Work in progress

1 Introduction

PyJJASim is a circuit simulator including Josephson Junctions as components, intended to be used on large Josephson Junction Arrays (JJAs).

PyJJASim is specialized in keeping track of Josephson vortices in the circuit. It can also compute static configurations that have vortices at desired locations in the circuit.

This requires that the circuit is a planar embedding (in 2D), such that one can unambiguously refer to faces of the circuit, and vortices reside at faces. This imposes that nodes in the circuit must be placed at 2D coordinates, and that no junctions can cross. This also means no hierarchical structure is supported.

2 Features

1. supports basic components (inductors, resistors, capacitors and current- and voltage sources)
2. keep track of (and place) Josephson vortices in the circuit
3. compute static configurations
4. determine dynamic stability of static configurations
5. maximize parameters that have stable static configurations
6. compute time evolutions
7. define external magnetic flux through each face
8. thermal fluctuations modeling nonzero temperature
9. visualization and animation of simulation results

3 Citing

Anyone is free to use pyJJASim, please cite it if used in a publication.
@MISC{Lank2021,
    author = "Lankhorst, M.",
    title = "PyJJASim: Circuit Simulator including Josephson Junctions in Python",
    year = "2022",
    url = "https://github.com/martijnLankhorst/pyJJAsim",
    note = "Release 2.1"
}

4 Example uses

4.1 normalized units

In PyJJASim, all quantities are dimensionless. The scalar quantities a_0, I_0, R_0 with units are used to normalize distances, currents, resistances respectively, and can be freely chosen. To represent normalized currents and resistances the symbols i and r are used respectively. Other quantities are normalized by derived quantities. These include:

Quantity	Normalizing scalar	Equals	normalized symbol
Voltages	V_0	$I_0 R_0$	$v = V/V_0$
Magnetic fluxes	Φ_0	$h/(2e)$	$f = \Phi/\Phi_0$ (frustration)
Times	t_0	$\Phi_0/(2\pi I_0 R_0)$	$\tau = t/t_0$
Energies	E_{J0}	$I_0\Phi_0/(2\pi)$	$\epsilon = E/E_{J0}$
Temperatures	T_0	E_{J0}/k_B	$\bar{T} = T/T_0$
Capacitances	C_0	$\Phi_0/(2\pi I_0 R_0^2)$	$c = C/V_0$
Inductances	L_0	$\Phi_0/(2\pi I_0)$	$l = L/L_0$
Areas	A_0	a^2	$a = A/A_0$

Where h, e and k_B are Planck's constant, the electron charge and the Boltzmann's constant respectively. Normalized external flux is also called frustration.

4.2 definition of circuit

A circuit is represented by an embedded graph G containing a set of nodes $N(G) = \{n_1, n_1, ..., n_{N_n}\}$ and a set of edges $E(G) = \{e_1, e_2, ..., e_{N_j}\}$, where each node n_i is located at coordinate (x_i, y_i). This graph must be planar so that no edges overlap. Such a graph has a set of faces $F(G) = \{f_1, f_2, ..., f_{N_f}\}$.

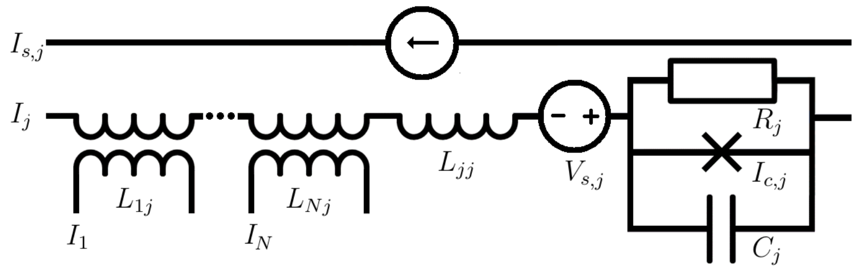

Figure 1: Basic circuit element at each edge of the network.

