# PyJJASim Whitepaper

Martijn Lankhorst

January 2022

# Contents

# 1 Abstract

This document describes the technical details of PyJJASim, which is an electrical circuit simulator written in python which can include Josephson junctions. It is intended to use on large Josephson Junction Arrays, and supports finding stationary solutions, time evolutions, stability analysis and parameter maximization. Here the mathematical equations are given and motivated, and the implementation details on the numerical procedures used such as Newtons iteration and finite difference schemes are explained.

# 2 Problem definition

## 2.1 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 $F(G) = \{f_1, f_2, ..., f_{N_f}\}$.

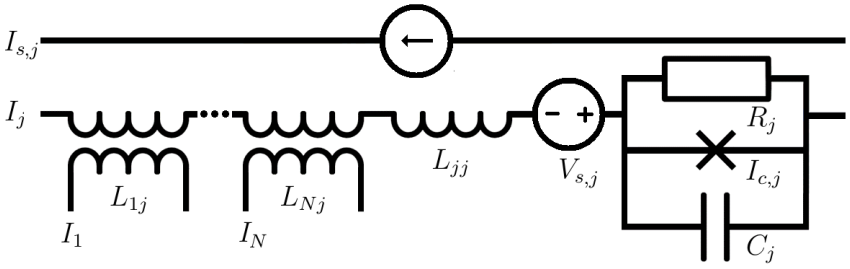


Figure 1: Basic circuit element at each edge of the network.

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 $\mathbf{L}$.

## 2.2 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 $\theta$, which is specified for each junction.

Time evolutions on a circuit require one (or two) initial condition(s) for $\theta$, 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).

## 2.3 —Kirchhoffs rules

The current and voltage of the basic elements at each edge (see figure 1) equals:

$$I(t) = \frac{\hbar}{2e}\mathbf{R}^{-1}\frac{d\theta(t)}{dt} + \eta(t) + \mathbf{I_c}\sin\theta(t) + \mathbf{C}\frac{d^2\theta(t)}{dt^2} \tag{1}$$

$$V(t) = \mathbf{L}\frac{dI(t)}{dt} + V_s(t) + \frac{\hbar}{2e}\frac{d\theta(t)}{dt} \tag{2}$$

| quantity | type (size) | description |
|---|---|---|
| $\theta$ | 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 |
| $\eta$ | 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 |

The circuit is described by Kirchhoff's laws. The current law (shorthand KCL) means the net current injected at each node is zero, the voltage law (KVL) says the sum of the voltage drops along a closed path must be zero (see figure 2). In matrix notation kirchhoff's laws reads:

$$\mathbf{M}(I(t) - I_s(t)) = 0 \tag{3}$$

$$\mathbf{A}V(t) = 0 \tag{4}$$

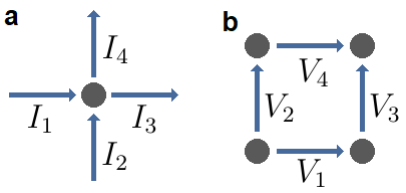


Figure 2: **a** KCL implies $I_1+I_2-I_3-I_4=0$ at a single node. Combined system of equations for all nodes is denoted $\mathbf{M}I=0$. **b** KVL implies $V_1+V_3-V_4-V_2=0$ around a single face. Combined system of equations for all faces is denoted $\mathbf{A}V=0$.

| quantity | type (size) | description |
|---|---|---|
| $\mathbf{M}$ | matrix ($N_n$, $N_j$) | cut matrix |
| $\mathbf{A}$ | matrix ($N_f$, $N_j$) | cycle matrix |

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. This implies that the combined Kirchhoff' laws form a closed system.

It turns out that if a circuit contains Josephson junctions, this is not sufficient to describe the circuit. How this is possible becomes clear when one integrates over the KVL (called IKVL):

$$\int_0^t \mathbf{A}V(t')dt' = \mathbf{A}\left(\mathbf{L}I(t) + \int_0^t V_s(t')dt' + \frac{\hbar}{2e}\theta(t)\right) = \text{constant} \tag{5}$$

Now, if a circuit does not contain Josephson junctions, the integration constant is an artificial quantity that can be chosen freely. However, in the presence of Josephson junctions it does matter because it results in offsetting $\theta$, which in turn changes the current through the current-phase relation.

