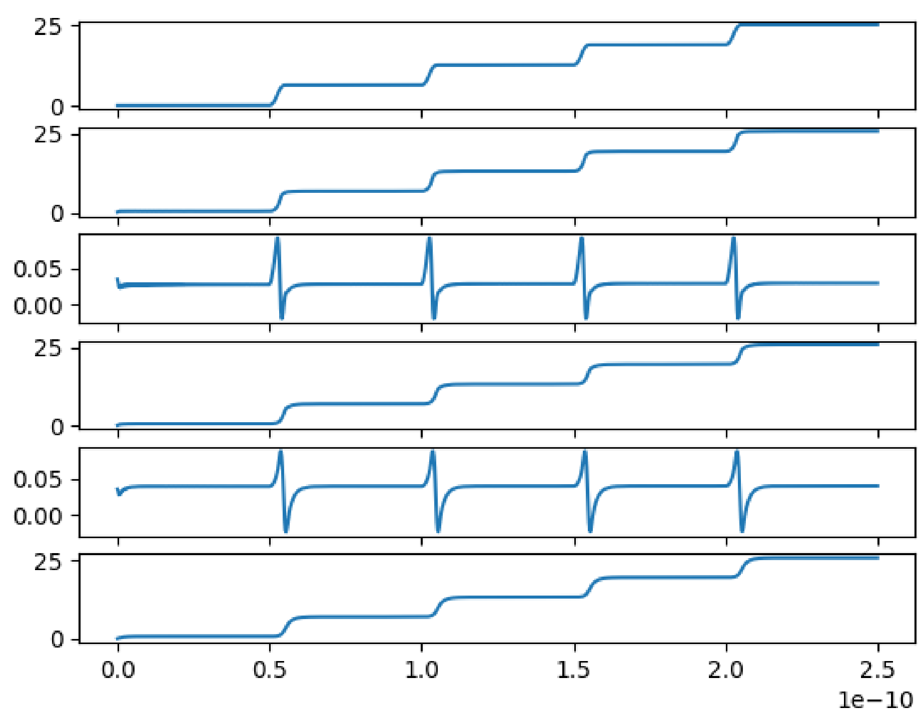


Figure 4: JTL Example simulated and plotted using JoSIM Pro Python interface

9. API Reference

This section provides an overview of the main classes and functions in the JoSIM Pro Python API. The API is exposed via the `josimpro` module.

Module-Level Attributes

`josimpro.VERSION`
  Type: `str`
  Description: The version string of the JoSIM Pro simulation engine.

Class `josimpro.Settings`

Global static settings used throughout the simulation engine.

Static Methods:

- `resetToDefault()` — Resets all settings to their default values.

Static Attributes: (Unless noted as read-only, these are user-configurable)

- `globalTemp` (double) — Global temperature.
- `globalNeb` (double) — Global Neb value.
- `globalSpread, globalRSpread, globalCSpread, globalLSpread, globalBSpread` — Various spread parameters.
- `minOutput` (bool) — Minimal output flag.
- `analysisType` (int) — Analysis type (e.g., phase or voltage).
- `verbose` (int) — Verbosity level.
- `integrationMethod` (int) — Integration method (BDF or TRAP).
- `tStep` (double) — Simulation time step.
- `tStop` (double) — Simulation stop time.
- `pStep` (double) — Print step value.
- `pStart` (double) — Print start value.
- `delayedStart` (bool) — Flag for delayed start.
- `compressedOutput` (bool) — Flag indicating whether output is compressed.
- `licensee` (`str`, read-only) — The licensed user.
- `validity` (`str`, read-only) — License validity as a human-readable date.
- `endUse` (int, read-only) — License end use flag.
- `license` (`str`) — License file path.

Class `josimpro.Input`

Represents the simulation input.

Constructors:

- `Input()` — Default constructor.
- `Input(filename: str)` — Constructs an `Input` object from a file.

Methods:

- `read_input()` — Reads input from a file or STDIN. Auto-called in file based constructor.
- `parse_input()` — Parses the input lines. Auto-called in file based constructor.

Properties:

- `inputFile` (`str`) — The file path (or STDIN) from which input is read.
- `netlist` (`josimpro.Netlist`) — The raw netlist data.

Class `josimpro.Netlist`

Holds the parsed netlist data.

Attributes:

- `fileLines (list[str])` — All lines from the input file.
- `subcircuits (dict[str, SubCircuit])` — Mapping of subcircuit names to their data.
- `controls (list[int])` — Indices of control lines.
- `maincircuit (list[int])` — Indices of main circuit lines.
- `prints (list[int])` — Indices of print commands.
- `parameters, models, variables (list[tuple[int, str]])` — Indices of parameter, model, and variable definitions.
- `includedFiles (list[Path])` — Files included via `.include` commands.

Class `josimpro.SubCircuit`

Represents a subcircuit in the netlist.

Attributes:

- `lines (list[int])` — Line indices.
- `io (list[str])` — I/O tokens.
- `parameters (list[str])` — Parameter definitions.

Class `josimpro.ExpandedLine`

Represents an expanded netlist line.

Attributes:

- `tokens (list[str])` — Tokens extracted from the line.
- `subc (str)` — Subcircuit label.

Class `josimpro.Model`

Represents a circuit model.

Constructors:

- `Model()` — Default constructor.

Attributes:

- `type (josimpro.ModelType)` — Model type.
- `modname (str)` — Model name.
- `vg (float)` — Voltage gap.
- `ic (float)` — Critical current.
- `cpr (list[float])` — Capacitance parameters.
- `fitParams (list[float])` — Fit parameters.
- `rtype (int)` — Resistance type.
- `rn, r0, c, t, tc, deltaV, d, icFct, phiOff (float)` — Various model parameters.
- `tDep (bool)` — Temperature dependence flag.
- `del, del0 (float)` — Additional parameters.

Class `josimpro.Param`

Represents a simulation parameter.

Constructors:

- `Param(exp: str = "", value: float = 0.0, parsed: bool = False)`

Attributes:

- `exp (str)` — The parameter expression.
- `value (float)` — The numerical value.
- `parsed (bool)` — Flag indicating if the parameter has been parsed.

Class `josimpro.ParsedNetlist`

Represents the parsed netlist with component data.

Constructors:

- `ParsedNetlist()` — Default constructor.

Attributes:

- `params` — Simulation parameters.
- `models` — Model definitions.
- `expandedLines` — List of expanded netlist lines.
- `components` — Read-only list of component pointers.

