

PSCAN2

Superconductor Circuit Simulator

Manual

rev. 1.5

07/13/2022

Content:

1) Introduction	p. 3
2) Circuit netlist	p. 5
3) Element models	p. 8
4) Circuit parameters definition files	p. 11
5) Python interface	p. 17
6) PSCAN2 Graphical User Interface (GUI)	p. 20
7) References	p. 26
8) Appendix A. PSCAN units	p. 27
9) Appendix B. Default netlist parameters	p.28
10) Appendix C. Using PSCAN Python-defined parameters	p.29

Introduction

PSCAN2 is a superconductor circuit simulator that came from ancestry of a long line of superconductor simulators since 1980-s. The development of PSCAN2 is based on experience obtained from the development and exploitation of the previous PSCAN (Portable Superconductor Circuit ANalyzer) program [1, 2] and JULIA superconductor simulator.

PSCAN2 comprises a Python module (with accessible source files) and a KLU-library-based linear equation system solver. The Python part of the program performs netlist, parameters, and HDL parsing and prepares the sparse matrix for the solver. On a higher level, there are functions for running circuit simulation, calculating margins on circuit parameters, and optimizing the circuit. All these functions are integrated under the PyQt-based GUI module.

Warning: PSCAN2 is optimized for simulating SFQ circuits with dc bias currents. If you need to simulate ac-biased circuits, - **set `psglobals.OptimizeSimulation` parameter to False (see Appendix C)**

How to start?

Download PSCAN2 distribution for Windows10 or Linux from www.pscan2sim.org with the detailed installation instructions.

PSCAN2 functions can be called directly from the Python shell by execution of Python code (like IV curve simulation or batch optimization, please, see examples)

```
python <program.py> <netlist_name>
```

or by means of PSCAN2 GUI module

```
python -m pscan2.gui <netlist_name>
```

The last argument is the name of the root cell in the schematics. The schematics Spice netlist should be in the file named as the root cell and with extension “.cir” (<netlist_name>.cir).

Expression syntax.

PSCAN2 operates with

- A numeric constant (e.g., 1.21). A real number in any format supported by Python.
- A string constant (e.g., “text”). A text string in parentheses.
- An identifier (e.g., xj1). A variable defined at initializing.
- A user-defined function (in the executable Python file).
- All functions from Python’s math module.

Table 1. Operators available in PSCAN2

Operator	Type	Function
"_"	unary arithmetic	Negate
"not" or "!"	unary logical	NOT
"*"	binary arithmetic	Multiply
"/"	binary arithmetic	Divide
"+"	binary arithmetic	Add
"-"	binary arithmetic	Subtract
"<"	Logical	less than
"<="	Logical	less than or equal to
">"	Logical	greater than
">="	Logical	greater or equal
"eq" or "=="	Logical	Equal
"ne" or "!="	Logical	not equal
"and" or "&&"	Logical	AND
"or" or " "	Logical	OR

PSCAN2 has the following built-in functions:

p(<node>/<element>) – returns superconductor phase on the circuit node or phase drop across the element at the current time step

v(<node>/<element>) – returns voltage drop between the circuit node and the ground or across the element at the current time step

i(<element>) – returns value of the current through the element at the current time step

psfq(<period>, <duration>, <start>) – a function that generates a periodical 2π jump with duration <duration> at the time moment <start> with the period <period>.. It is usually applied to a phase generator for defining input SFQ signals.

n(<jjname>) – number of flux quanta passed through the Josephson junction or the phase source, i.e., the normalized by 2π phase drop.

pyeval(<python expression>) – executes a python expression in the text string <python expression>. (see Appendix C for details)

PSCAN2 also supports all functions from Python's math module (e.g., $I1=a1*\sin(0.01*tcurr)$)

Circuit netlist

PSCAN2 imports the circuit schematics as a standard SPICE netlist format supported by the majority of schematic editors. As an example, Fig. 1 shows a simple RSFQ circuit and its netlist. This is a non-hierarchical (“flat”) circuit, so, no sub-circuit is present here. PSCAN2 parser is case insensitive.

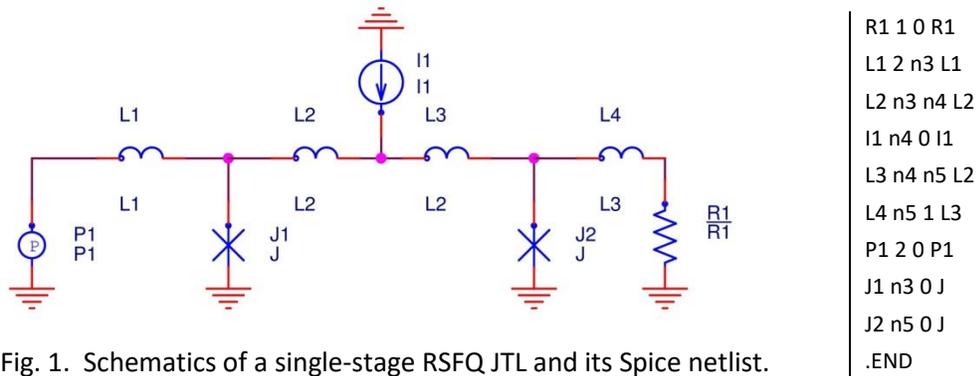


Fig. 1. Schematics of a single-stage RSFQ JTL and its Spice netlist.

Each line, describing the element, has a format:

<elements' _name> <node_1> ...<node_n> <elements' _model>

Element's type defined by the first letter of its name:

'L' – an inductor

'R' – a resistor

'C' – a capacitor

'I' – a current source

'P' – a phase source

'U' – a voltage source

'J' – a Josephson junction

'M' – a special element called “mutual inductance”

'X' – a subcircuit in hierarchical schematics

<node_1> ...<node_n> are the names of the nodes, the element is connected to. The number of nodes corresponds to the number of terminals (2 in most of the cases). Node names are strings and can be either numbers or text identifiers. The node named “0” (zero) is a designated node, representing connection to the ground.

An elements' model is a simulation model describing phase/voltage-current relation at the nodes of the element, i.e., a numerical simulation of the circuit. The majority of circuit linear element models, such as inductance, capacitance, resistance, etc., are described simply by a single parameter (a variable) that may be represented as an arithmetic expression. A more complex element (like a Josephson junction or a transformer) requires a specially written model (a function) with multiple parameters. Please, see Appendix A for detailed models description.

For example, the Spice record

*L2 n3 n4 L2*XL2*XL*

Describes inductor named L2 inserted between nodes “n3” and “n4” with its inductance value equal to the product of three parameters (L2, XL2 and XL). Such representation of the inductance value is very convenient for optimization procedure, as we shall discuss it later. For convenience, PSCAN2 substitutes in element L2 model description “?” with the construct “L2*XL2*XL”. The same goes with other single-parameter primitives. See Appendix B for more details.

The record

*J1 n3 0 rsj(J1*XJ1*XJ, 1.0, 1.0)*

represents a Josephson junction J1 connected between node “n3” and the ground (0) and described by RSJ (Mc Camber-Stewart) model (see p. 8). Its critical current is the product of three parameters (J1, XJ1 and XJ), its normal resistance and capacitance are equal to 1.0 (a nominally shunted JJ, i.e., with $\beta_c=1.0$).

And, finally, record

M1 0 llm(L1, L2, Lm12)

Indicates that the mutual inductance between inductors L1 and L2 is equal to parameter Lm12. The name and the node of this specific element is not relevant for the netlist, as long as it starts with “M” and the schematic editor accepts them.

Hierarchical netlist

PSCAN2 fully supports hierarchical SPICE format. Name of the hierarchical element in the netlist starts with the letter ‘X’. The record format is

X<name> <node_1> ... <node_n> <circuit_name>

Here,

<name> is the name of the subcircuit instance,

<node_n> connected nodes, in accordance to cells’ terminals (pins)

<circuit_name> is the subcircuit name.

Fig. 2 shows hierarchical schematics from the example of inverter cell NOT (you can download file testnot.zip with this example from www.pscan2sim.org). All element symbols in the picture are schematics-editor-dependent (Cadence/OrCad in this case) and are not relevant for the netlist. Every hierarchical object in the circuit has the name of its instance in the value field.

Schematics of all subcircuits for this test structure are in Fig. 3. The resulting netlist file is in the testnot.zip example. The hierarchy depth is not limited (i.e., Fig. 3b shows subcircuit STDIN, that, in turn, uses two subcircuits JTL1). Subcircuits JTL (Fig. 3c) and JTL1 (Fig. 3d) have a minor difference in a single inductor L3. Even if their schematics was identical, but having different parameter, they should be named differently, because circuit name identifies the file with parameters for this circuit.