It turns out the integration constant must equal integer multiples of $2\pi$, called $z$. Note that if $\theta(t)$ solves the system (KCL + IKVL($z$)), there exists a $z'$ for which $\theta(t)+2\pi m$ obeys (KCL + IKVL($z'$)), and thus $\theta$ can be interpreted as an angle. This also means that with no loss of generality one can set $z$ to all zeros, keeping in mind that adding integer multiples of $2\pi$ to $\theta(t)$ may not necessarily solve (KCL + IKVL(0)), but rather one in another phase-zone. In pyJJASim it is assumed $z=0$ and any $\theta$ solves (KCL + IKVL(0)).

$$\text{constant} = 2\pi z \tag{6}$$

$$z \in \mathcal{Z} \tag{7}$$

Lastly, in pyJJASim allows one to define an external magnetic flux through each face, $\Phi_{\text{ext}}(t)$. This must be included in the IKVL, so that the final form of IKVL(0) is:

$$\mathbf{A}\left(\mathbf{L}I(t)+\int_0^t V_s(t')dt'+\frac{\hbar}{2e}\theta(t)\right)+\Phi_{\text{ext}}(t)=0 \tag{8}$$

## 2.4 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 | $V_0$ | $I_0R_0$ | $v=V/V_0$ |
| Magnetic fluxes | $\Phi_0$ | $h/(2e)$ | $f=\Phi/\Phi_0$ (frustration) |
| Times | $t_0$ | $\Phi_0/(2\pi I_0R_0)$ | $\tau=t/t_0$ |
| 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 | $C_0$ | $\Phi_0/(2\pi I_0R_0^2)$ | $c=C/V_0$ |
| 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.

In normalized units, the current, voltage and IKVL(0) become:

$$i(\tau)=\mathbf{r}^{-1}\frac{d\theta(\tau)}{d\tau}+\eta(\tau)+\mathbf{i_c}\sin\theta(\tau)+\mathbf{c}\frac{d^2\theta(\tau)}{d\tau^2} \tag{9}$$

$$v(\tau)=\mathbf{l}\frac{di(\tau)}{d\tau}+v_s(\tau)+\frac{d\theta(\tau)}{d\tau} \tag{10}$$

$$\mathbf{A}\left(\mathbf{l}i(\tau)+\int_0^\tau v_s(\tau')d\tau'+\theta(\tau)\right)+2\pi f_{\text{ext}}(\tau)=0 \tag{11}$$

## 2.5 Other formulas

| 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 | $\boldsymbol{\beta_L}=2\pi\mathbf{L}\mathbf{I_c}/\Phi_0$ | $\boldsymbol{\beta_L}=\mathbf{li_c}$ |
| capacitance parameter | $\boldsymbol{\beta_C}=2\pi\mathbf{I_c}\mathbf{R}^2\mathbf{C}/\Phi_0$ | $\boldsymbol{\beta_C}=\mathbf{i_c}\mathbf{r}^2\mathbf{c}$ |
| 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 |
| $\delta$ | scalar | Dirac delta function |

# 3 Embedded Graph Algorithms

This section describes algorithms used in the EmbeddedGraph class.

## 3.1 Generate Cycle Matrix $A$

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

Faces are generated by traversing the graph counter-clockwise, making the sharpest possible turn at each node, until one arrives back at the starting edge.

The cycle matrix is made from the unique set of cycles that are generated from this procedure by starting at every edge. One only needs to start at half-edges starting in the direction going from the node with lowest id to the node with highest id to ensure all faces are found. This is true because in every face-cycle such a half edge must occur.

This will also generate all "boundary"-cycles, which envelope an entire connected components. These will have opposite signed area compared to face-cycles.

## 3.2 Integral solutions cycle equation

This section describes the algorithm used in PyJJASim to solve equations $\mathbf{A}m = n$ for integral $m \in \mathcal{Z}$ where $n \in \mathcal{Z}$.

This is done by generating a tree on the dual graph of faces, whose nodes are the faces and two nodes are connected if the faces they represent share an edge. The solution $m$ is constructed by traversing the tree and accumulating $n$ of all the faces that are traversed. Every time an edge is crossed, add the accumulated value to the corresponding entry in $m$.

# 4 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. Such states can only exist if all edges contain a Josephson junction. A configuration can be fully described with the vector $\theta$, which must satisfy the following system:

$$\mathbf{M}\left(\mathbf{i_c}\sin\theta - i_s\right) = 0 \tag{12}$$

$$\mathbf{A}\left(\mathbf{li_c}\sin\theta + \theta\right) + 2\pi f_{\text{ext}} = 0 \tag{13}$$

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 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:

$$\mathbf{A}\left(\mathbf{li_c}\sin\theta + \text{pv}(\theta)\right) + 2\pi f_{\text{ext}} = 2\pi n \tag{14}$$

where

$$\text{pv}(\theta) \in [-\pi, \pi) \tag{15}$$

$$n \in \mathcal{Z} \tag{16}$$

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 $\rightarrow$ vector | select principle value of angle between $-\pi$ and $\pi$ |

In pyjjasim, a StaticProblem requires as input a circuit, $I_s$, $f_{\text{ext}}$ 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.