Class `josimpro.NetlistParser`

Class that parses a raw netlist into a ParsedNetlist.

Methods:

- `parse()` — Converts a raw netlist into a fully parsed netlist. Runs below methods.
- `parseParameters()` — Parses parameters from the raw netlist and stores them in the ParsedNetlist.
- `parseSimControl()` — Parses simulation control directives from the raw netlist.
- `parseSpreads()` — Parses spread commands from the raw netlist.
- `parseNoise()` — Parses noise parameters from the raw netlist.
- `parseModels()` — Parses model definitions from the raw netlist.
- `parseComponents()` — Parses and generates component objects from the raw netlist.

Class `josimpro.Matrix`

Represents the simulation matrix built from the input netlist.

Constructors:

- `Matrix()` — Default constructor.
- `Matrix(input: josimpro.Input)` — Constructs a matrix from an Input object.

Methods:

- `finalizeComponents()` — Finalize components, updating indices and wiring the matrix.
- `parse_mutual_inductances()` — Parses mutual inductances between inductors.
- `create_compressed_row_storage()` — Generates a compressed row storage representation.

Attributes:

- `non_zeros` — Non-zero elements.
- `rp` — Row pointer vector.
- `column_indices` — Column indices.

Class `josimpro.Results`

Contains raw simulation results.

Attributes:

- `size (int)` — Number of time steps.
- `time` — Raw pointer to the time array (use `get_time()`).
- `x` — Raw pointer to simulation data (use `get_x_full(matrix)`).

Helper Methods:

- `get_time()` — Returns the time array as a NumPy array.
- `get_x_full(matrix)` — Returns the entire 2D simulation data as a NumPy array.

Class `josimpro.Simulation`

Runs the simulation.

Constructors:

- `Simulation()` — Default constructor.
- `Simulation(matrix: josimpro.Matrix)` — Constructs a Simulation from a Matrix.

Methods:

- `simulate(matrix: josimpro.Matrix)` — Executes the simulation using the provided Matrix.

Attributes:

- `x` — The solution vector.
- `b` — The right-hand side vector.
- `isSimOK` — Simulation status flag.
- `simSize` — Simulation size (number of time steps).
- `simProgress` — Simulation progress.
- `results` — A `Results` object containing simulation data.

Enum `josimpro.PrintType`

Specifies print command types.

Values:

- `Voltage` — Voltage print.
- `Phase` — Phase print.
- `Current` — Current print.
- `Unknown` — Unknown print type.

Class `josimpro.Print`

Represents a print command.

Constructors:

- `Print()` — Default constructor.

Attributes:

- `name (str)` — The print command name.
- `type (josimpro.PrintType)` — The type of print.
- `idx (list[int])` — Index vector.
- `values (list[float])` — Output values.
- `printIndex (int)` — Print index.

Operators:

- `__eq__()` — Checks equality between two `Print` objects.

Class `josimpro.Output`

s Handles output formatting and printing.

Constructors:

- `Output()` — Default constructor.
- `Output(input: josimpro.Input, matrix: josimpro.Matrix)` — Constructs an Output object.

Methods:

- `format_time_vector(results: josimpro.Results, matrix: josimpro.Matrix, resultSize: int)` — Formats the output time vector.
- `compute_print_values_for_index(print_index: int, matrix: josimpro.Matrix)` — Computes the output values for a specific print command and stores them in its values vector.
- `compute_all_print_values(matrix: josimpro.Matrix)` — Computes the output values for all print commands and stores them in their respective values vectors.
- `print_output(simulation: josimpro.Simulation, matrix: josimpro.Matrix)` — Prints the output.
- `print_CSV_DAT(del: str, fname: str, printIndex: int = 0)` — Exports output in CSV/DAT format.
- `print_BIN(fname: str, printIndex: int = 0)` — Exports output in binary format.
- `print_RAW(fname: str, printIndex: int = 0)` — Exports output in RAW format.
- `print_COUT(printIndex: int = 0)` — Prints output to the console.

Attributes:

- `plist (list[josimpro.Print])` — List of print commands.
- `time (list[float])` — Output time vector.

Optimum – Verify, Margin, Yield Optimize

1. Overview

Optimum is JoSIM-Pro's analysis and design-tuning suite. It runs a circuit against a formal specification, then explores how robust it is to parameter variation and (optionally) steers parameters toward a better operating point.

Optimum supports four modes:

- **Verify** – Checks a simulation against a specification (SP-style spec file or CSV truth table).
- **Margin** – Measure per-parameter robustness (negative/positive scaling margins or full pass/-fail "pockets").
- **Yield** – Monte-Carlo yield estimation (and optional yield roll-off vs. spread).
- **Optimize** – Iteratively improve a parameter set (currently via a Center-of-Gravity-style stochastic search).

  Baseline requirement. All modes begin by verifying the baseline configuration. If the baseline does not verify, Optimum stops and reports the first failure (time and trace).

1.1 Inputs & Outputs at a Glance

- **Configuration**: a TOML file (this section documents all keys).
- **Circuit**: path to your netlist (`verify.circuit`).
- **Specification**: either an SP-style file or a numeric CSV (`verify.method`).
- **Results**: structured console output; optional text/CSV via `margin.output` and `yield.output`.

2. Quick Start

1. Create a configuration (e.g., `DFF_Example_config.toml`)

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

[verify]
method    = "spec_file"                   # or "csv_file"
file      = "specs/dff.sp"                # spec (SP or CSV)
circuit   = "circuits/dff.cir"            # netlist path
threshold              = 0.05             # global tolerance (see 4)
current_threshold      = 0.05
amplitude_threshold    = 0.50
slope_threshold        = 0.02
min_pulse_width        = 2.5e-12
min_stable_samples     = 3
merge_gap_threshold    = 1e-12

[margin]
subcircuit             = "TOP"            # restrict params to this subckt (
    optional)
accuracy               = 0.01             # step/tolerance
binary_search_steps    = 10
min_search             = 0.10
max_search             = 1.90
exhaustive             = false
pocket_search          = false
output                 = "margins.txt"
parameter_dependence = false              # advanced; see 3.2

# Pick parameters via regex OR enumerate explicitly (choose one approach):

[parameters]
search_string = "^(IBIAS|L_.*|C_.*)$"  # regex against parameter names in
    subcircuit

# or explicit:
# B1 = {}
# IB1 = {}
```

2. Run Optimum with your config as the input file. Use the same CLI style shown in the *Usage* chapter; pass the TOML as the input.

