pyJJAsim: A circuit simulator with Josephson junctions

M. Lankhorst

April 11, 2022

Currently in preparation for submission.

1 Abstract

pyJJAsim is an electrical circuit simulator written in python which can include Josephson junctions. Unlike existing solutions, like JSPICE3, PSCAN2 and JoSIM, pyJJAsim supports explicit placement of Josephson vortices in the faces of a circuit. The cost for this is that a circuit is required to be a 2D planar embedding so that the meaning of faces are unambiguous. pyJJAsim can find time-independent zero-voltage configurations (called static configurations), where one can include external magnetic fields, current biasses and optimize for a problem parameter. Furthermore pyJJAsim can do transient analysis where it offers state of the art performance compared to competitors by operating in the cycle space.

2 Introduction

Electrical circuit simulation software like SPICE is widely used to analyse integrated circuits. The use of superconductors in ones circuits adds a new component called a Josephson junction, which electrically acts like a non-linear inductor. Its special properties allows for many applications, such as SQUIDs, RSFQ [1], mixers, etc. In particular RSFQ drives the need to simulate large-scale circuits and multiple software packages were developed, such as JSPICE3 [2], WRSPICE, PSCAN2 [3, 4], JSIM [5, 6] and more recently JoSIM [7] all adhering to the SPICE standard. An overview of how they compare is given in [8].

However, all lack the ability to control and visualize Josephson vortices. This is addressed in pyJJAsim, and is particularly useful in 2D arrays of Josephson junctions which form a grid of potential vortex sites, called Josephson Junction Arrays (JJAs). Possible applications are determining the pinning strength of vortices and manipulating their positions using current pulses. The latter may

become useful in the context of topological quantum computing using Majorana bound states [9, 10].

Josephon vortices are defined as the winding number of the order parameter along the boundary of each face of the circuit, and so it is an integer property associated with each face. To unambiguously speak of faces of the circuit, it must be a planar embedding, such that no two wires in the circuit cross.

As circuit faces take centre stage in pyJJAsim, the conventional method of defining circuits using netlists based on recursive synthesis of subcircuits is abandoned. Rather, a circuit is defined by an embedded graph containing nodes at specified 2D coordinates and edges, where each edge has the same template circuit (or symbol) containing sources, inductors and an RCSJ element, see figure 1. If components are unwanted, one simply assigns zero to its respective value.

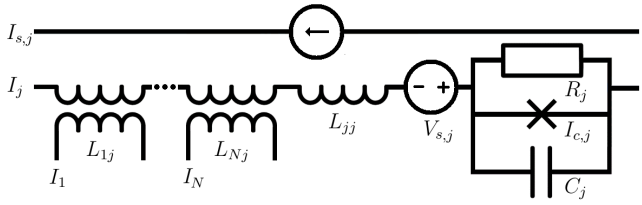

Figure 1: Basic element for each edge.

Another advantage of embedded graphs with explicit faces is that computing the cycle space is a far more tractable problem. In pyJJAsim the cycle spaces is computed explicitly and all algorithms are based on it, where other implementations avoid computing the cycle space. This results in very fast transient analysis, which on large circuits is an order of magnitude faster than JoSIM, which seems to be the fastest code to date (*Faster than JoSIM, which is faster than JSIM and WRSPICE, only need to compare to PSCAN2*).

In pyJJAsim all quantities are dimensionless. Currents, resistances and distances are normalized with free to choose scalars I_0, R_0 and a_0 respectively. Normalizations for other quantities are shown in table 2, where $\Phi_0 = h/(2e)$ is the flux quantum and e and h are the electron charge and Planck's constant respectively.

Quantity	Symbol	Normalized by
Voltage	V	$I_0 R_0$
Time	t	$\Phi_0/(2\pi I_0 R_0)$
Energy	E	$\Phi_0 I_0/(2\pi)$
Inductance	L	$\Phi_0/(2\pi I_0)$
Magnetic flux	Φ	Φ_0
Capacitance	C	$\Phi_0/(2\pi I_0 R_0^2)$

Matrices are boldface, where $\mathbf{R}$, $\mathbf{C}$ and $\mathbf{I_c}$ are diagonal matrices. All other quantities are vectors of length equal to either the node count, junction count

or face count.