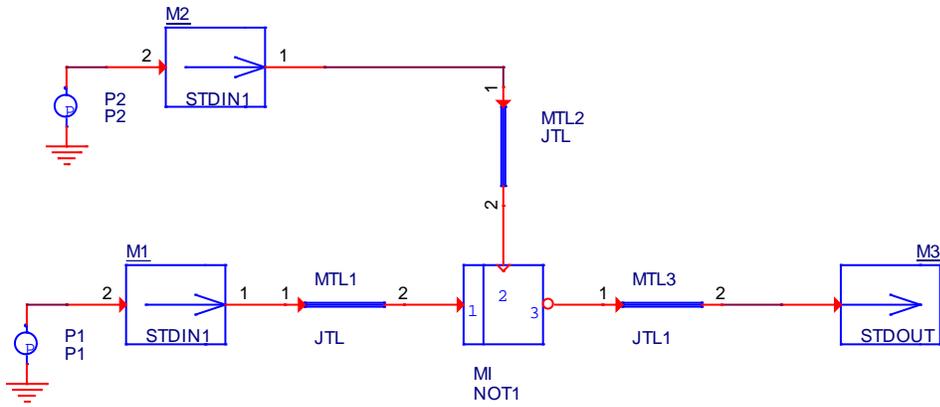


Fig. 2. Hierarchical circuit TESTNOT consisting of the following sub-circuits: NOT1, JTL, JTL1, and STDIN1

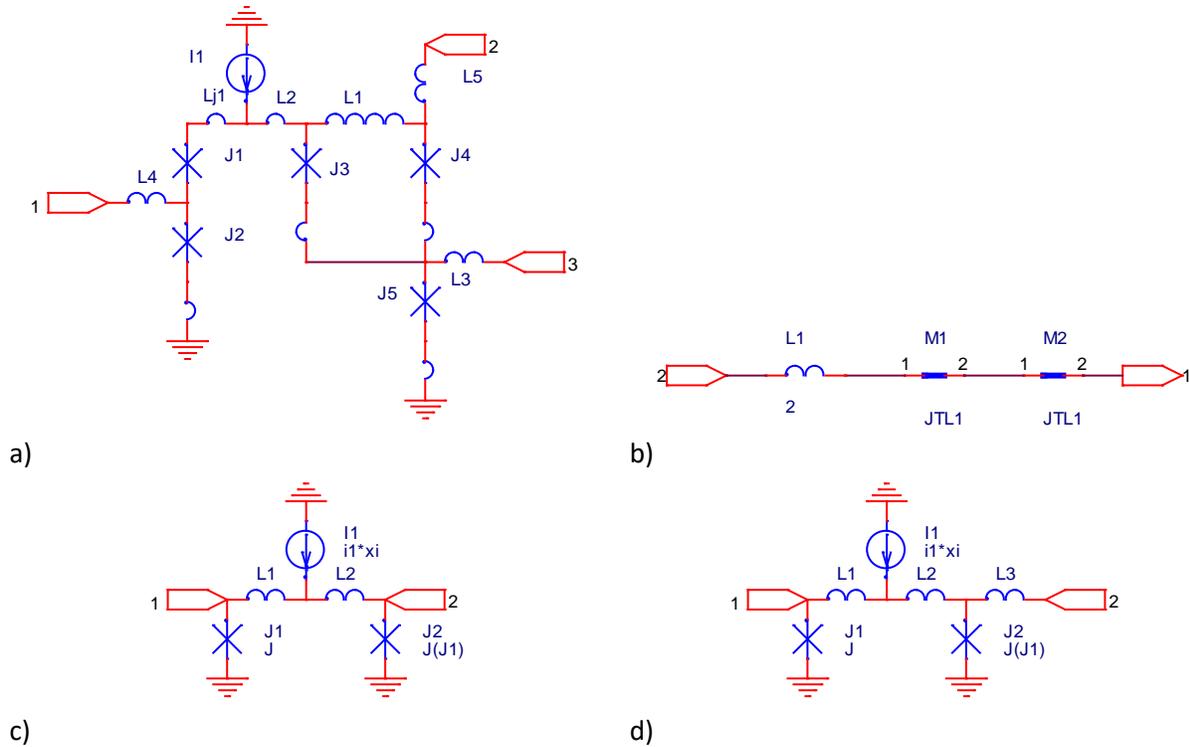


Fig. 3. Sub-cells of circuit TESTNOT: a) NOT1, b) STDIN1, c) JTL, and d) JTL1

Element models

The current version of PSCAN2 operates with a quite extensive set of superconductor element models. Here is the list of basic PSCAN elements and their numerical description:

Inductor

$I_{i,j} = \frac{\phi_i - \phi_j}{L}$, where ϕ_i and ϕ_j are values of superconductor phase at nodes "i" and "j" correspondently.

Resistor

$I_{i,j} = \frac{\dot{\phi}_i - \dot{\phi}_j}{R}$, where $\dot{\phi}_i$ and $\dot{\phi}_j$ are values of the first time derivative of superconductor phase (voltage) at nodes "i" and "j" correspondently.

Capacitor

$I_{i,j} = C * (\ddot{\phi}_i - \ddot{\phi}_j)$, where $\ddot{\phi}_i$ and $\ddot{\phi}_j$ are values of second time derivative of superconductor phase at nodes "i" and "j" correspondently.

Mutual inductance model

LLM(L1, L2, Lm12), where L1 and L2 is the name of inductors having magnetic coupling Lm12 between them. Assuming, that the inductors values are L1 and L2 correspondently, the phase-current relation is

$$\phi_i - \phi_j = L1 \cdot I_{i,j} + Lm12 \cdot I_{k,l}$$

$$\phi_k - \phi_l = L2 \cdot I_{k,l} + Lm12 \cdot I_{i,j}$$

Inductances L1 and L2 are connected to nodes i & j and k & l respectively. $I_{i,j}$ and $I_{k,l}$ are currents from node "i" to node "j" and from node "k" to node "l".

Josephson junction models

RSJ model

A linear resistive Josephson junction model, also known as Mc Cumber-Stewart model [3].

$rsj(I_c, R_n, C)$

where

" I_c " is junctions' critical current,

" R_n " is its normal resistance, and

" C " is its capacitance.

The Mc Cumber-Stewart relation [3] establishes the current-phase relation in RSJ model.

$$I_{i,j} = I_c \cdot \sin(\phi_i - \phi_j) + \frac{\dot{\phi}_i - \dot{\phi}_j}{R_n} + C \cdot (\ddot{\phi}_i - \ddot{\phi}_j)$$

For convenience, there are a few “function closures” (substitutions) for RSJ model defined in PSCAN2 parser:

$VB(I_c, V_c, b_c) \Rightarrow RSJ(I_c, V_c/I_c, b_c \cdot I_c/V_c^2)$, where V_c is $I_c R_n$ product and b_c is a McCumber parameter;
 $J(J1) \Rightarrow RSJ(J1 \cdot X_j, 1.0/J1, J1)$, where $J1$ is a critical current parameter, $X_j = 1.0$ is a global parameter describing a critical current density deviation. The function assumes $I_c \cdot R_n = 1.0$ and $b_c = 1.0$ (i.e., a nominally shunted Josephson junction);

$JJ(J1, X_j1) \Rightarrow RSJ(J1 \cdot X_j1 \cdot X_j, 1.0/J1, J1)$, where $J1$ is a critical current parameter, $X_j1 = 1.0$ is a hierarchical (EXTERNAL) parameter describing individual JJ critical current deviation (this is very handy at circuit optimization), $X_j = 1.0$ is a global parameter describing a critical current density deviation. The function assumes that $I_c \cdot R_n = 1.0$ and $b_c = 1.0$ (i.e., a nominally shunted Josephson junction);

$JJJ(J1, X_j1, V_j1) \Rightarrow RSJ(J1 \cdot X_j1 \cdot X_j, V_j1/J1, J1 \cdot X_j1)$, where $J1$ is a critical current parameter, $X_j1 = 1.0$ is a hierarchical (external) parameter describing individual JJ critical current deviation (i.e. area), V_j1 is a $I_c \cdot R_n$ product factor, and $X_j = 1.0$ is a global parameter describing a critical current density deviation.

RSJN model of Josephson junction

A non-linear resistive Josephson junction model. This model describes non-linear behavior of a Josephson junction better than its linear predecessor does.