The nodes are electrically connected by edges with a basic circuit element containing a Josephson element, resistor, capacitor, inductors, current source and voltage source, see figure 1. Such an element is also called a junction. Each of these components has a value associated with it at each junction, called critical current I_c, normal-state-resistance R_n, capacitance C, self-inductance L_s, source current I_s and source voltage V_s respectively. Each junction can also have mutual inductance with every other junction, the self- and mutual inductances are collected in the inductance matrix $\mathbf{L}$.

```
class Circuit(graph: EmbeddedGraph, critical_current_factors=1.0,
              resistance_factors=1.0, capacitance_factors=0.0,
              inductance_factors=0.0, matrix_format="csc")
```

```
class EmbeddedGraph(x, y, node1, node2,
                    require_single_component=False,
                    require_planar_embedding=False)
```

4.3 definition of problem types

Two types of problems are of interest; static configurations and time evolutions.

In static problems one aims to find a current configuration that is constant in time, solves the circuit, is equipotential at each node and satisfies in each face the specified number of Josephson vortices. For a static problem, one can further specify the external magnetic flux passing through each face. A solution is represented with the gauge-invariant phase difference θ, which is specified for each junction.

```
class StaticProblem(circuit:  Circuit, current_sources=0.0,
                    frustration=0.0, vortex_configuration=0,
                    current_phase_relation=DefaultCPR())
```

Time evolutions on a circuit require one (or two) initial condition(s) for θ, a timestep and the number of timesteps. Furthermore one can specify external physical parameters, such as temperature at each junction and external magnetic flux through each face; and these can be time-dependent. The current and voltage sources of the circuits can also be time-dependent (resistance, capacitance, inductance and critical current values cannot).

```
class TimeEvolution(circuit:  Circuit, time_step=0.05,
                    time_step_count=1000,
                    current_phase_relation=DefaultCPR(),
                    frustration=0.0, current_sources=0.0,
                    voltage_sources=0.0, temperature=0.0,
                    store_time_steps=None, store_theta=True,
                    store_voltage=True, store_current=True,
                    config_at_minus_1=None,config_at_minus_2=None)
```

Note that values for the current sources and the voltage sources are defined at a problem, not at a circuit.

quantity	type (size)	description
θ	vector (N_j)	gauge invariant phase difference at each junction
I	vector (N_j)	current through each junction
V	vector (N_j)	voltage drop over each junction
t	scalar	time
e	scalar	electron charge
$\hbar$	scalar	Planck's constant
$I_s(t)$	vector (N_j)	source current in each junction
$V_s(t)$	vector (N_j)	source voltage in each junction
η	vector (N_j)	Johnson noise in each junction
$\mathbf{I_c}$	diagonal matrix (N_j, N_j)	Critical current of each junction on diagonal
$\mathbf{R}$	diagonal matrix (N_j, N_j)	Resistance in each junction on diagonal
$\mathbf{C}$	diagonal matrix (N_j, N_j)	Resistance in each junction on diagonal
$\mathbf{L}$	matrix (N_j, N_j)	Inductive coupling between each pair of junctions

4.4 Other formulas

5 Static problems

In static problems one aims to find a configuration in the circuit that is constant in time. As a result, all nodes are equipotential. Such states can only exist if all edges contain a Josephson junction. A configuration can be fully described with the vector θ, which must satisfy the following system:

The current associated with a configuration is equal $\mathbf{i_c} \sin\theta$. This is a closed non-linear system that can in general have many solutions, but most

equation	non-normalized	normalized
Josephson energy	$E_J = \sum (2\pi)^{-1}\Phi_0 \mathbf{I_c}(1-\cos(\theta))$	$\epsilon_J = \sum \mathbf{i_c}(1-\cos(\theta))$
Magnetic energy	$E_M = 0.5I^T\mathbf{L}I$	$\epsilon_M = 0.5i^T\mathbf{l}i$
Capacitive energy	$E_C = 0.5V^T\mathbf{C}V$	$\epsilon_C = 0.5v^T\mathbf{c}v$
inductance parameter	$\boldsymbol{\beta}_\mathbf{L} = 2\pi\mathbf{L}\mathbf{I_c}/\Phi_0$	$\boldsymbol{\beta}_\mathbf{L} = \mathbf{l}\mathbf{i_c}$
capacitance parameter	$\boldsymbol{\beta}_\mathbf{C} = 2\pi\mathbf{I_c}\mathbf{R}^2\mathbf{C}/\Phi_0$	$\boldsymbol{\beta}_\mathbf{C} = \mathbf{i_c}\mathbf{r}^2\mathbf{c}$
Johnson noise	$\langle\eta(t),\eta(t+t')\rangle = 2k_B\mathbf{R}^{-1}T\delta(t')$	$\langle\bar{\eta}(\tau),\bar{\eta}(\tau+\tau')\rangle = 2\mathbf{r}^{-1}\bar{T}\delta(\tau')$