```
optimum DFF_Example_config.toml
```

3. Read results in the terminal; if you specified `margin.output` or `yield.output`, open those files for a machine-readable summary.

3. Configuration Structure

At the top level, set mode to one of `"verify" | "margin" | "yield" | "optimize"`. Each mode uses its own table(s) plus the shared `[verify]` block for baseline checking and spec parsing.

3.1 `[verify]` (common to all modes)

Key	Type	Default	Meaning
`method`	string	"spec_file"	"spec_file" for SP-style spec; "csv_file" for numeric CSV truth table.
`file`	string	—	Path to the spec file.
`circuit`	string	—	Path to the circuit netlist.
`threshold`	float	0.05	Global tolerance used by CSV verification and as base for SP thresholds.
`current_threshold`	float	0.05	Fraction of expected current used in SP checks (see §4.2).
`amplitude_threshold`	float	0.50	CSV event detection: fraction (of column max) to qualify pulses.
`slope_threshold`	float	0.02	CSV event detection: fraction (of max slope) defining "stable" segments.
`min_pulse_width`	float	2.5e-12	Minimum pulse duration (CSV).
`min_stable_samples`	int	3	Min consecutive stable samples (CSV).
`merge_gap_threshold`	float	1e-12	Merge adjacent pulses closer than this (CSV).

3.2 `[margin]` (Margin & Pocket Search)

Key	Type	Default	Meaning
`subcircuit`	string	-	If set, only parameters in this subcircuit are considered. If omitted, only top-level parameters are considered.
`accuracy`	float	0.01	Step size / convergence tolerance.
`binary_search_steps`	int	10	Safety cap for binary search iterations.
`min_search`	float	0.10	Minimum per-parameter scale tested.
`max_search`	float	1.90	Maximum per-parameter scale tested.
`exhaustive`	bool	false	If true, scan in steps of accuracy instead of binary search.
`pocket_search`	bool	false	If true, emit full pass/fail "pocket" maps (no margin summary).
`output`	string	-	Optional text file to write the summary/ASCII graph.
`parameter_dependence`	bool	false	Enable coupled-mode parameter dependence inside the circuit model (advanced).

3.3 `[yield]` (Yield & Roll-Off)

Key	Type	Default	Meaning
`num_samples`	int	100	Monte-Carlo samples per yield estimate.
`spread`	float	0.0	Global fractional spread; if zero, per-parameter defaults are used (10% of nominal).
`roll_off`	bool	false	If true, sweep spread from step_size → 1.0 and output a Spread vs Yield table.
`step_size`	float	0.1	Step when roll_off=true.
`output`	string	-	If set and roll_off=true, write CSV with columns Spread,Yield.

Sampling model: Each parameter is drawn from a normal distribution with σ = (spreadmap/3). Thus ±spreadmap roughly corresponds to the 3σ range. Negative draws are clamped to 0.

3.4 `[optimize]` (Stochastic Optimization)

Key	Type	Default	Meaning
`method`	string	"cog"	Center-of-Gravity–style heuristic (sample, verify, reflect failures, update center).
`spread`	float	0.5	Sampling amplitude around the current center (fractional).
`spread_decay`	float	0.75	Multiplicative shrink when stalled (clamped to [0.25, 0.99]).
`spread_grow`	float	1.05	Multiplicative expand on improvement (clamped to [1.00, 1.25]).
`samples`	int	10000	Samples per iteration.
`samples_max`	int	20000	Cap for adaptive upsampling.
`resample_every`	int	5	Refresh CRN epoch every N iterations.
`seed`	int	1234567	Base seed for reproducibility.
`trim_frac`	float	0.10	Two-sided trim fraction for trimmed mean.
`ema_alpha`	float	0.50	EMA blend: `new=alpha×cog + (1-alpha)×old`.
`sig_digits`	int	4	Significant digits when writing .PARAM lines. For manufacturability quantization.
`max_iterations`	int	500	Iteration cap.
`patience`	int	10	Early-stop counter for no-improvement.
`converge`	float	-1.0	Stop if improvement per iteration drops below this ($>$0 to enable).
`target`	float	-1.0	Stop when target objective is met (enable by setting $>$0).
`conv_k`	int	4	Consecutive iterations within `converge` to stop.
`output`	string	-	Resulting optimized circuit with optimized values back-annotated.

Note. The optimizer uses verification as its oracle. It samples parameter sets, keeps (and may reflect) passing samples to form a robust "center," and iterates until `converge` or `target` triggers, or `max_iterations` is reached.

3.5 `[parameters]`

The margin, yield and optimize methods require specific parameters to target. If none are provided it will attempt to use all the available parameters (in the `subcircuit`). Parameters can be specified in the following ways.

3.5.1 Regex selection

Use a regex string to match parameters in the circuit. Modified ECMAScript is the default grammar for this regex string.

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

3.5.2 Explicit enumeration

Specify exact parameters to be used. Brackets are reserved for future use. Will be used to limit certain values to minimum and maximum values i.e. bias currents.

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

Subcircuit scope. Parameter names are matched within `margin.subcircuit` (if set). If subcircuit is omitted, only parameters at the top level are considered.

Note. Not providing parameters is not recommended as this will significantly slow down each operation.

4. Verification Details

4.1 Spec File Options

- `method="spec_file"` — *SP-style* textual spec: first column is time; subsequent columns are labeled expectations. Labels should begin with a type letter:
  - L… — currents (uses `verify.current_threshold`).

- B… — phases (threshold scaled by 2π; SP phase expectations are interpreted in cycles and internally converted to radians).

For each time index, the simulator compares `true_value := trace(t)- trace(t0)` to the expected value in that column. A failure is raised if `|expected - true_value|` exceeds the relevant threshold. The value `t0` refers to the first time value in each column which is used as a startup offset taken at the specified time point.