The notation is

$rsjn(I_c, R_n, V_g, n, C)$

“ I_c ” is junctions’ critical current,

“ R_n ” is its normal resistance,

“ V_g ” is superconductor gap voltage,

“ C ” is its capacitance, and

“ n ” is a polynomial approximation degree (a positive even number, practically - 6, 8 or 10),

The current-phase relation in RSJN model is described by the following relation

$$I_{i,j} = I_c \cdot \sin(\phi_i - \phi_j) + \frac{V_{i,j}}{R_n} \cdot \frac{\left(\frac{V_{i,j}}{V_g}\right)^n}{1 + \left(\frac{V_{i,j}}{V_g}\right)^n} + C \cdot (\ddot{\phi}_i - \ddot{\phi}_j)$$

Here, $V_{i,j} = \dot{\phi}_i - \dot{\phi}_j$ is a voltage drop on the junction.

For convenience, there are a few “function closures” (substitutions) for RSJN model defined in PSCAN2 parser:

$JN(J1) \Rightarrow RSJN(J1 \cdot X_j, 1.0/J1, RSJN_VG, RSJN_N, J1)$, where $J1$ is a critical current parameter, $X_j = 1.0$ is a global parameter describing a critical current density deviation. The function assumes $I_c \cdot R_n = 1.0$ and $b_c = 1.0$ (i.e., a nominally shunted Josephson junction). $RSJN_VG$ and $RSJN_N$ are globally defined variables for RSJN model;

$JJN(J1, XJ1) \Rightarrow RSJN(J1 * XJ1 * Xj, 1.0/J1, RSJN_VG, RSJN_N, J1 * XJ1)$, where $J1$ is a critical current parameter, $Xj = 1.0$ is a global parameter describing a critical current density deviation, $Xj1 = 1.0$ is a hierarchical (external) parameter describing individual JJ critical current deviation (i.e. area). The function assumes $I_c * R_n = 1.0$ and $b_c = 1.0$ (i.e., a nominally shunted Josephson junction). $RSJN_VG$ and $RSGN_N$ are globally defined variables for RSJN model;

Tunnel model of Josephson junction

TJM is the most complex and adequate model of a Josephson junction. It is based on Werthamer model [4] and employs Dirichlet series approximation of its integral kernels [5].

$tjm(coeff_set_name, I_c, Wbc, Wvg, Wvcrat, Wvrrat)$

coeff_set_name – a TJM approximation set name (text string). It is normally listed in the file `psconfig.py`, but can be custom made. There are four default Dirichlet series coefficient sets for different widths of Riedel peak [6] (fabrication process dependent). To obtain graphical representation of the available TJM kernels, run `tjmplot.py` file from the auxiliary directory in PSCAN2 package (for an expert).

I_c – critical current

Wbc – dimensionless capacitance of a JJ (β_c)

Wvg – gap voltage (V_g)

$Wvcrat$ – $I_c R_n / V_g$ ratio (Ambegaokar-Baratoff relation) [7]

$Wvrrat$ – R_n / R_{sg} (normal-to-subgap resistance ratio).

For convenience, there are a few “function closures” (substitutions) for TJM model defined in PSCAN2 parser:

$jt(J1) \Rightarrow tjm(“tjm1”, J1 * Xj, Wbc, Wvg, Wvcrat, Wvrrat)$, where $J1$ is a critical current parameter, $Xj = 1.0$ is a global parameter describing a critical current density deviation. Wbc , Wvg , $Wvcrat$, and $Wvrrat$ are globally defined TJM parameters. It uses “tjm1” coefficients set from the default set in `psconfig.py`.

$jjt(J1, Xj1) \Rightarrow tjm(“tjm1”, J1 * Xj1 * Xj, Wbc, Wvg, Wvcrat, Wvrrat)$ where $J1$ is a critical current parameter, $Xj = 1.0$ is a global parameter describing a critical current density deviation, $Xj1 = 1.0$ is a hierarchical (external) parameter describing individual JJ critical current deviation (i.e. area). Wbc , Wvg , $Wvcrat$, and $Wvrrat$ are globally defined TJM parameters. It uses “tjm1” coefficients set from the default set in `psconfig.py`.

Circuit parameters definition files

While circuit netlist is in a single hierarchical netlist file, the parameters and HDL descriptions are in *.hdl files. Every cell (a subcircuit) in the simulated circuit netlist should have a corresponding “<cellname>.hdl” file either in the current work directory or in one of the library paths listed in \$PSCAN_CIRCUIT_PATH. The environment variable PSCAN_CIRCUIT_PATH includes library directories in format

```
PSCAN_CIRCUIT_PATH = “dir1;dir2;dir3; ...”
```

The file with circuit definition has the following format:

Parameter

```
<global_param1> = <val 1>,
```

...

```
<global_paramN> = <val N>;
```

Internal

```
<global_internal 1> = <exp1>,
```

...

```
<global_internal N> = <expN>;
```

```
circuit <cell_name> ()
```

```
{
```

Parameter

```
<param1> = <pval 1>,
```

...

```
<paramN> = <pval N>;
```

External

```
<external 1> = <extval 1>,
```

...

```
<external N> = <extval N>;
```

Internal

```
<internal 1> = <intexp1>,
```

...

```
<internal N> = <intexpN>;
```

Value

```
<value1> = <val exp1>,
```

...

```
<valueN> = <val expN>;
```

freeze

```
<j name1>, ..., <j nameN>;
```

```
rule <rname1>(<init_expr>
```

```
<expr1>,
```

...

```
<exprN>;
```

```
rule <rname2>(<init_expr>
```

```
<expr1>,
```

...

```
<exprN>;
```

...

```
}
```

Here, *<cell_name>* - is the subcircuit (instance) name. There are six types of record in the instance description file, - *parameter*, *external*, *internal*, *value*, *freeze*, and *rule*.

parameter - is a subcircuit variable parameter. A PARAMETER variable has floating-point value and can be used in element's models. It has the same value in all instances of the subcircuit. For example, if parameter J1 in the cell named JTL changes, it would change in all instances of JTL subcircuit. The access to the PARAMETER variable is through its cell name, e.g., critical current of junction J1 in the cell JTL has a full name "jtl.j1"

external - is a circuit hierarchical parameter. An EXTERNAL variable has floating-point value and can be used in element's models. But, unlike a PARAMETER variable, it has individual value for every instances occurrence in the circuit. For example, external parameter XJ1 in JTL circuit MTL1 can have different value from the value of XJ1 EXTERNAL parameter in circuit MTL2. For convenience, we use a non-binding convention of starting EXTERNAL parameter names with "X". The access to an EXTERNAL variable is through the cell's full path, e.g., optimization factor for critical current of junction J1 (Xj1) in the second cell JTL1 in first cell STDIN in TESTNOT example circuit (fig. 2) has a full name ".m1.m2.xj1" (note the leading dot in the name). A variable gets its full name only after PSCAN2 initialization, so, in the current version of the program, a user cannot assign different initial values to the same EXTERNAL parameter in different cell instances.

internal - is the a circuit parameter, that changes every time step during circuit simulation. An INTERNAL parameter is a floating-point expression and can be used in element's models. Different instances of the same subcircuit may have different values of the same internal parameter. An INTERNAL parameter can be a function simulation time (there is a reserved global internal parameter called 'TCURR', that is a current time in the simulation). An INTERNAL parameter cannot be a function of resulting phase, current, or voltage value across circuit elements or nodes. After initialization, an INTERNAL parameter can be accessed through its full name (same as EXTERNAL)

value - is an expression of circuit's calculable parameters (e.g., phase, voltage, current, etc.). A VALUE parameter is a floating-point expression that can depend on simulation time, phase, current, or voltage across circuit elements or nodes. It may not be used as a parameter for circuit element's models. Obviously, different instances of same circuit have different values of the same VALUE variable.

freeze - is a list of "frozen" Josephson junctions. These junctions are not to be monitored during SFQHDL simulation of the circuit. Switching of a "frozen" junction would not generate an event during circuit simulation.

rule - is definition of SFQHDL rule, describing expected behavior of the circuit at some certain condition. A "rule" has a name followed by *<init_expr>*, an expression, that activates the rule, when it is equal to logical "True" (any non-zero value). Then, follows the list of events. See SFQHDL language description below.

Global parameters or internals can be declared outside of any circuit. They are visible in all circuits and parameters declared inside circuit definitions cannot have similar names.

SFQHDL syntaxes

SFQHDL is a hardware description language developed mainly for simulating digital superconductor RSFQ circuits [8]. On logic level, (R)SFQ circuit functionality can be described as a chain of “switching” events of Josephson junctions. Under “a switching event”, we assume a 2π phase slip corresponding to the magnetic flux (equivalent to a flux quantum) penetration through the junction. That results in a short SFQ voltage pulse across the Josephson junction.