2.1 Circuit theory

The current at and voltage over each primitive element equal:

$$I(t) = \mathbf{R}^{-1}\dot{\theta}(t) + \eta(t) + \mathbf{I_c}\sin\theta(t) + \mathbf{C}\ddot{\theta}(t) \tag{1}$$

and

$$V(t) = \mathbf{L}\dot{I}(t) + V_s(t) + \dot{\theta}(t). \tag{2}$$

Here the current through a Josephson element is given by the Josephson current rule and equals $\mathbf{I_c}\sin\theta$ where $\mathbf{I_c}$ is the critical current of the element and θ is the gauge-invariant phase difference across the element. The voltage over a Josephson element according to the Josephson voltage rule equals $(\hbar/2e)\dot{\theta}$ (in SI units). Furthermore $\eta(t)$ is the noise current generated in the resistor, see appendix D.

The circuit is described by Kirchhoff's laws:

$$\mathbf{M}\left[I(t) - I_s(t)\right] = 0 \tag{3a}$$

$$2\pi\dot{\Phi}_{\text{ext}}(t) + \mathbf{A}V(t) = 0 \tag{3b}$$

The current law (eq. 3a shorthand KCL) means the net current injected at each node is zero, the voltage law (eq. 3b, shorthand KVL) says the sum of the voltage drops along a closed path plus the time derivative of the external flux throught that path must be zero. Here the vector space spanned by $\mathbf{M}$ is called the cut-space and is the orthogonal complement of the space spanned by $\mathbf{A}$, which is called the cycle space.

If a circuit contains Josephson junctions, this is not sufficient to describe the circuit. If one takes the time integral of KVL (called IKVL), one introduces an integration constant which can be chosen freely if no JJs are present, but must be equal to 2π times an integer z if JJs are present, resulting in IKVL(z) [11, 12, 13, 14]:

$$\mathbf{A}\left[\theta(t) + \mathbf{L}I(t) + \int_0^t V_s(t')dt'\right] + 2\pi\Phi_{\text{ext}}(t) = 2\pi z \tag{4}$$

If a solution for $\theta(t)$ is found in system (KCL + IKVL(z)) with phase zone z, a solution $\theta(t)'$ in phase zone z' will only differ by multiples of 2π, and so with no loss of generality one can set $z = 0$. However, for finding approximate solutions, a nonzero phase zone can be convenient.

2.2 Josephson vortices

A closed loop in the circuit containing only Josephson junctions contains a Josephson vortex if the winding number of the phase of the superconducting order parameter along the loop equals 1. In general, the winding number is called the vorticity n and it "counts" the number of Josephson vortices in a loop, and if it is negative it counts the number of anti-vortices. As the circuit is a planar embedding, these loops are the boundaries of the faces, so each face surrounded by JJs has vorticity associated with it at any time.

The definition for Josephson vortices in the context of circuits is implicitly given by [14]:

$$\mathbf{A}\left[\mathrm{pv}(\theta(t)) + \mathbf{L}I(t) + \int_0^t V_s(t')dt'\right] + 2\pi\Phi_{\mathrm{ext}}(t) = 2\pi n(t). \tag{5}$$

Here $\mathrm{pv}(\theta) \in [-\pi, \pi)$ is the principle value of θ and is required because otherwise n could attain any value by adding multiples of 2π to θ and vorticity would be meaningless (vorticity defined by eq 5 is gauge independent).