```
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
2.5E-10    1   3     1
3E-10      3   3     1
3.5E-10    3   4     2
4.5E-10    3   5     2
6E-10      4   6     2
7E-10      6   7     3
8E-10      7   8     4
9.5E-10    7   10    5
```

- `method="csv_file"` — *numeric truth table*: first column is time; subsequent columns are numeric behaviors. Optimum:

  1. detects stable segments using `slope_threshold` and `min_stable_samples`,
  2. detects pulses using `amplitude_threshold` (relative to that column's max),
  3. merges near-adjacent pulses (`merge_gap_threshold`),
  4. refines/filters low-amplitude events, and
  5. compares simulated averages over each event window to CSV averages for that column.

  The allowed difference per event is `verify.threshold * average_transition[col]` (an adaptive tolerance derived from the column's event dynamics).

4.2 Choosing Tolerances

- **SP**: use `current_threshold` and the implied phase tolerance ($2\pi \times$ `verify.threshold`) as fractions of the expected value.
- **CSV**: start with `threshold=0.05`, `amplitude_threshold=0.5`, `slope_threshold=0.02`, then adjust based on how aggressively you want pulses merged and flat regions detected.

5. Margin Analysis

Goal. For each selected parameter *p*, find the largest negative/positive scale around 1.0 that still verifies:

- Negative margin: smallest `scale ∈ [min_search, 1.0]` that passes (reported as `(1-scale)×100%`).

- Positive margin: largest `scale ∈ [1.0, max_search]` that passes (reported as `(scale-1)×100%`).

Two search styles are available:

- **Binary search** (default): fast; controlled by `accuracy` and `binary_search_steps`.
- **Exhaustive** (`exhaustive=true`): step through the interval in increments of `accuracy`.

Set `pocket_search=true` to print full pass/fail maps instead of summarized margins. This relies on `parameter_dependence`.

5.1 Output

- **Margin (summary**

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

- **Pockets (maps)**

```
IBIAS : [  ** ** *** *** *****    |  ***** ** ** **  ]
        0.64, 0.72, 0.83, 0.91, 1.23, 1.37, ...
```

If `margin.output` is set, these are also written to the specified file.

5.2 Parameter Dependence

When circuits are parameterized, often dependencies appear. In a normal margin analysis (tolerance) we ignore dependencies and evaluate the circuit at the firs time evaluated parameter expressions, adjusting only one value at a time. With `parameter_dependence=true`, each parameter that depends on the parameter being adjusted is re-evaluated to maintain this dependency. This type of analysis is no longer tolerance based and becomes coupled. When `pocket_search=true`, pockets of operation can be observed due to this coupling (parameter dependency).

6. Yield Analysis

Goal. Estimate the pass fraction under random parameter variation.

- **Without roll-off**: draw `num_samples` parameter sets (normal distribution; σ = spread/3), verify each, and print Passes/Fails/Yield%.

- **With roll-off**: if `roll_off=true`, sweep spread from `step_size` to 1.0 and write a two-column CSV (Spread,Yield) when `yield.output` is set.

Example (roll-off CSV):

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

7. Optimization

Goal. Improve the circuit's critical margin by iteratively nudging parameters toward a robust "center." The optimizer samples around the current center, keeps passing samples, computes a trimmed/EMA-smoothed center-of-gravity (CoG), evaluates it via fresh Margin analysis, and accepts it only if the best critical margin (percent) improves.

7.1 Algorithm

Let `params` be the current center; `best_*` be the best-so-far.

1. Sampling (stable MC cloud)
   - Use common random numbers (CRNs) across up to `resample_every` iterations by seeding `Yield` with `seed + epoch`.
   - Draw `samples` parameter sets with normal noise (σ = spread/3 × nominal), clamp negatives to 0.
   - Keep the passing draws; reflect failing draws across the center and re-verify; keep reflect-passes.
2. Aggregate & smooth
   - For each parameter, compute a trimmed mean with two-sided `trim_frac`.
   - Blend with the previous center using EMA: `new = ema_alpha×cog + (1-ema_alpha)×old`.
3. Evaluate
   - Run Margin on the EMA'd center (with original baseline for other params) to get the current critical margin (percent).
   - If current $>$ best, accept and:
     - update `best_*`, reset patience, grow spread (`×spread_grow`, bounded by `[min_spread, 1]`), and reset `samples` to its starting value.
     - if `output` is set, write an updated circuit file (see §7.3).
   - Otherwise:

  - increment patience; occasionally decay spread and upsample on a schedule tied to patience (about every $\sim \frac{1}{3}$ and $\sim \frac{1}{2}$ of the patience window).

4. Stopping The loop ends when any of the following fires:
   - `patience` exhausted → **Early stopping (patience)**.
   - `converge>0` and the p.p. gap to best is $\leq$ `converge` for `conv_k` consecutive iterations → **Convergence reached**.
   - `target≥0` and best critical margin $\geq$ target (after unit normalization) → **Target reached**.
   - `iteration == max_iterations` → **Hard cap**.

   Degenerate sampling guard. If very few/no passing samples are found in an iteration, the run is treated as "no progress": patience++, decay spread toward `min_spread`, possibly upsample, and continue.

7.2 Tuning guide

- Start with `spread = 0.4 - 0.6`. If early iterations stall, either increase `samples` or loosen `trim_frac` (e.g., 0.05).
- For noisy landscapes, consider more inertia: lower `ema_alpha` (0.3–0.4) and slightly increase samples.
- Use `converge` in percentage points (e.g., `converge=0.2` with `conv_k=4`) to terminate once improvements are negligible.
- If you want controlled exploration after an improvement, keep `spread_grow` modest (1.02–1.06).
- Bump `resample_every` if you want more independent clouds; lower it to leverage CRNs for variance reduction across adjacent steps.

7.3 Writing the optimized circuit (`optimize.output`)

When you get a new best (or on stop), and output is set, Optimum emits a modified netlist where:

1. **S** = set of parameters the optimizer touched.
2. It computes transitive dependents of **S** from the JoSIM-Pro reverse-dependency graph, restricted to the same subcircuit (if any) and to names that actually have `.PARAM` lines in the file.
3. It overwrites:
   - **S** with the optimized numeric values (formatted with `sig_digits`; compact scientific/fixed with trailing zeros trimmed).
   - **Dependents S** with the baseline numeric values (locking them to the original numerics).

- Everything else remains unchanged (including expressions).

This preserves expressions outside the optimized/locked set while yielding a reproducible best configuration bounded to the chosen scope.

8. Example

We provide a DFF example to showcase the use of this tool. Parts of the required files have already been shown in this chapter. The rest are provided in the `examples` directory:

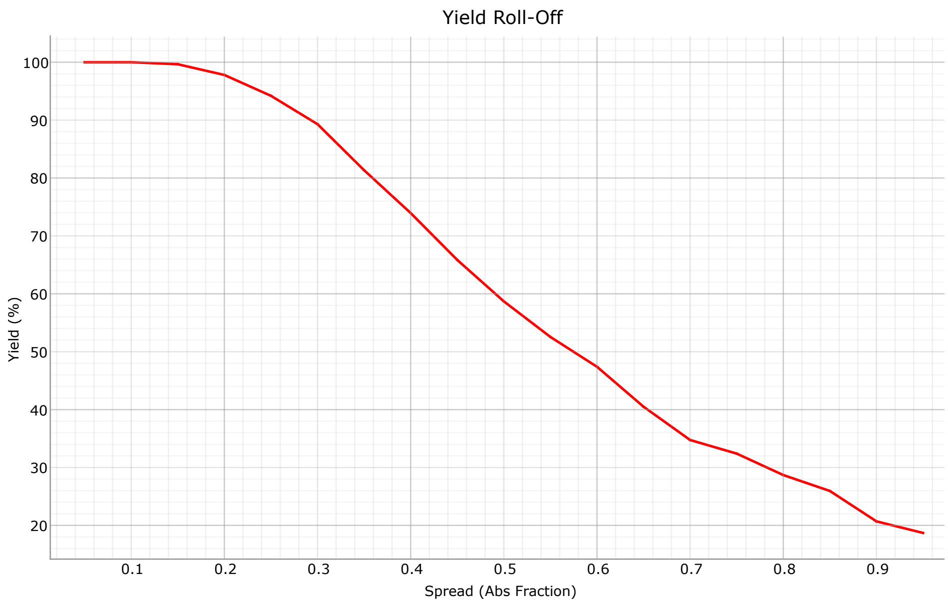
```
#mode="verify"
mode="margin"
#mode="yield"
#mode="optimize"