quantity	type (size)	description
E_J	vector (N_j)	Josephson energy
E_M	vector (N_j)	Magnetic energy
E_C	vector (N_j)	Capacitive energy
$\boldsymbol{\beta}_\mathbf{L}$	matrix $(N_j,\ N_j)$	inductance parameter
$\boldsymbol{\beta}_\mathbf{C}$	diagonal matrix $(N_j,\ N_j)$	capacitance parameter
k_B	scalar	Boltzmann constant
T	vector (N_j)	Temperature
δ	scalar	Dirac delta function

are dynamically unstable and irrelevant. The stable physically relevant solutions turn out to have a structure. One can define an integer quantity for every face, called the vorticity n and find solutions that have the additional property:

```
def static_problem.compute(initial_guess, ...) →
           static_config, status, newton_iter_info
```

In general, for every vortex configuration there is at most one stable solution associated with it.

quantity	type (size)	description
n	vector $(N_f,)$	vortex configuration (or vorticity)
pv	function vector → vector	select principle value of angle between $-\pi$ and π

In pyjjasim, a StaticProblem requires as input a circuit, I_s, f_{ext} and n. Then functions are provided to compute fast approximations to the solution, compute exact solutions using newtons iteration and determine if exact solutions exist, determine if solutions are dynamally stable and find maximal parameters for which a stable solution exists.

5.1 approximations

pyjjasim provides two algorithms to generate approximate solution; the arctan approximation and the London approximation.

In pyjjasim the funcion approximate() puts the coordinates at face centers whereas approximate_placed_vortices() allows one to manually specify the coordinates.

5.2 exact solutions

The exact solution is obtained using Newton iteration.

Iteration stops if:

Stops at iter if	result is	optional
error(iter) $<$ tol	converged	no
error(iter) $>$ 0.5	diverged	no
iter $>$ maxiter	indeterminate	no
get_n(θ(iter)) $\neq n$	diverged	yes
error(iter) $>$ error(iter $-$ 3)	diverged	yes

5.3 stability

Determines if a static configuration is dynamically stable. Resistances and capacitances at the junctions do not matter for stability.

5.4 parameter optimization

The goal of parameter optimization is to find the maximum value of the scalar λ

```
def maximal_parameter(..., $i_s(\lambda)$, $f_{\text{ext}}(\lambda)$, init,
                              upper_estimate, accept_tol):
```

has a solution (stability can optionally be required). This requires one to give an estimate for the upperbound of λ `upper_estimate`. The method returns a lower-bound (λ_-) and an upper-bound (λ_+) for λ, the longer the algorithm runs the narrower the bounds. It continues until $(\lambda_+ - \lambda_-)/\lambda_+$ falls below the requested tolerance `accept_tol`. It also returns the solution at λ_-.

Furthermore there are three built-in methods that use `maximal_parameter()`:

optimization method	i_s	f_{ext}
`compute_frustration_bounds`	i_s	$\pm\lambda f_{\text{ext}}$
`compute_maximal_current`	λi_s	f_{ext}
`compute_stable_region`	$\lambda \sin(\alpha) i_s / I_{\max}$	$\lambda \cos(\alpha) f_{\text{ext}} / f_{\text{ext},\max}$

6 Time evolution

```
def time_evo_problem.compute() → time_evolution_result
```

```
class TimeEvolutionResult(...)
```

```
def time_evolution_result.get_theta(select_time_points=None)
def time_evolution_result.get_n(select_time_points=None)
def time_evolution_result.get_phi(select_time_points=None)
def time_evolution_result.get_I(select_time_points=None)
```