3 Algorithms

3.1 Static Problem Algorithms

In static problems one aims to find a configuration in the circuit that is constant in time. As a result, all nodes are equipotential. Nonzero current can only exist in edges that contain a Josephson junction. A configuration can be fully described with the vector θ, which must satisfy the following system (in phase zone $z = 0$):

$$\mathbf{M}\left(\mathbf{I_c}\sin\theta - I_s\right) = 0 \tag{6a}$$

$$\mathbf{A}\left(\mathbf{L}\mathbf{I_c}\sin\theta + \theta\right) + 2\pi\Phi_{\mathrm{ext}} = 0 \tag{6b}$$

One can obtain solutions θ by using Newton iteration starting at a state θ_0. Then the supercurrent associated with it equals $\mathbf{I_c}\sin\theta$.

The system can in general have many solutions and the initial condition determines to which the iteration converges. A solution has a vortex configuration n which can be computed as follows:

$$n = \mathbf{A}\left(z - \mathrm{round}\left(\frac{\theta}{2\pi}\right)\right) \tag{7}$$

This follows from equations 4 and 5. It is possible to choose the initial condition such that it converges to a desired vortex configuration. This is explained in section 3.2. However, for a given vortex configuration, a solution may not exist. For example if the external current exceeds the vortex-de-pinning current. Even if a solution is found with the desired vortex configuration, it may

not be dynamically stable, meaning that if it is used as initial condition for a time evolution it will move away from this solution. In section 3.3 a criterion for dynamic stability is given.

Newton iteration results in the following iterative scheme for θ_{n+1} in terms of θ_n:

$$y_n = \cos\left(\boldsymbol{\theta}_n\right)^{-1}\left(\sin\theta_n - \mathbf{I}_c^{-1} I_s\right) \tag{8a}$$

$$\mathbf{J}_n = \mathbf{A}\left[\mathbf{I}_c^{-1}\cos\left(\boldsymbol{\theta}_n\right)^{-1} + \mathbf{L}\right]\mathbf{A}^T \tag{8b}$$

$$j_n = \mathbf{J}_n^{-1}\left(\mathbf{A}\left[\theta_n - y_n - \mathbf{L}I_s\right] + 2\pi(\Phi_{\text{ext}} - z)\right) \tag{8c}$$

$$\theta_{n+1} = \theta_n - \mathbf{I}_c^{-1}\cos\left(\boldsymbol{\theta}_n\right)^{-1}\mathbf{A}^T j_n - y_n \tag{8d}$$

Iteration is done until the residual is below some threshold. This residual is computed as:

$$\text{error}_1 = \frac{\text{norm}(\mathbf{M}\left(\mathbf{I_c}\sin\theta - I_s\right))}{||\mathbf{M}||(\text{norm}(\mathbf{I_c}\sin\theta) + \text{norm}(I_s))} \tag{9}$$

$$\text{error}_2 = \frac{\text{norm}(\mathbf{A}\left(\mathbf{L}\mathbf{I_c}\sin\theta + \theta\right) + 2\pi\Phi_{\text{ext}})}{||\mathbf{A}||(\text{norm}(\mathbf{L}\mathbf{I_c}\sin\theta) + \text{norm}(\theta)) + 2\pi\text{norm}(\Phi_{\text{ext}})} \tag{10}$$

$$\text{error} = \max(\text{error}_1, \text{error}_2) \tag{11}$$

All norms are 2-norms. Note that $0 \leq \text{error} \leq 1$. Other stop conditions are also imposed (some optional), see table 3.1.

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

3.2 approximations to static problems

A static configuration θ with associated vortex configuration n can be approximated quite well, and this approximation can be used as initial condition for the Newton iteration to find an exact solution. For arrays often the arctan approximation is used, but it generalizes non-trivially to general circuits so in pyJJAsim the alternative London approximation is used.