[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+)$"
#B1={}
#B2={}
#B3={}
#B4={}
#B5={}
#B6={}
#B7={}

#IB1={}
#IB2={}
#IB3={}
#IB4={}

#L1={}
#L2={}
#L3={}
#L4={}
#L5={}
#L6={}
#L7={}


[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
```

This produces the margin output:

```
Optimum: Verify, Margin, Yield and Optimize
Copyright (C) 2025 SUN Magnetics
v1.0.250829

Input:  "DFF_Example_config.toml"

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
B6 :  90.00[   *************************|*************************   ] 90.00
B7 :  44.30[                ************|*************************   ] 88.59
IB1:  90.00[   *************************|*************************   ] 90.00
IB2:  39.37[                 ***********|**************              ] 49.92
IB3:  90.00[   *************************|*************************   ] 90.00
IB4:  83.67[     ***********************|******************          ] 65.39
L1 :  90.00[   *************************|*************************   ] 90.00
L2 :  90.00[   *************************|*************************   ] 90.00
L3 :  90.00[   *************************|*************************   ] 90.00
L4 :  90.00[   *************************|*************************   ] 90.00
L5 :  90.00[   *************************|*************************   ] 90.00
L6 :  90.00[   *************************|*************************   ] 90.00
L7 :  90.00[   *************************|*************************   ] 90.00
Critical margin: 14.06% ['B5+']
Average margin: 72.70%
```

A yield analysis with roll-off for this DFF circuit produces the following curve:



Performing an optimization on the circuit potentially yields the following improved margins. Since optimization is performed using gaussian spread and sampling of a Monte-Carlo cloud it's not deterministic and will not always produce the same optimized results.

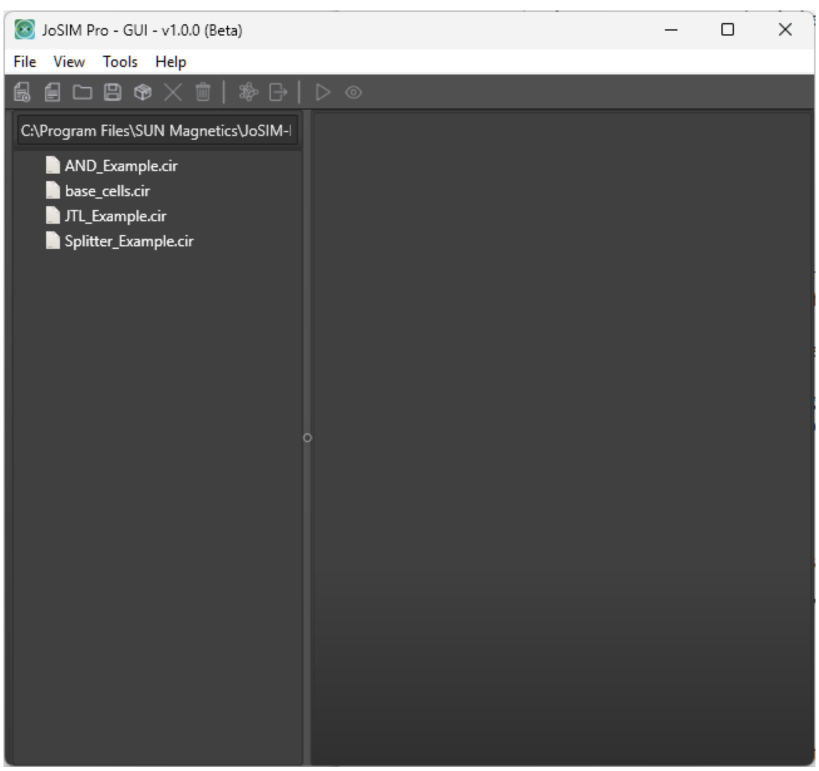
```
Optimum: Verify, Margin, Yield and Optimize
Copyright (C) 2025 SUN Magnetics
v1.0.250829

Input:  "DFF_Example_config.toml"