An SFQHDL script consists of a set of structures, called “rules”. As we already mentioned, each rule has a unique name, the initialization condition, and a body comprising a chain of logical expressions (events).

```
rule <rname>(<init_expr>
    <expr1>,
    [<expr2>, <expr3>, <expr4>],
    <expr5>,
    ...
    <exprN>;
```

After initializing, the rule starts consequently querying the sequence of expressions $\langle \text{expr1} \rangle \dots \langle \text{exprN} \rangle$ for “True” return value, until the “False” is occurred or the last expression returns “True” (success). Each logical expression returns logical value True or False. For the successful “rule” execution, every expression should return “True” value in the order listed in the rule body. For instance, if $\langle \text{expr2} \rangle$ in the example above returned value before $\langle \text{expr1} \rangle$ did so, the execution of rule $\langle \text{rname} \rangle$ would have failed. However, a several expressions can be grouped into a cluster within brackets (e.g., “[e_2, e_3, e_4]”). In this case the order of execution of these expressions is not being traced, and the next expression ($\langle \text{expr5} \rangle$) should not be executed before the last expression in the cluster. The rule deactivates itself on success or interrupts simulation with corresponding return code on failure. All rules are being processed in parallel at each circuit simulation time step.

PSCAN2 has the following built-in functions usable in HDL script.

inc($\langle \text{jelement} \rangle / \langle \text{pelement} \rangle$) – returns True, if number of flux quanta across the element has increased by 1 during the current time step

dec($\langle \text{jelement} \rangle / \langle \text{pelement} \rangle$) – returns True, if number of flux quanta across element has decreased by 1 during the current time step

set($\langle \text{pin} \rangle$) – sets node’s logical variable to “active” at the next time step

get($\langle \text{pin} \rangle$) – returns value of node’s logical variable

exit($\langle \text{message} \rangle, \langle \text{condition} \rangle$) – terminates execution of the rule with error message $\langle \text{message} \rangle$ when $\langle \text{condition} \rangle$ becomes non-zero (true). Used in rules to abort circuit simulation upon some condition. Example: *exit(“Wrong state”, tcurr > 300 and n(j1) > n(j2))*

print($\langle \text{v1} \rangle, \dots$) – prints string representation of argument values to the *stdout* during rules execution. Always returns True. Example: *print(“State value stst1=”, stst1, “ at “, tcurr)*

freeze($\langle \text{jname1} \rangle, \dots$) – places junctions into the list of “frozen” junctions. Switching of a junction from this list will not produce an HDL event during circuit simulation (i.e., will be ignored). Example: *freeze(j1, j2, j13)*

unfreeze(<jname1>, ...) – removes junctions from the list of “frozen” junctions. Example:
unfreeze(j1, j2)
pyeval(“<python expression>”) – executes a python-syntaxes expression and returns its value.

Let’s look at the example of HDL script for circuit TESTNOT (Fig.2)

```

circuit testnot()
{
INTERNAL
    p1=0.85+psfq(400,4,200),
    p2=0.85+psfq(400,4,100)+psfq(400,4,300);
rule m1go(inc(p1))
    set(m1.2);
rule m2go(inc(p2))
    set(m2.2);
}

```

Here, psfq(p,d,s) function generates an SFQ-like 2π phase jump (see above). TESTNOT is a “root” circuit, - it has got no Josephson junctions and, thus, its script is very simple. Two rules (*m1go* and *m2go*) detect input pulses from the corresponding phase sources (p1 and p2) and initialize execution of internal rules in sub-circuits m1 and m2 by activating their input pins (m1.2 and m2.2). Here (Fig. 2), all sub-cells are directly connected to each other (there is no primitives between them). So, e.g., pins M1.1 and MTL1.1 share the same node and the output of cell M1 activates cell MTL1 automatically.

This is not the case in sub-cell STDIN1 (Fig. 3b). Here, the input 1 of sub-cell M1 should be explicitly activated by the following rule

```

circuit stdin1()
{
rule input (get(2))
    set(m1.1);
}

```

As you can see, HDL script for subcell STDIN1 requires only one rule. The rule “*input*” is to “jump over” the inductor L1. This is a good example of using functions get() and set().

The simplest RSFQ cell is a Josephson transmission line (Fig. 3c, 3d). SFQHDL script of unidirectional JTL consists of a single rule “go”.

```

circuit jtl()
{
...
rule GO(GET(1))
    INC(J1),
    SET(2),
    INC(J2);
}

```

The rule “go” activates at signal at pin 1, then, junction J1 switches. Here, for better timing purpose, we activate pin 2 (the output) before J2 switching, so, the next cell would be ready to accept incoming pulse well before its arrival. There are many nuances in SFQHDL script writing, this is one of them. This script assumes that SFQ pulses propagate from pin 1 to pin 2. While making “bidirectional” script is possible, we do not recommend it for inexperienced users.

Let us write SFQHDL script for more complex cell, e.g., inverter (Fig. 3a).

```

circuit not1()
{
rule rd0(GET(2) and n(j3)==n(j4))
    set(3),
    [inc(j5), inc(j1)];
rule set1(GET(1) and n(j3)==n(j4))
    inc(j2),
    inc(j3);
rule rd1(GET(2) and n(j3)==n(j4)+1)
    inc(j4);
}

```

This cell has memory and can be in two states (0 and 1). Here, we use function N() for determining the state of the cell. The quantizing (flux storage) loop consists of junctions J3 and J4 and storage inductor L1 (Fig. 3a). The number of SFQ stored (either 1 or 0) in the loop is equal to N(j3) - N(j4). So, the cell is in state “0”, if n(j3) is equal to n(j4), and in state “1” otherwise.

The rule “rd0” activates when an SFQ pulse arrives to the clock pin (2) in state “0”. The inverter produces output by switching junction j5. The first expression in the rule (set(3)) signals to the outside node that the output is coming. At about the same time, junction J1 switches to prevent pulse from going to the data line. Because the order of switching of junctions J1 and J5 is irrelevant for functionality, these two events are placed in brackets (as it’s already been mentioned, the analyzer ignores the order of events placed between brackets). The other two rules (“set1” and “rd1”) are more simple and easy to interpret.

Input signals and Test vectors

An input signal in RSFQ is represented by a 2π phase jump associated with an SFQ pulse. In PSCAN2, we use phase sources at the root level of the circuit for generating input signals. The phase drop on such a generator is time dependent and usually described by an “Internal” parameter. Looking back into *testnot.hdl* file from the example, we see two different ways of defining inputs for the circuit. One is defining “Internal” parameters using time-dependent function psfq(p, d, s)

```

INTERNAL
    p1=0.85+psfq(400, 4, 200),
    p2=0.85+psfq(400, 4, 100)+psfq(400, 4, 300);

```

Here, we define two inputs, - “data” (p1) and “clock” (p2). On a span of 400 PSCAN units of time (PUT), one “Clock” pulse with width 4 PUTs comes at 100 PUT, then – “Data” pulse comes at 200 PUT, and then

– another “Clock” pulse at 300 PUT. Although this way of defining input signals is simple, it is highly inconvenient for simulating a large circuit with many inputs.

In order to cope with a large circuit or/and with a long serial input pattern, PSCAN2 has a convenient object class called TestVector. The user can define test vectors in the separate file *pscanrc.py* situated in the current directory. This file is the place for user-defined structures, variables or functions. PSCAN2 loads this file if it is present in the current working directory from which python is launched.

The Python class TestVector is defined in *pscan2/TestVector.py* file and can be loaded from *pscan2.testvector* module.

```
TestVector(<name>, <vec>, tunit = 20.0, period = 0, repeat = 0, start = 20.0, pulse_duration = 5.0)
```

Here,

<name> – is the name of test vector. If *name* is not an empty string (“”), the test vector will be placed in the global test vector table and would be available in the circuit.

<vec> – is a definition of the test vector. It is a text string of “0”s and “1”s, - a digital representation of the test vector. The total number of “bits” in the string *<vec>* ($N = \text{length}(\text{vec})$) defines the duration of the sequence, unless *period* is not 0.

tunit – is a “time unit”, i.e. the time distance between “bits” in *<vec>*.

period – is the length of the test vector *<vec>* section in *tunits*. If *<period>* is greater than the length of *<vec>* parameter ($\text{period} > N$), the vector string *<vec>* will be padded with $(\text{period} - N)$ “0”s.

repeat – is a repeat count. The test vector will be repeated *repeat* times and then stay constant. If *repeat* = 0 (default), the test vector would repeat indefinitely.

start – is the start time of the test vector. This is the time at which the whole test vector procedure should begin, i.e. the time shift of 0 count.

pulse_duration – is the duration (width) of the SFQ pulses in the test vector.