The London approximation uses the property that if the phase zone equals n, it is true that $\theta = \text{pv}(\theta)$, and thus that one can take the approximation $\sin\theta \approx \theta$ with no extra multiples of 2π. Under this assumption $\mathbf{I}_c\theta - I_s$ must lie in the cycle space and equal $A^T x$, which can be plugged into IKVL(n) to solve for x resulting in:

$$\mathbf{J} = \mathbf{A}(\mathbf{I_c}^{-1} + \mathbf{L})\mathbf{A}^T \tag{12a}$$

$$\theta_{\text{approx}} = \mathbf{I_c}^{-1}\mathbf{A}^T\mathbf{J}^{-1}\left[2\pi(n - \Phi_{\text{ext}}) - \mathbf{A}(\mathbf{I_c}^{-1} + \mathbf{L})I_s\right] + \mathbf{I_c}^{-1}I_s \tag{12b}$$

pyJJAsim operates entirely in the phase zone $z = 0$ so it translates the resulting θ to the zero phase zone by adding multiples of 2π appropriately. This is explained in appendix B.

3.3 stability

A static configuration θ_0 obeying equation 6 is stable if J is positive semi-definite where J equals:

$$\mathrm{J} = \begin{bmatrix}\mathbf{M}\\ \mathbf{AL}\end{bmatrix}\mathbf{i_c}\cos\boldsymbol{\theta}\begin{bmatrix}\mathbf{M}^T & \mathbf{LA}^T\end{bmatrix} + \begin{bmatrix}0 & 0\\ 0 & \mathbf{ALA}^T\end{bmatrix} \tag{13}$$

.

Here J equals minus the Jacobian of the time-dependent system, a proof of this theorem is given in the supplementary materials. In pyJJAim stability is checked by first evaluating the simpler criterion $\cos\theta \geq 0$. If this is met the system is stable, but if not, a Choleski factorization is done on J and the system is stable if all diagonal entries of L are greater than or equal to zero.

3.4 Transient analysis

Transient analysis is done with the finite difference method with a constant timestep Δt such that $t_n = n\Delta t$. Backward differences are used to approximate $\dot{\theta}$, $\ddot{\theta}$ and $\sin\theta$ to a precision of $\mathcal{O}(\Delta t^3)$:

$$\dot{\theta}_n = \frac{3}{2}\theta_n - \theta_{n-1} + \frac{1}{2}\theta_{n-2} + \mathcal{O}(\Delta t^3) \tag{14a}$$

$$\ddot{\theta}_n = \theta_n - 2\theta_{n-1} + \theta_{n-2} + \mathcal{O}(\Delta t^3) \tag{14b}$$

$$\sin\theta_n = \sin\left(2\theta_{n-1} - \theta_{n-2}\right) + \mathcal{O}(\Delta t^3) \tag{14c}$$

so

$$\mathbf{I}_c\sin\theta_n + \mathbf{R}^{-1}\dot{\theta}_n + \mathbf{C}\ddot{\theta}_n = \mathbf{I}_c\sin\left(\sum_{i=1}^{d} c_i\theta_{n-i}\right) + \sum_{i=0}^{d}\mathbf{C}_i\theta_{n-i} + \mathcal{O}(\Delta t^{d+1}) \tag{15}$$

.

The source voltage is integrated with the trapezoid rule:

$$\theta_n^V \equiv \int_0^t V_s(t')dt' \approx \Delta t\left(-\frac{V_0^s + V_n^s}{2} + \sum_{i=0}^{n} V_i^s\right) \tag{16}$$

This results in a scheme where θ_n can be obtained from data at previous timesteps as follows:

$$y_n = \eta_n - I_{s,n} + \mathbf{I}_c \sin\left(\sum_{i=1}^{d} c_i \theta_{n-i}\right) + \sum_{i=1}^{d} \mathbf{C}_i \theta_{n-i}. \tag{17a}$$

$$J_n = \left[\mathbf{A}\left(\mathbf{C}_0^{-1} + \mathbf{L}\right)\mathbf{A}^T\right]^{-1} \mathbf{A}\left(\mathbf{C}_0^{-1} y_n - \theta_n^V\right) + 2\pi(z - \Phi_{\text{ext},n}) \tag{17b}$$

$$\theta_n = \mathbf{C}_0^{-1}\left(\mathbf{A} J_n - y_n\right) \tag{17c}$$

Where $J(t)$ are cycle-currents defined by $I(t) - I_s(t) = \mathbf{A}^T J(t)$ following from KCL and the orthogonality of the cut and cycle space.