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%
B1 :  90.00[   *************************|*************************   ] 90.00
B2 :  90.00[   *************************|**************              ] 49.22
B3 :  49.92[              **************|***********                 ] 38.67
B4 :  90.00[   *************************|**********                  ] 37.27
B5 :  90.00[   *************************|*************               ] 47.81
B6 :  90.00[   *************************|*************************   ] 90.00
B7 :  45.00[               *************|****************            ] 56.25
IB1:  90.00[   *************************|*************************   ] 90.00
IB2:  37.27[                  **********|*************               ] 47.11
IB3:  90.00[   *************************|*************************   ] 90.00
IB4:  90.00[   *************************|**********************      ] 80.16
L1 :  90.00[   *************************|*************************   ] 90.00
L2 :  90.00[   *************************|*************************   ] 90.00
L3 :  90.00[   *************************|*************************   ] 90.00
L4 :  90.00[   *************************|*************************   ] 90.00
L5 :  90.00[   *************************|*************************   ] 90.00
L6 :  90.00[   *************************|*************************   ] 90.00
L7 :  90.00[   *************************|*************************   ] 90.00
Critical margin: 37.27% ['B4+']
Average margin: 74.80%
```

Graphical User Interface (GUI)

From JoSIM-Pro v1.4 onwards, a GUI is included to aid the user in using the software.

1. Overview

The intention of this GUI is to create an all encompassing simulation environment where the user can create, simulate and analyze their design without the need for 3rd party external tools. This requires a certain feature set which is documented below.



2. Toolbar

At the top of the GUI window a toolbar with most common actions will appear. These actions are either enabled or disabled depending on the state of the GUI.

3. File Browser

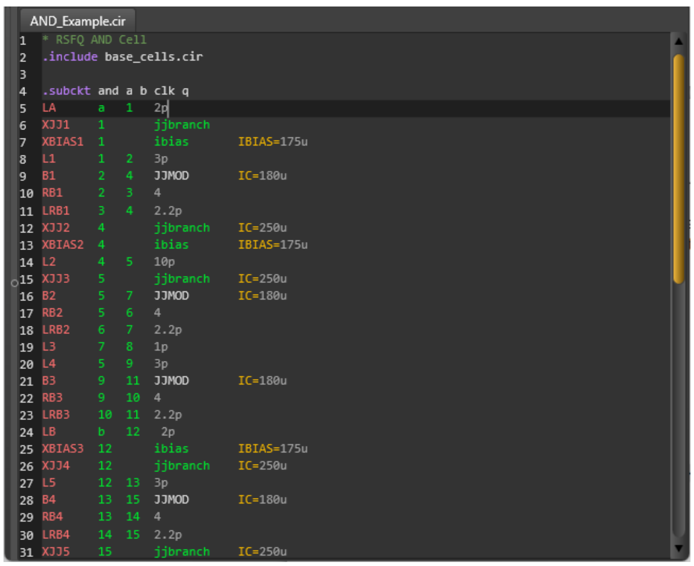
A file browser has been added to allow the user to open a design via the `File→Open File` command. The files that can be opened have been filtered to only include netlists.

The file browser is also available in the form of a collapse-able file tree dialog to the left of the window. This allows for quick access to the files in the current working directory. The directory of this file tree dialog can be altered with `File→Open Folder` or by the address bar at the top of the dialog.

4. Text Editor

Once a netlist has been opened, the contents is displayed in a syntax highlighted text editor window to the right of the file tree dialog. This text editor supports most basic file editing features as well as a few advanced features such as similar label highlighting to quickly see which nodes are connected or where else a label appears. Additionally, this editor has a find and replace functionality similar to most basic text editors. Some basic features include:

- Line numbering
- Current line indication
- Syntax highlighting
- Find & Replace
- Copy, Paste, Cut and Delete
- New file creations through `File→New` or the New File button in the toolbar.



Planned future improvements:

- Sanity checks – Missing nodes, duplicate component names, unclosed subcircuits, invalid syntax.
- Component suggestions, code completion.
- Static circuit analysis.

5. Netlist Visualizer

The GUI allows the user to visualize the open netlist. This will place the components found within the circuit in the most appealing way a machine can and create the relevant connections between them.

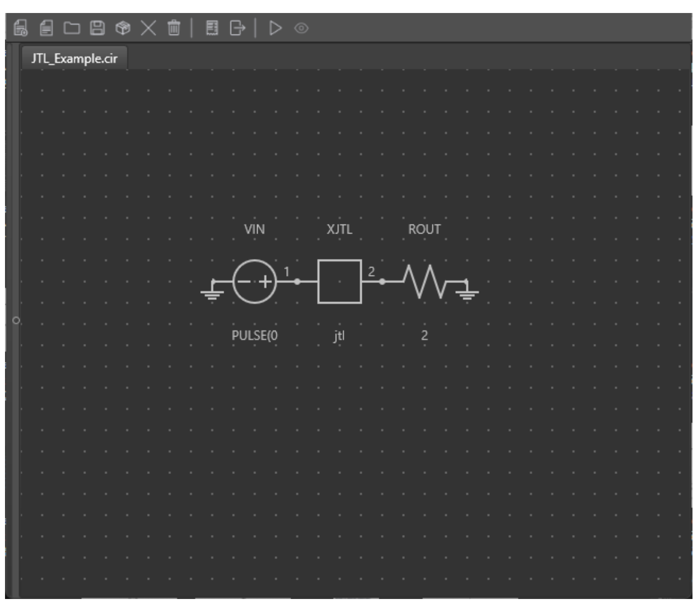
Components in this visualized netlist can then be dragged around and rotated to mimic the designers' intended appearance. In essence this will become the schematic view for the circuit.

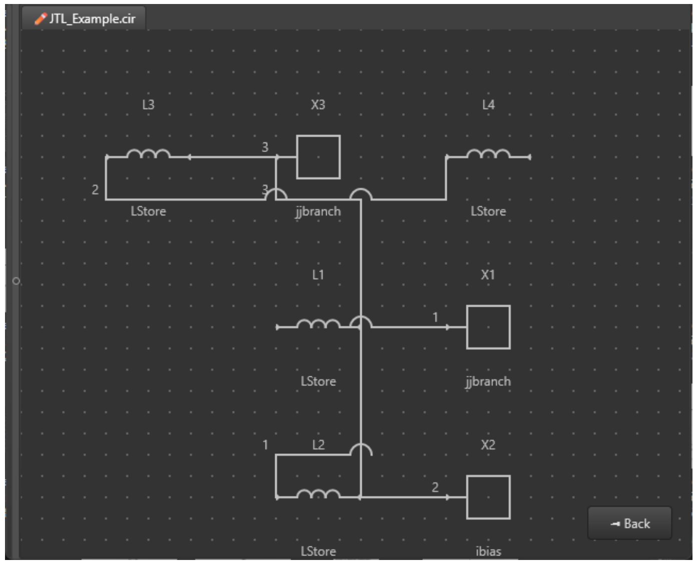
Subcircuits are visualized as blocks with I/O pins and the subcircuit label and name in the center. These blocks can be double clicked to enter and visualize the referenced subcircuit. Multiple nested subcircuits can continually be entered with a button appearing in the right-hand corner to move back to the surface.

Visualized netlists can be saved. This will append a commented hash to the bottom of the netlist file. This hash contains the encoded positions and orientations of all the components in the circuit. Upone visualization the hash is read, if present, and will recall the users' set component positions. If a component is added that is not present in the hash, a strategic location for this component will be selected on new visualization. Saving will overwrite the existing hash with a new hash.

Subcircuits visualized and saved will append an extra hash (per subcircuit) for unique visualizations of subcircuits pertaining to the current design.

Visualizations can also be exported in the form of portabnle network graphics (PNG) or scalar vector graphics (SVG). This allows for visualized netlists to be effortlessly re-used in presentations.





Planned future improvements:

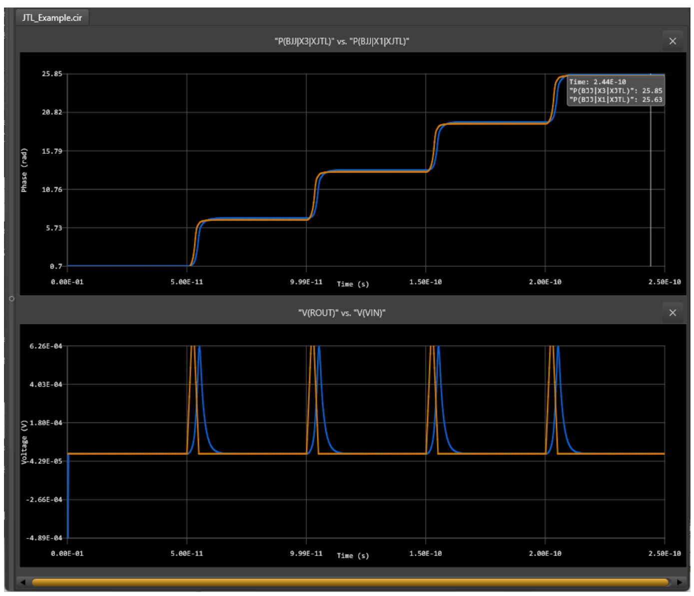
- In place component creation (schematic editor)
- Component value editing
- Connection anchoring

6. Simulation