Let us open, as an example file *pscanrc.py* in directory *testnot*. Its content is only three strings defining two input test patterns:

```
from pscan2.TestVector import TestVector
TestVector("d1", "010", 100.0, 0, 1, 100.0)
TestVector("d2", "101", 100.0, 0, 1, 100.0)
```

These records define two test vectors: “d1” (invertors’ data input) and “d2” (invertor’s clock). In the root description file of the circuit (*testnot.hdl*), the following records assign the test vectors to the corresponding phase generators by means of built-in function *tecv()*. A dc phase shift of 0.85 simply compensates the phase shift of a nominally dc-biased Josephson junction.

INTERNAL

```
p1 = 0.85 + tvec("d1"),
p2 = 0.85 + tvec("d2");
```

Python interface.

PSCAN2 is implemented as a Python module “pscan2”. A set of python-based functions is provided for running PSCAN2 from the PYTHON console. The interface includes the following functions:

initialize(<args>) – initializes PSCAN2 and load circuit netlist and definition files

<args> is the array of arguments, where the last argument is the name of root circuit and base name of the circuit netlist file (the full name of the netlist file should have “.cir” extension appended). All previous arguments, if existed, are the names of circuit definition files that should be preloaded into memory. After loading netlist file, PSCAN2 will try to load circuit definition files used in the netlist. If circuit definition was preloaded, it will use that definition, otherwise it will try to load a corresponding “*.hdl” file.

find(<name>[, <type>])

Finds objects in circuit hierarchy by name.

<name> - a hierarchical name of object. For global parameters or global internals it is just a name. For example: find(“XI”). For the objects in root circuit, name should start with “.”. For example: find(“.J1”). For objects on deeper levels of hierarchy, name should contain all subcircuit instances, separated by “.”. For example: find(“m1.jt1.XJ1”)

<type> is a type of object to find:

'e' – circuit element

'n' – node

'p' - parameter or external. Default value

'v' - internal or value

The function returns instance of the found object or raises an exception, should the object has not been found.

param_find(<circuit_name>, <parameter_name>)

Finds parameter <parameter_name> in a circuit <circuit_name>. There is no need to specify full hierarchical path for parameters, - they can be found by the circuit name. Function will return object instance of the parameter, or raise exception.

find_re(<pattern>[, <type>])

Similar to find() function, but returns a list of objects, that are matching simple regular expression. It uses Python module “fnmatch” and supports the following special symbols in the pattern:

* - a “wild card” (matches any set of characters)

? - matches any single character

[seq] - matches any character from “seq”

[!seq] - matches any character but from “seq”

<type> is a type of objects to find, the same as in find() function

Function will return list of found objects. If no objects match the pattern, list will be empty.

For example: find_re(“m1.xj*”) will find all parameters in circuit m1, started with “xj”.

param_re(<circuit_name>, <pattern>)

Similar to param_find() function, but returns list of circuit parameters matches simple regular expression, with same syntax as in find_re().

For example: `param_re("not1", "j*")` returns all parameters in circuit "not1", whose name starts with character "j".

simulate(`<tseq>`, `print_messages = False`, `pretty_print = False`)

Simulate circuit and test SFQHDL rules.

`<tseq>` is the maximal simulation time. If during the simulation some junction incorrectly switches (i.e., there are no any rule waiting for its switching in `inc()` or `dec()` expression), the simulation stops with the corresponding error message.

`print_message` - if *True*, there will be trace of rule execution printed out during the simulation.

`pretty_print` - if *True*, rule traces will be printed in the pseudo-graphical form.

The function returns *True*, if there was no unexpected Josephson junction switches during the simulation and there are no any active rules left at the end the simulation time (*False* otherwise).

margins(`<parameter>`, `<tseq>`, `messages_level=0`, `max_margin=0.4`) - Calculates operating margins of the parameter.

`<parameter>` - name of the parameter or circuit parameter or external object instance, obtained with functions `find()` or `param()`.

`messages_level` - defines extent of printed messages:

0 – no messages (silent mode)

1 – basic messages related to calculating margins of the parameter

2 – all messages, including rule's execution traces during the simulation

`max_margin` – is the maximal interested value of the parameter margins. Meaning, e.g., in case of `max_margin=0.3`, the correct operation of the circuit will not be verified beyond +/- 30% range. The default value is 0.4 (i.e., +/-40%).

Function returns the list of the left and the right margin. The list of two *None* objects is returned, if the circuit does not operate correctly at the current point (initial value of the parameter).

save_parameters(`<circuit_list>`) – save circuit parameters into parameters values files.

`<circuit_list>` - is a list of subcircuit names to save parameters from. A file `<circuit>.par` will be created for every cell name from the list.

load_parameters(`<circuit_list>`) – load circuit parameters from *.par files.

`<circuit_list>` - is a list of circuit names for which parameters are to load. For each `<circuit>` from the list, if file `<circuit>.par` exists, it will be loaded.

save_all_parameters(`<fname>`) – Save all parameters in the circuit, including global parameters, to the file `<fname>`.

load_all_parameters(`<fname>`) – Loads (updates) circuit parameters from the file `<fname>` created by `save_all_parameters()` procedure.

ivcurve(`<xpar>`, `<xmin>`, `<xmax>`, `<dx>`, `<xback>`, `<yexpr>`, `<yeps>`, `<twait>`, `<tmin>`, `<tmax>`) – calculates dc current-voltage dependence curve.

`<xpar>` - parameter to change. Can be found by function `find()`. For example `find('.i1', 'p')`

`<xmin>` - minimal value of parameter

`<xmax>` - maximal value of parameter

`<dx>` - step for parameter's changes

<xback> - if `False` parameter will be scanned from <xmin> to <xmax> with step <dx>. If `True`, - parameter will be scanned from <xmin> to <xmax> and back to <xmin>.

<yexpr> - expression to average over the time for each fixed value of parameter <xpar>. It is defined by function `create_expression()`, e.g., `create_expression("v(j1)", ".")`

<yeps> - relative accuracy for averaging expression <yexpr> over time

<twait> - setup time after the changing parameter <xpar>

<tmin> - minimal averaging time

<tmax> - maximal averaging time. The averaging process stops after <tmax>, even if the averaging precision <yeps> has not been reached.

Function returns a list of calculated points ([x, y] pairs).

PSCAN2 Graphical User Interface (GUI)

The PyQt-based graphical user interface can be called as a python module:

```
python -m pscan2.gui <root_cell_name>
```

Let us look at the example “testnot” again. In the directory *testnot*, run

```
python -m pscan2.gui testnot
```