Backwards differences are needed for stability. This can be shown for the special case of $\mathbf{C} = 0$, where the growth number $J_n \propto G J_{n-1}$ equals:

$$G = \frac{||\mathbf{A}\mathbf{R}\mathbf{A}^T||}{||\mathbf{A}\left(\mathbf{R} + \mathbf{L}\right)\mathbf{A}^T||} \tag{18}$$

As both $\mathbf{R}$ and $\mathbf{L}$ are positive (semi) definite, $0 < G \leq 1$. In a forward differences scheme the growth number would be equal to $1/G$ and thus unstable.

At each timestep a linear system has to be solved with the same matrix. This allows one to lu-factorize the matrix once and use these factors to solve the system at every timestep. In pyJJAsim the scipy implementation of SuperLU is used for this.

The $\sin\theta_n$ term was approximated in terms of data at previous timesteps because θ_n is unknown and solving for it would require solving a nonlinear system at each timestep. This can be done using Newton iteration, but is computationally very expensive as it not only adds extra iterations, but the iterations all involve solving a linear system with a new matrix.

4 Performance comparison

The performance of transient analysis in pyJJAsim is compared with JoSIM on square arrays of varying sizes with a uniform current bias. Circuit1 has no inductance, whereas circuit2 has inductive coupling between junctions up to a distance of 2.1 times the lattice constant (measured from the midpoints of the junctions). The result is shown in figure 2.

JoSIM is faster for small circuits but pyJJAsim is approximately 20x faster for large circuits. The difference may be attributed to the fact that in pyJJAsim at every timestep a linear system of size `face_count` is solved, compared to a system of size `edge_count` + `node_count` in JoSIM, which is generally much larger (3x for a square array).

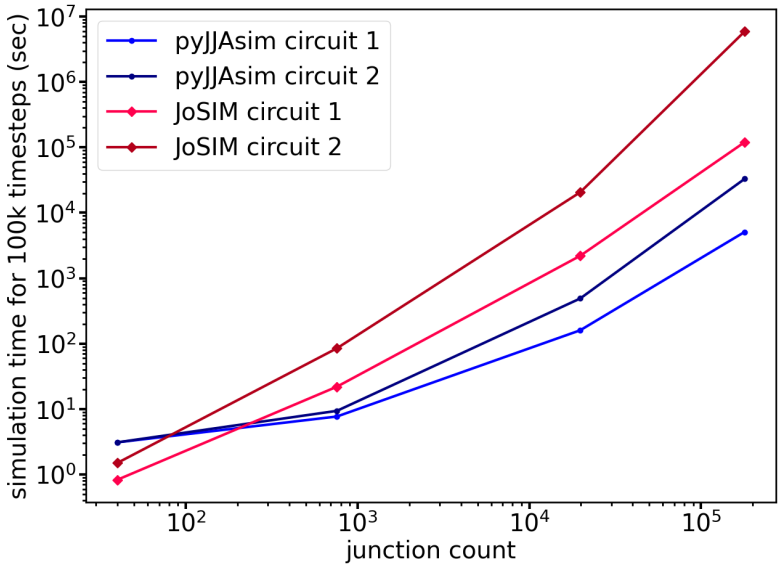

Figure 2: Performance comparison for transient analysis between pyJJAsim and JoSIM. Circuit1 and circuit2 are both square arrays with uniform current bias, but circuit2 has inductive coupling between junctions up to a distance of 2.1 times the lattice constant (measured from the midpoints of the junctions) whereas circuit1 has no inductances. Arrays of four different sizes are benchmarked. For small circuits JoSIM is faster but for large circuits pyJJAsim is up to 20 times faster.