A netlist can be simulated from text or visualized point of view. This will simulate the netlist using the specified control commands and produce the results in a waveform visualizer in the same tab. Output traces will be best placed for optimal visibility, enabling a scroll bar to view off screen traces.

Traces can be hidden (closed) and made visible again using the visibility (eye) toolbar button.

Traces can re-arranged by click and drag functionality, and additionally combined by dropping traces on top of another. This hides the two original traces which can then be made visible again via visibility settings.



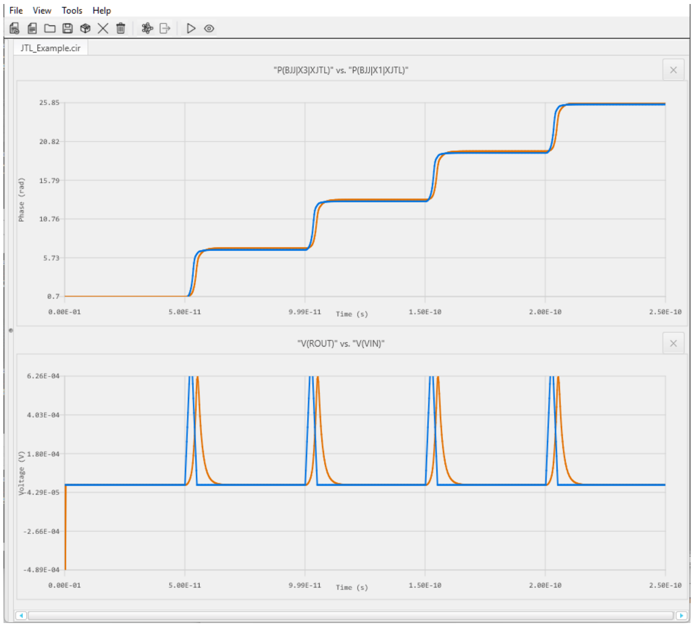
Planned future improvements:

- Saving outputs to file (PNG, SVG and CSV).
- Trace customization (colour, line width).

7. Settings

Settings are stored in a configuration file in the users' directory and will be read upon start-up. Settings include:

- Working directory
- Window Theme
- Grid style (netlist visualization)
- Grid size (netlist visualization)
- Grid on/off (trace visualization)



Troubleshooting

In this chapter, we will go over common errors and warnings generated by JoSIM-Pro, their possible causes, and recommended solutions. These messages are meant to assist you in resolving issues during simulation setup, input handling, and execution.

1. Utility Functions

- **Error**: `String (value) cannot be converted to double.`
  Cause: JoSIM-Pro is trying to convert a string to a numerical value but has failed. This could happen if a string that is expected to be a number contains non-numeric characters.
  Solution: Double-check the input value that is supposed to be numeric and ensure it is formatted correctly (e.g., avoid characters in numeric fields). If this seems like an internal issue, contact support.
- **Error**: `Cannot calculate mean of an empty vector.`
  Cause: A function is trying to calculate the mean of an empty set of values.
  Solution: Ensure that you are passing a valid set of values to functions that require numeric input.
- **Error**: `No values to calculate the mean from the given start index.`
  Cause: The function is attempting to calculate a mean, but there are no values beyond the specified starting index.
  Solution: Make sure the index is valid and that values exist after the starting point.

2. Command-line Arguments

- **Error**: `Unknown switch: switch_name`
  Cause: The command-line switch provided is unrecognized by JoSIM-Pro.
  Solution: Refer to the documentation to check the correct syntax and available command-line arguments.
- **Error**: `Input file and output file names are the same. Operation canceled to prevent data loss.`
  Cause: The input and output files have been given the same name.
  Solution: Ensure that the output file has a different name to avoid overwriting the input file.

3. Components

- **Error**: `Unknown parameter param found in component label.`
  Cause: A component includes an unrecognized parameter.
  Solution: Check the component syntax and verify that all parameters are correctly defined in the netlist.

- **Error**: `Duplicate component key.name detected.`
  Cause: More than one component with the same label or name has been defined.
  Solution: Ensure that component labels are unique throughout the netlist to avoid conflicts.

- **Error**: `Missing model for label.`
  Cause: The component has not been assigned a model.
  Solution: Define the required model and ensure it is referenced correctly.

- **Error**: `Invalid transmission line definition found for label.`
  Cause: The syntax or parameters of a transmission line definition are incorrect.
  Solution: Check the transmission line syntax and parameters, ensuring they are properly defined.

- **Warning**: `No area or Ic scalar specified for label. Using unity scalar.`
  Cause: No scaling factor (area or critical current) has been defined for the Josephson Junction. JoSIM-Pro will use a default scalar of 1.
  Solution: Specify a scalar or leave it if the default is acceptable.

4. Functions

- **Error**: `Unsupported function type: function_name`
  Cause: JoSIM-Pro has encountered a function type that it doesn't support.
  Solution: Ensure that the function you are trying to use is supported and correctly defined.

- **Error**: `Unexpected number of tokens. Expected pairs of (time, amplitude).`
  Cause: The input to a time-varying function is incorrectly formatted.
  Solution: Ensure that pairs of time and amplitude values are correctly provided for the function.

- **Error**: `Timestep and amplitude mismatch. Please ensure the correct syntax is followed.`
  Cause: The number of timesteps does not match the number of amplitude values provided.
  Solution: Ensure that each time value has a corresponding amplitude value.

- **Error**: `Too few values to form pulse function.`
  Cause: The pulse function has an insufficient number of values.
  Solution: Check the pulse function definition and ensure it includes all required parameters.

- **Error**: `Too few values to form sin function.`
  Cause: The sine function has an insufficient number of values.
  Solution: Verify that the sine function is correctly defined with the proper values.

- **Error**: `Too few values to form custom function.`
  Cause: The custom waveform function has too few values.
  Solution: Ensure that all necessary values are provided in the custom function.

- **Error**: `The file filepath could not be found.`
  Cause: JoSIM-Pro is unable to locate the file specified.