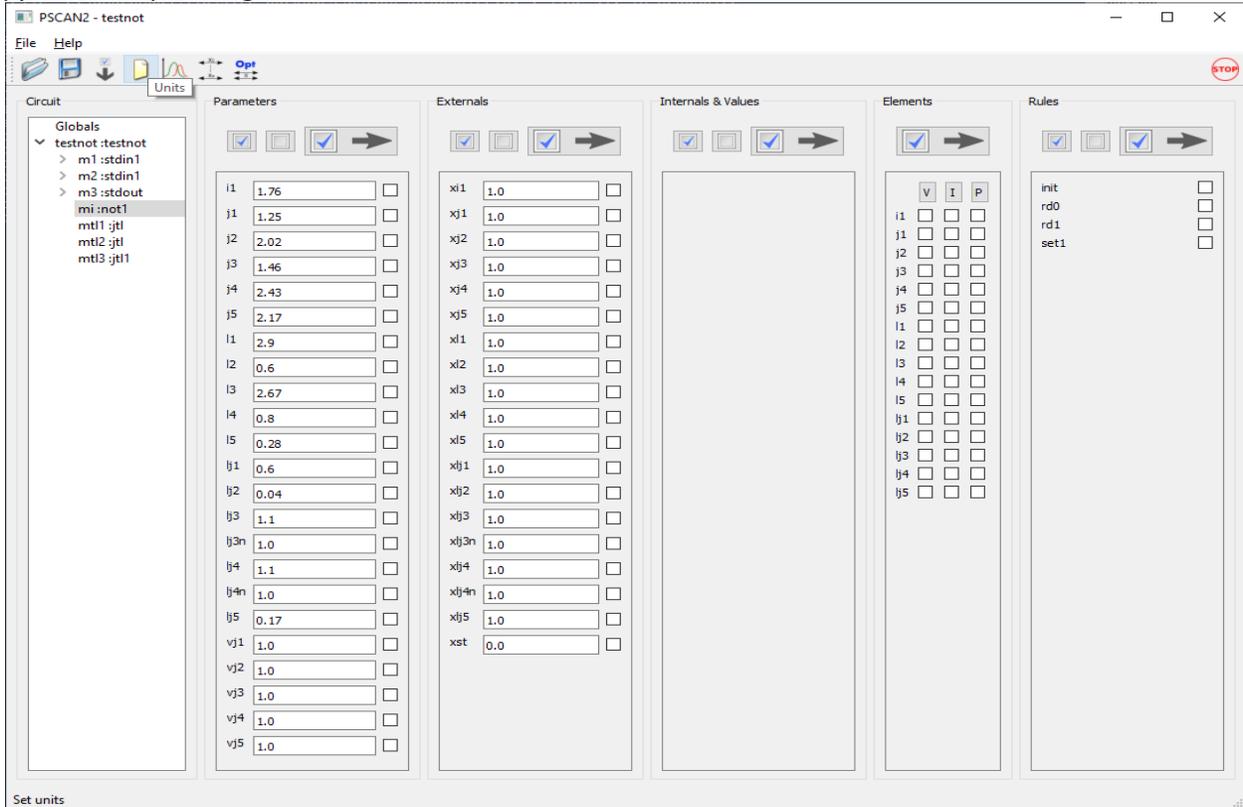


Fig. 4. A screenshot of PSCAN2 graphical user interface

The interface window has six frames: “Circuit”, “Parameters”, “Externals”, “Internals & Values”, “Elements”, and “Rules”. The whole circuit hierarchy is in the left frame, - please, expand the cell tree and brows the schematics (Fig. 2,3). In frames “Parameters” and “Externals”, the numerical fields are writable. In order to change the parameters value, the user should select the field he wants to change, type the new numerical value and press “Enter” on the keyboard.

The toolbar is on a top of the main window. It has six icons (from left to right):

“Load parameters” – loads parameters from the chosen file.

“Save all parameters” – saves all parameters to the chosen file.

“Save circuit parameters” – saves parameters in the format convenient for updating *.hdl files.

“Units” – sets PSCAN dimensionless units.

“Setup Graph” – launches the window for setting up transient analysis.

“Calculate margins” - launches the window for setting up parameters operational margins extraction.

“Optimize margins” - launches the window for setting up multi-parameter optimization procedure.

“Stop” – the icon in the right top corner, interrupts simulations and saves the intermediate results.

Transient analysis

In the toolbar, pick “Setup Graph” icon (the name of the selected icon is on the bottom of the GUI window).



Fig. 5. Setup Graph window. Initial (a) and filled (b)

There are two types of graphical windows, - “Values” and “Rules”. By browsing the schematics in “Circuit” frame, pick wanted for observing parameters from “Elements” frame and SFQHDl script tokens from “Rules” frame. Then, distribute them between corresponding plot frames in the “Graph Setup” window. You may add or remove plot frames with “Add plot” and “Delete” buttons. To select the current frame, you need to point at the frame’s border.



Fig. 6. Transient analysis.

Now, let's leave this exercise for later and just press button "Load" to load configuration that came with the example (Fig. 5b). In the future, once you gathered all parameters for the transient analysis, press "Save" button to store the created configuration; - this should save you a lot of time. After finishing configuring the graphics, press "Done" button for activating the simulation.

In the obtained graphic window, you may change scaling (zooming in or out) of each trace individually or for all traces together. By pressing and dragging the left button of the mouse, you can pan the trace, while the right button provides zoom. By right clicking on the trace, you will get into the synchronization menu.

Operational margins analysis

Close the graphical simulation window and click on the "Calculate Margins" icon in the main toolbar (Fig. 7). Then, select "Globals" in frame "Circuit" and "XI", "XJ", and "XL" in frame "Parameters". Press button "Move" (with an arrow) to move selected parameters to the "Margins" window. Select cell "mi :not1" in "Circuit" frame and all parameters XJ* in the "Externals" frame. Move them too. Press "Done" for running parameters operational margins calculation. The results will be printed to the console (STDOUT) (Fig. 7b). By default, all margins start at 40%. It is set by optional parameter *max_margin* in function *margin()*.

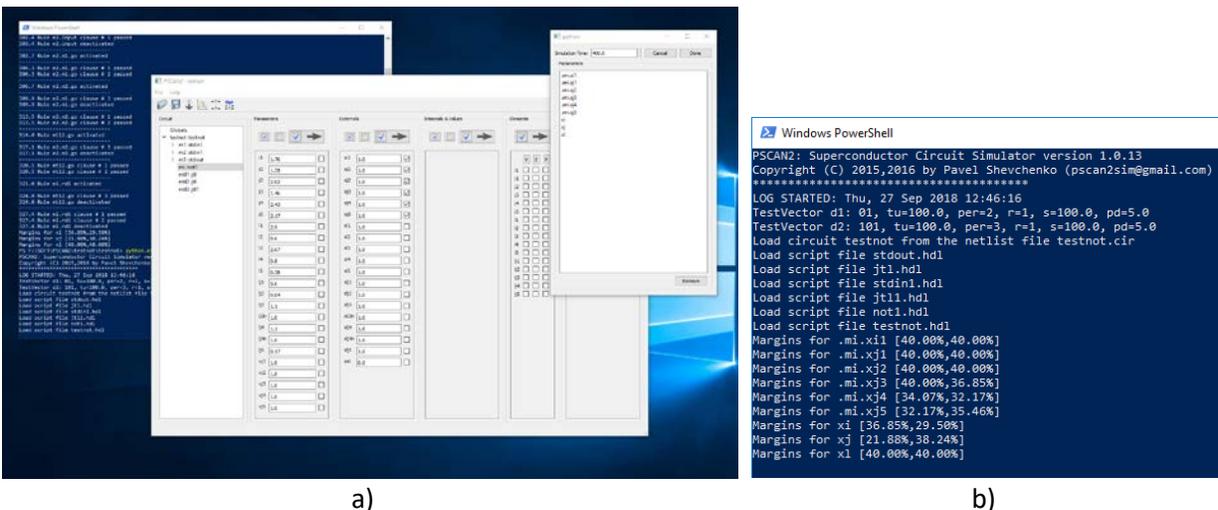


Fig. 7. Margins calculation procedure. Setup (a) and the result (b)

Parameters optimization

The multi-parameter circuit optimization algorithm is based on COWB optimizer [2] in old PSCAN. To call optimization setup window, click on "Optimize Margins" icon in the toolbar. In order to create the optimization procedure, the user has to create two lists of parameters, - the list of parameters to optimize with desired margins and the list of parameters allowed to change during the optimization with their variation limits. Usually, parameters to optimize are "Externals" or "Globals", while parameters allowed to be modified are "Parameters".

In order to create the list of being optimized parameters, select them the same way as described in previous section (Fig. 8a) and then, highlight and move them to “Parameters to Optimize” frame by pressing button “To Optimize”. The highlighted parameters will be moved to the “Parameters to Optimize” frame (Fig. 8b). To each parameter there will be assigned a default “required margins” value that can be customized.

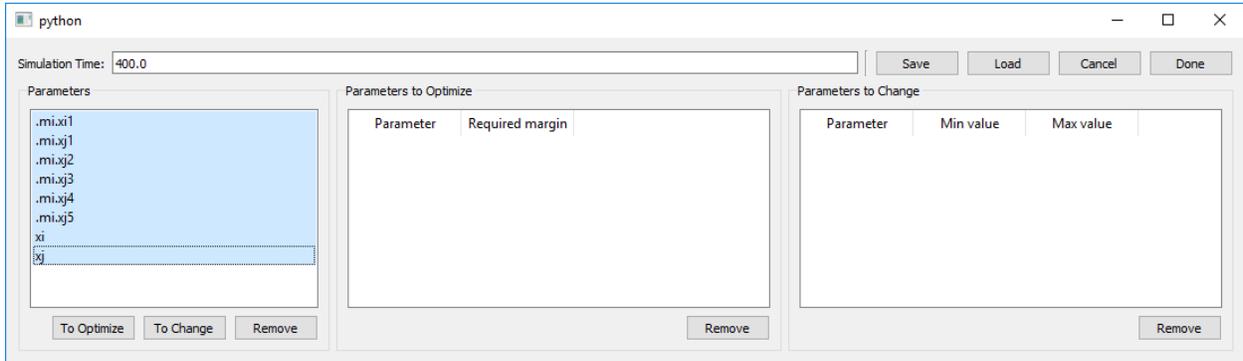


Fig. 8a. Parameters optimizer. Creating the list of parameters to optimize.