5 Discussion

pyJJAsim is a competitive Josephson circuit simulator for circuits that can be embedded in a 2D plane. It is imagined that for most applications circuits obey this criterion, and it has multiple advantages. Firstly, it offers the unique functionality of computing static configurations and control over Josephson vortices. Secondly, it allows one to compute the cycle space which allows for algorithms that seem significantly faster. Lastly, for the computation of inductances one needs the geometric information anyway and this is more convenient to incorporate in this approach. Possible extensions are conversion to and from netlists, and the inclusion of non-linear resistance.

References

[1] K. K. Likharev and V. K. Semenov, "RSFQ logic/memory family: a new Josephson-junction technology for sub-terahertz-clock-frequency digital systems," *IEEE Transactions on Applied Superconductivity*, vol. 1, no. 1, pp. 3–28, 1991.

[2] S. Whiteley, "Josephson junctions in spice3," *IEEE Transactions on Magnetics*, vol. 27, no. 2, pp. 2902–2905, 1991.

[3] S. V. Polonsky, V. K. Semenov, and P. N. Shevchenko, "PSCAN: personal superconductor circuit analyser," *Superconductor Science and Technology*, vol. 4, pp. 667–670, nov 1991.

[4] P. Shevchenko, "PSCAN2 superconducting circuit simulator," 2016.

[5] E. S. Fang, "A josephson integrated circuit simulator (jsim) for superconductive electronics application," 1989.

[6] J. Satchell, "Stochastic simulation of sfq logic," *IEEE Transactions on Applied Superconductivity*, vol. 7, no. 2, pp. 3315–3318, 1997.

[7] J. A. Delport, K. Jackman, P. l. Roux, and C. J. Fourie, "Josim—superconductor spice simulator," *IEEE Transactions on Applied Superconductivity*, vol. 29, no. 5, pp. 1–5, 2019.

[8] C. J. Fourie, "Digital superconducting electronics design tools—status and roadmap," *IEEE Transactions on Applied Superconductivity*, vol. 28, no. 5, pp. 1–12, 2018.

[9] L. Fu and C. L. Kane, "Superconducting proximity effect and Majorana fermions at the surface of a topological insulator," *Phys. Rev. Lett.*, vol. 100, p. 096407, 2008.

[10] M. Lankhorst, T. Jansen, A. Brinkman, and A. Golubov, "Majorana bound state manipulation by current pulses," *Superconductor science and technology*, vol. 34, Feb. 2021.

[11] D. Domínguez and D. J. V. José, "Magnetic and transport dc properties of inductive Josephson-junction arrays," *Phys. Rev. B*, vol. 53, pp. 11692–11713, 1996.

[12] J. R. Phillips, H. S. J. van der Zant, J. White, and T. P. Orlando, "Influence of induced magnetic fields on the static properties of Josephson-junction arrays," *Phys. Rev. B*, vol. 47, pp. 5219–5229, 1993.

[13] J. R. Phillips, H. S. J. van der Zant, J. White, and T. P. Orlando, "Influence of induced magnetic fields on shapiro steps in Josephson-junction arrays," *Phys. Rev. B*, vol. 50, pp. 9387–9396, 1994.

[14] D. Reinel, W. Dieterich, T. Wolf, and A. Majhofer, "Flux-flow phenomena and current-voltage characteristics of Josephson-junction arrays with inductances," *Phys. Rev. B*, vol. 49, pp. 9118–9124, 1994.

[15] B. D. Josephson, "Possible new effects in superconductive tunnelling," *Physics Letters*, vol. 1, no. 7, pp. 251 – 253, 1962.

[16] S. T. Ruggiero and D. A. Rudman, "Superconducting devices,"

[17] V. Ambegaokar and B. I. Halperin, "Voltage due to thermal noise in the dc Josephson effect," *Phys. Rev. Lett.*, vol. 22, pp. 1364–1366, 1969.