  Solution: Ensure the file path is correct, the file exists, and you have permission to access it.

- **Error**: `Too few values to form noise function.`
  Cause: The noise function has an insufficient number of values.
  Solution: Ensure the noise function is defined with the correct values.

- **Error**: `Too few values to form exponential function.`
  Cause: The exponential function has an insufficient number of values.
  Solution: Verify the exponential function definition.

5. Input

- **Error**: `The file inputFile could not be found.`
  Cause: JoSIM-Pro cannot find the input file specified.
  Solution: Double-check the file path and ensure the input file is available and accessible.

- **Error**: `Invalid .include statement: line.`
  Cause: The `.include` command has incorrect syntax.
  Solution: Verify the syntax of the `.include` statement, ensuring the file path is correct and accessible.

- **Error**: `Invalid subcircuit line: line.`
  Cause: The subcircuit definition is invalid or has incorrect syntax.
  Solution: Ensure that subcircuits are properly defined, following correct syntax.

- **Error**: `Missing end of subcircuit subc.`
  Cause: A subcircuit is not properly closed with an `.ends` statement.
  Solution: Ensure that all subcircuits are correctly closed with the appropriate `.ends` command.

- **Warning**: `Cyclic file include. The file filepath has already been included. The included file will be ignored.`
  Cause: A file has been included recursively, which can lead to errors.
  Solution: Remove any cyclic references by ensuring a file is not included more than once.

6. Matrix

- **Error**: `Attempting to set simulation step size larger than simulation stop time.`
  Cause: The step size provided is greater than the total simulation time.
  Solution: Ensure that the simulation step size is smaller than the stop time.

- **Error**: `Attempting to start storing output values beyond the stop time of the simulation.`
  Cause: JoSIM-Pro is trying to store values after the simulation end time.
  Solution: Adjust the time frame of your output requests.

- **Error**: `Invalid parameter definition found: parameter.`
  Cause: A parameter is defined incorrectly.
  Solution: Check the syntax for defining parameters and ensure it adheres to the rules.

- **Error**: `Unparseable parameters found.`
  Cause: Parameters contain invalid characters or values.
  Solution: Ensure that all parameters are valid and parsable.

- **Error**: `Invalid spread command defined.`
  Cause: The `spread` command is missing values or defined incorrectly.
  Solution: Check the `spread` command syntax.

- **Error**: `Unknown subcircuit specified.`
  Cause: A subcircuit is referenced that does not exist.
  Solution: Ensure all subcircuits are defined within the netlist scope.

- **Error**: `Insufficient nodes specified for subcircuit.`
  Cause: The subcircuit call does not have enough nodes specified for its inputs and outputs.
  Solution: Check the subcircuit definition and provide the correct number of nodes.

- **Error**: `Cyclic subcircuit found.`
  Cause: A subcircuit is calling itself, creating a loop.
  Solution: Avoid cyclic subcircuit calls as they cause infinite loops in simulation.

- **Error**: `Invalid mutual inductance line specified.`
  Cause: Mutual inductance is defined incorrectly.
  Solution: Ensure correct syntax is followed for mutual inductance.

- **Error**: `Unable to locate inductors in mutual inductance.`
  Cause: One or both inductors involved in mutual inductance are missing.
  Solution: Ensure all inductors are defined in the netlist.

7. Model

- **Error**: `Invalid model line: Missing parameters.`
  Cause: The model line is missing essential parameters.
  Solution: Check the model syntax and add the required parameters.

- **Error**: `Unknown model type.`
  Cause: The specified model type is unsupported by JoSIM-Pro.
  Solution: Ensure you are using a valid model type.

- **Error**: `Unknown model parameter.`
  Cause: The model has an unrecognized parameter.
  Solution: Verify the parameter in the model definition.

8. Parameters

- **Error**: `Mismatched parentheses in expression.`
  Cause: The parentheses in the parameter expression are unbalanced.
  Solution: Ensure proper syntax for parentheses in expressions.
- **Error**: `Invalid RPN deduced from expression.`
  Cause: The reverse polish notation (RPN) derived from the expression is invalid.
  Solution: This may be a bug. Contact support.
- **Warning**: `Parameter already defined. Overwriting.`
  Cause: A parameter is being redefined.
  Solution: Check if the parameter needs to be defined multiple times.

9. Simulation

- **Error**: `Simulation failed. Matrix has no solution.`
  Cause: The matrix equation derived from the circuit has no solution.
  Solution: Contact support, as this is most likely a bug.

10. Licensing

- **Error**: `No license detected.`
  Cause: The license file is missing or invalid.
  Solution: Contact support@sun-magnetics.com for assistance.
- **Error**: `Invalid license detected.`
  Cause: The license file cannot be validated.
  Solution: Contact support to resolve the licensing issue.

11. Output

- **Warning**: `The component requested for output does not exist.`
  Cause: The requested output component is not present in the circuit.
  Solution: Verify that the output request references valid circuit components.