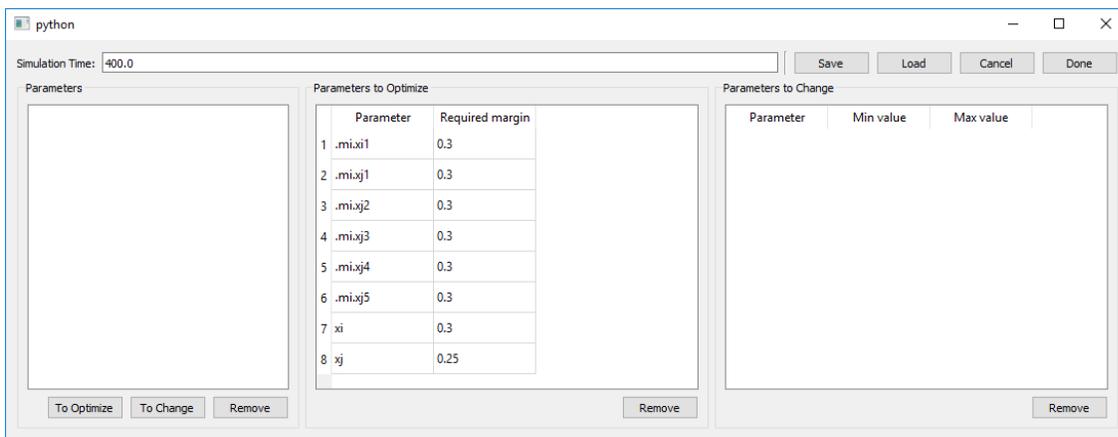


Fig. 8b. Parameters optimizer. List of parameters to optimize with desired margins value.

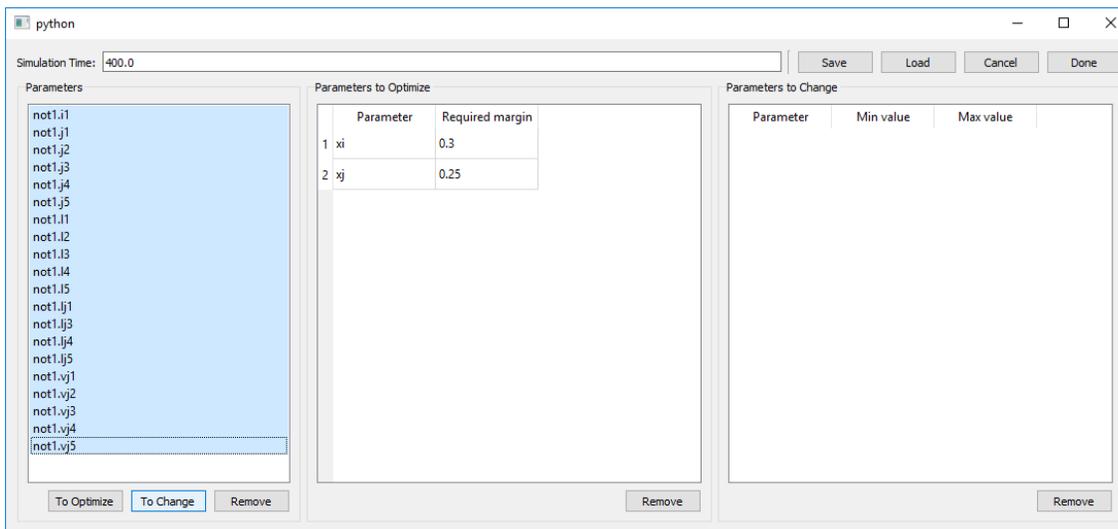


Fig. 8c. Parameters optimizer. Creating the list of parameters allowed to change.

Similarly, the list of parameters allowed to change (Fig. 8c) can be created. Each parameter in this list has a couple of default numbers assigned to it, - minimal and maximal values of the parameter. These numbers limit parameter's variation. They can be changed the same way as described above.

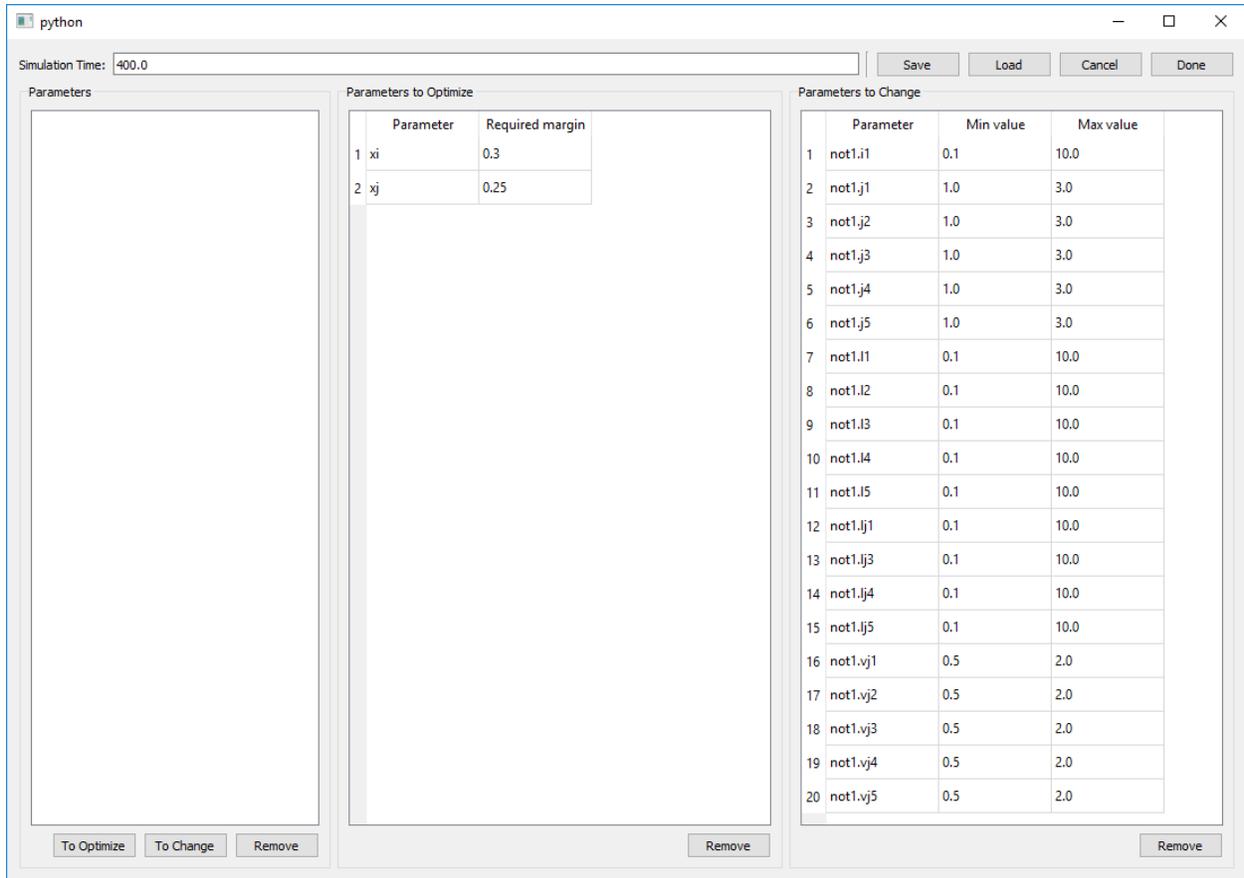


Fig. 8d. Parameters optimizer. Creating the list of parameters allowed to change with their limits.

“Save” button on a top of the window saves the created tables to the file “opt_menu.dat in the current directory. Similarly, “Load” button restores these tables into the open optimization setup window.

For the demonstration purpose, we have created the demo layout that can be loaded by pressing “Load” button, - please, do so. After that, you can launch the optimizer by pressing “Done” button. The optimization process might be quite time consuming, depending on circuit complexity and the desired margins values. In the present example, this should take 5-10 min. You may interrupt the optimization by pressing “Stop” button in the top-right corner of the main window.

Save/Load Parameters

The four left icons in the toolbar (fig. 4) belong to the procedures related to saving or restoring current parameters of the circuit.

“Load parameters” button calls procedure that loads all circuit parameters from the file created by the “Save all parameters” procedure.

“Save all parameters” procedure saves all parameters in text format to the file defined by the user. The format of the file is easily readable and support dimensional units as well. Each line with the parameter and its dimensionless value, has a comment with its physical value, should it had any (see Appendix A).

“Save circuit parameters” procedure saves parameters in the format convenient for updating *.hdl files. This is useful for updating cell descriptions. Each cell will have its parameters saved into a file named <cell_name>.par. This files then can be used for updating *.hdl files of the circuit. After that, the new parameter values become permanent and default.