[18] C. J. Lobb, D. W. Abraham, and M. Tinkham, "Theoretical interpretation of resistive transition data from arrays of superconducting weak links," *Phys. Rev. B*, vol. 27, pp. 150–157, 1983.

[19] C. J. S. Teitel, "Resistive transitions in regular superconducting wire networks," *J. Physique Lett.*, vol. 46, no. 1, pp. 33–38, 1985.

[20] M. S. Rzchowski, S. P. Benz, M. Tinkham, and C. J. Lobb, "Vortex pinning in Josephson-junction arrays," *Phys. Rev. B*, vol. 42, pp. 2041–2050, 1990.

[21] M. Lankhorst, A. Brinkman, H. Hilgenkamp, N. Poccia, and A. Golubov, "Annealed low energy states in frustrated large square Josephson junction arrays," *Condensed Matter*, vol. 3, p. 19, Jun 2018.

[22] M. Lankhorst, "JJAsim," 2019, http:/www.jjasim.com.

[23] M. Lankhorst, "pyJJASim," 2022.

A Generate Cycle Matrix A

In a planar embedded graph the cycles surrounding each face span the cycle space. Thus, the problem reduces to finding all faces.

This is done by constructing a map $f(e) \to e'$ from every half-edge to another half-edge. A half-edge is defined as the directed edge between two nodes, and has a tail-node and head-node. f at half-edge e points to a half-edge e' whose tail-node equals the head-node of e for which when traversing from e to e' one makes the sharpest counter-clockwise turn.

A face is constructed by starting at a half-edge e and recursively applying f until one arrives back at e. This procedure is applied to all half-edges to ensure that all faces are found.

This will also generate all boundary-cycles, which envelope an entire connected component in clock-wise direction. These are linearly dependent on the face-cycles and can be removed.

As an optimization, one need not apply the procedure to all half-edges to find all face-cycles. Rather, one can enumerate the nodes and only start at half-edges whose head node has a higher index than the tail node (such edges are called positive half-edges, ones pointing in the opposite direction are called negative half-edges). Every counter-clockwise oriented face-cycle must contain at least one such half-edge and thus all will be found.

These cycles can be combined into the cycle matrix $\mathbf{A}$ of shape N_f (face count) by N_j (junction or edge-count). For a face f, $\mathbf{A}_{fe}$ is assigned $+1$ if positive halve-edge e is in f, -1 if negative half-edge e is in f and 0 otherwise.

The pyJJAsim implementation is vectorized using numpy (so no for loops are used) and makes heavy use of the `repeat` and `cumsum` functions to deal with the fact that all faces can have different lengths.

B Changing phase zone

One can transform a solution $\theta(z)$ in phase zone z (thus obeying system KCL + IKVL(z) to the solution $\theta'(z')$ in phase zone z'. This obeys:

$$\theta'(z') = \theta(z) + 2\pi m \text{ where } \mathbf{A}m = z' - z \tag{19}$$

Where m is a vector of integers. So, changing phase zone requires finding any integral solutions to the under-determined linear system $\mathbf{A}m = z' - z$. The entries of A are either -1, 0 or 1, so Gaussian elimination will result in solutions m with rational entries. Multiplying by the lowest common denominator will result in integers.

In pyJJAsim an alternative algorithm is used which involves the dual graph of faces, called D, over which a tree is defined called T. The nodes of D are the faces of the circuit and two nodes are connected if the faces they represent share an edge. The solution m is constructed by traversing the tree T and accumulating $z' - z$ of all the faces that are traversed. Every time an edge is crossed, add the accumulated value to the corresponding entry in m.

C parameter optimization

The goal of parameter optimization is to find the maximum value of the scalar λ such that a static problem as a function of λ `problem`(λ) has a dynamically stable solution (meaning it obeys eq. 6 and 13 is positive semi-definite).

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 `tol`. It also returns the solution at λ_-. The algorithm looks as follows:

```
def maximal_parameter(problem(λ), init, upper_estimate, tol):
    solution, status = get_solution(problem(0), init)
    if status ≠ 0
        return nan
    λ = upper_estimate
    Δλ = upper_estimate
    init = solution
    error = 1
    while error > tol:
        solution, status = get_solution(problem(λ), init)
        if status == 0:  (solution found)
            λ = λ + Δλ
            init = solution
        if status == 1:  (no solution exists)
            λ = λ − 0.58Δλ
            Δλ = 0.42Δλ
        if status == 2:  (unsure if solutions exists)
            break
        error = Δλ/λ
    λ_ = λ − Δλ
    λ+ = λ
    solution_at_lower = init
    return λ_, λ+, solution_at_lower
```

D Johnson noise

Thermal fluctuations in the current through the resistors can be included by setting the temperature to a nonzero value. This is called Johnson-Nyquist noise and turns the differential equation into a stochastic differential equation. These current fluctuations η represent a Wiener process. This is implemented as:

$$\eta_n = \sqrt{2\Delta t^{-1}\mathbf{R}^{-1}T_n}Z_n \text{ where at each } n,\ Z_n \text{ comes from } N(0,1) \tag{20}$$

Where $N(\mu, \sigma)$ is a normal distribution.

E API overview

E.1 Circuit definition

A circuit (represented as a `Circuit` object) is defined by an embedded graph and component values.

One constructs an embedded graph with an `EmbeddedGraph` object which needs as input coordinates `x`, `y` and edges defined by the indices of the nodes at their endpoints with the fields `node1` and `node2`. Here `x`, `y` are arrays of length equal to the node count, and `node1` and `node2` are arrays with length equal to the junction count. The component values are defined by the fields `critical_currents`, `resistances`, `capacitances` and `inductances`, and are all dimensionless. The first three are arrays of length equal to junction count. `inductance_factors` is a symmetric square (sparse) matrix with self-inductances on the diagonal and couplings as off-diagonal elements.

```
EmbeddedGraph(x, y, node1, node2)

Circuit(embedded_graph, critical_currents, resistances, ...
 capacitances, inductances)
```

E.2 static problems

Static configurations can be found by defining a `StaticProblem` object. It is initialized as follows:

```
StaticProblem(circuit, current_sources, frustration,...
 vortex_configuration)
```

It has the methods `approximate()` and `compute()`:

```
approx_config = static_problem.approximate()

config, status, info = static_problem.compute(initial_config)
```

Both return a `StaticConfiguration` object which contains the approximation or solution respectively. For `.compute()` one can specify an initial guess, but this is optional. If none is specified, the approximation is used as initial guess. One can obtain properties from it about the solution, for example with

the methods `.get_error()`, `.is_stable()`, `.get_vortex_configuration()`, `.get_current()`, etc. `status` indicates if a solution is found and `info` contains information about the Newton iteration. The method `.plot(...)` can be used to visualize the result.

E.3 time evolution

Time evolutions or transient analysis (defined as a `TimeEvolutionProblem` object) require as input a circuit, a timestep, the number of timesteps, sources, frustration, temperature and initial condition(s) for θ and instructions on what data to store:

```
TimeEvolutionProblem(circuit, timestep, timestep_count, ...
 current_sources, frustration, temperature, ...
 config_at_minus1, config_at_minus2, ...
 store_time_steps, store_theta, store_current, store_voltage)
```

Its method `.compute()` returns a `TimeEvolutionResult` object from which one can query properties with the methods `.get_vortex_configuration(time_steps)`, `.get_current(time_steps)`, `.get(time_steps)`, etc. if the data required to compute the properties is stored. The methods `.plot(...)` and `.animate(...)` can be used to visualize the result.

The parameters and sources can be time-dependent and junction/face dependent. Also, multiple time evolutions on the same circuit and time but with different parameters can be run simultaneously. This can be much faster than running them sequentially for small circuits.

Lastly one can define an `AnnealingProblem` where one uses time-evolution to find low-energy static configurations by doing transient analysis with gradually decreasing temperature.