“Units” procedure sets PSCAN dimensionless units. PSCAN, as it has been described in Appendix A, operates with dimensionless parameters. In order to convert them into the real physical values at “Save all parameters” procedure execution, a user should define the units first. To do so, the user should pick arbitrarily units for two physical values: current and voltage. This should be done at each PSCAN session started.

The default units are set in *psglobals.py* file in the installation directory and can be redefined in *pscanrc.py* file in the current work directory.

References

- 1) S. Polonsky, V. Semenov, and P. Shevchenko, PSCAN: Personal superconductor circuit analyzer, *Supercond. Sci. Technol.*, vol. 4, no. 11, pp. 667-70, 1991.
- 2) S. Polonsky, P. Shevchenko, A. Kirichenko, D. Zinoviev, A. Rylyakov, PSCAN'96: new software for simulation and optimization of complex RSFQ circuits, *IEEE trans. on appl. Supercond.* 7 (2), 2685-2689
- 3) W.C. Stewart, Current-voltage characteristics of superconducting tunnel junctions, *Journal of Applied Physics*, 45, 452, 1974
- 4) N. R. Werthamer, Nonlinear Self-Coupling of Josephson Radiation in Superconducting Tunnel Junctions, *Phys. Rev.* 147(1), 255, 1966
- 5) A. A. Odintsov, V. K. Semenov, and A. B. Zorin, SPECIFIC PROBLEMS OF NUMERICAL ANALYSIS OF THE JOSEPHSON JUNCTION CIRCUITS, *IEEE TRANSACTIONS ON MAGNETICS*, VOL. MAG-2, NO. 2, MARCH 1987
- 6) Riedel E., Zum Tunneleffekt bei Supraleitern im Mikrowellenfeld, *Zeitschrift Naturforschung Teil A*, Vol. 19, p.1634
- 7) V. Ambegaokar, A. Baratoff, Tunneling Between Superconductors, *Phys. Rev. Lett.* 10, 486-489 (1963).
- 8) K.K. Likharev and V.K. Semenov, RSFQ Logic/memory Family: A New Josephson-Junction Technology for Sub-Terahem-Clock Frequency Digital Systems, *IEEE Trans. on Appl. Supercond.*, vol. 1, pp. 3-28, 1991

Appendix A

PSCAN units

Like its predecessor, PSCAN2 operates with dimensionless units. For convenience, PSCAN uses normalized units system, where the magnetic flux unit is equal to $\Phi_0/2\pi = \hbar/2e \approx 0.33 \cdot 10^{-15}$ Wb

At these units, the magnetic flux in a loop is equal to the superconductor phase drop in it.

In order to normalize all other units, a user should define two of them: the current unit and the voltage unit. The choice of these units depends on the fabrication process. For instance, it is convenient using close-to-minimal Josephson junction critical current for the current unit and the $I_c R_n$ product of a nominally shunted ($\beta_c=1$) junction, i.e., plasma voltage, as a voltage unit.

PSCAN determines the physical value of the parameter by its first letter (see table below). This dependence is set by *psglobals.Par2Units* variable and can be redefined.

First letter in the parameters name	Units of	Notation	Relation
P	Magnetic flux (supercond. phase)	Φ_u	$\Phi_0/2\pi$
I, J	Current	I_u	Arbitrary
V	Voltage	V_u	Arbitrary
T	Time	t_u	Φ_u / V_u
R	Resistance	R_u	V_u / I_u
L	Inductance	L_u	Φ_u / I_u
C	Capacitance	C_u	$\Phi_u \cdot I_u / V_u^2$

The typical units for known fabrication processes are in table below

Notation	MIT-LL SFQ5ee	HYPRES 4.5 kA/cm ²	HYPRES 10 kA/cm ²
I_u	0.05 mA	0.125 mA	0.125 mA
V_u	0.67 mV	0.47 mV	0.63 mV
t_u	0.49 ps	0.7 ps	0.52 ps
R_u	13.4 Ohm	3.48 Ohm	5.04 Ohm
L_u	6.6 pH	2.64 pH	2.64 pH
C_u	36.6 fF	201 fF	103 fF

Appendix B

Default netlist parameters

For convenience, PSCAN2 netlist parser has a built-in function for automatically assigning default parameters to the netlist elements. It happens, when in the “value” field of the netlist element there is a “?” symbol. The parser generates default values based on the elements’ name. For instance, a netlist string

L2 n3 n4 L?

will be translated to

*L2 n3 n4 L2*XL2*XL*

A special case is a Josephson junction element (that has several models) and a mutual inductance element (that does not have a default value).

The default netlist element value fields are in the table below.

Element	Name of the element (example)	Value field of the element	Default substitution
Inductor	L13	<i>L?</i>	<i>L13*XL13*XL</i>
Resistor	R8	<i>R?</i>	<i>R8*XR8*XR</i>
Current source	IB1	<i>I?</i>	<i>IB1*XIB1*XI</i>
Josephson junction	J2	<i>J</i>	<i>RSJ(J2*XJ, 1.0/J2, J2)</i>
	J2	<i>JJ</i>	<i>RSJ(J2*XJ2*XJ, 1.0/J2, J2)</i>
	J2	<i>JJJ</i>	<i>RSJ(J2*XJ2*XJ, VJ2/J2, J2*VJ2)</i>
	J2	<i>JT</i>	<i>TJM(“tjm1”, J2*XJ, Wbc, Wvg, Wvcrat, Wvrrat)</i>
		<i>JJT</i>	<i>TJM(“tjm1”, J2*XJ2*XJ, Wbc, Wvg, Wvcrat, Wvrrat)</i>

Here, e.g., *XL13* is an EXTERNAL parameter representing an individual inductance deviation, and *XL* is a GLOBAL parameter representing a global inductance deviation. In case of a Josephson junction, *J2* is its critical current, *XJ2* is a hierarchical (EXTERNAL) parameter describing individual critical current deviation (i.e. area), *VJ2* is an $I_c R_n$ product of the junction, and *XJ* is a GLOBAL parameter describing critical current density deviation.

Appendix C

Using PSCAN Python-defined parameters.

PSCAN2 has an elaborate set of configuration parameters. They are mainly defined in three PSCAN system files: *psglobals.py*, *psconfig.py*, and *SimulationParameters.py*. An experienced user can manipulate these parameters to adjust PSCAN to his needs.

When loading, PSCAN2 looks for the file ***pscanrc.py*** in the current directory and executes all Python command from it before initializing. This is very convenient for changing the default values of PSCAN global parameters.

PSCAN2 parser has a built-in function for evaluating a Python expression.

pyeval(<string expression>) function evaluates a Python expression in the text string parameter and returns the result. This is very useful for accessing PSCAN global variables and settings.

A very important global variable is a logical variable “OptimizeSimulation” from file “psglobals.py”. If it is “True” (default), PSCAN will query only phase sources and only in the root circuit (RSFQ mode). So, if the user has, say, variable current sources in his circuit and HDL rules invoke them, he must set this variable to “False”. This can be done in ***pscanrc.py*** file in the current directory with the record

psglobals.OptimizeSimulation = False

And to query the status of this variable, the user can employ *pyeval*() function. Here is an example how it can be done along with some parameters initialization:

```
rule init(tcurr < 5)
exit("Set OptimizeSimulation off!", pyeval("psglobals.OptimizeSimulation"))
xst1=0, xst2=0,
tcurr > 5;
```

In this example, rule “INIT” starts executing in the beginning of the simulation, querying variable *psglobals.OptimizeSimulation* and exiting the simulation if it’s *True*. The rule initializes variables *xst1* and *xst2*, otherwise.

For instance, variable *INITIAL_RAMP* (defined in *SimulationParameters.py* file) indicates the time needed for PSCAN to set up initial values of all currents in the circuit. Its default values is 10, but a user, for some reason, may change it in *pscanrc.py* file. So, the user should set the input pulses delay to exceed this time. This may look like below.

```
INTERNAL
TRamp = pyeval("SimulationParameters.INITIAL_RAMP"),
CircuitReady = TCURR > Tramp,
p1 = psfq(400, 4, TRamp),
i2 = ai2 * sin(w1 * tcurr);
```

Here, we define a logical variable `CircuitReady` that indicates that circuit is ready for simulation. And a phase source `P1` will launch the first SFQ pulse when all bias currents are set.

The last line defines a variable current source with a sine wave generator by employing a built-in function from the Python standard module “`math`”. This current source will obviously start setting the current from 0.0 time, regardless the `INITIAL_RAMP` variable.

Another couple of important variables are

`SimulationParameters.DTMax = 10.0`

and

`SimulationParameters.DTMin = 0.01`

They defined the range of variation of the time step during the simulation. An experienced user can change these parameters in order to improve speed or quality of the simulation.