## 4.1 approximations

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

For the arctan approximation, given a set of coordinates $(x_{n_i}, y_{n_i})$ at which $n_i$ vortices are placed, one can get the following approximation:

$$\phi(x,y) = 2\pi \sum_i n_i \text{atan2}(y - y_{n_i}, x - x_{n_i}) \tag{17}$$

$$\theta = (\mathbf{I} + \mathbf{li_c})^{-1} \left( \mathrm{pv}(\mathbf{M}^T \phi) + g \right) + 2\pi \mathrm{round} \left( \frac{\mathbf{M}^T \phi}{2\pi} \right) \tag{18}$$

Here $g$ can be any vector obeying $\mathbf{A}g + 2\pi f = 0$. In pyjjasim the following solution is chosen (it is underdetermined):

$$g = -2\pi \mathbf{A}^T \left( \mathbf{A}\mathbf{A}^T \right)^{-1} f_{\mathrm{ext}} \tag{19}$$

The idea behind this approximation is that if one places vortices sufficiently in the centre of faces, the quantity $\mathbf{A}\mathrm{round}\left(\frac{\mathbf{M}^T\phi}{2\pi}\right)$ equals $-n$. In pyjjasim the funcion approximate() puts the coordinates at face centers whereas approximate_placed_vortices() allows one to manually specify the coordinates.

For the London approximation $\theta$ equals:

$$\theta = 2\pi \left[ \mathbf{A}(\mathbf{I} + \mathbf{li_c})\mathbf{A}^T \right]^{-1} (n - f_{\mathrm{ext}}) - 2\pi m \tag{20}$$

Where the equation $\mathbf{A}m = n$ must be solved for $m \in \mathcal{Z}$. This is a nontrivial problem and described at section 3. (Footnote: if one would operate in the phase zone $z = n$, $m$ would be zero. This is why the first version of the code required one to manually set the phase zone. Solving this problem allowed me to operate entirely in the $z = 0$ phase zone and remove $z$ from the interface, so that the user does not have to deal with all the subtleties of this pesky phase zone.)

| quantity | type (size) | description |
|---|---|---|
| $\mathbf{I}$ | matrix ($N_j$, $N_j$) | Identity matrix |
| $\phi$ | vector ($N_n$) | gauge dependent phase at each node |
| $g$ | vector ($N_j$) | Any vector obeying $\mathbf{A}g + 2\pi f = 0$ |
| $m$ | vector ($N_f$) | Any vector of integers obeying $\mathbf{A}m = n$ |

## 4.2 exact solutions

The exact solution is obtained using Newton iteration.

$$\theta_{n+1} = \theta_n - \begin{bmatrix} \mathbf{M}\mathbf{I_c}\cos\boldsymbol{\theta} \\ \mathbf{A}(\mathbf{I} + \mathbf{l}\mathbf{I_c}\cos\boldsymbol{\theta}) \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{M}\left(\mathbf{i_c}\sin\theta - i_s\right) \\ \mathbf{A}\left(\mathbf{li_c}\sin\theta + \theta - g\right) \end{bmatrix} \tag{21}$$

Or shorthand $\theta_{n+1} = \theta_n - \mathbf{J}(\theta)^{-1} F(\theta)$ where $\mathbf{J}(\theta) = \nabla F(\theta)$ is the Jacobian. The residual is computed as:

$$\mathrm{error}_1 = \frac{\mathrm{norm}(\mathbf{M}\left(\mathbf{i_c}\sin\theta - i_s\right))}{||\mathbf{M}||(\mathrm{norm}(\mathbf{i_c}\sin\theta) + \mathrm{norm}(I_s))} \tag{22}$$

$$\mathrm{error}_2 = \frac{\mathrm{norm}(\mathbf{A}\left(\mathbf{li_c}\sin\theta + \theta - g\right))}{||\mathbf{A}||(\mathrm{norm}(\mathbf{li_c}\sin\theta + \theta) + \mathrm{norm}(g))} \tag{23}$$

$$\text{error} = \max(\text{error}_1, \text{error}_2) \tag{24}$$

All norms are 2-norms. Note that $0 \leq \text{error} \leq 1$. Iteration stops if:

| Stops at iter if | result is | optional |
|---|---|---|
| $\text{error}(\text{iter}) < \text{tol}$ | converged | no |
| $\text{error}(\text{iter}) > 0.5$ | diverged | no |
| $\text{iter} > \text{maxiter}$ | indeterminate | no |
| $\text{get\_n}(\theta(\text{iter})) \neq n$ | diverged | yes |
| $\text{error}(\text{iter}) > \text{error}(\text{iter} - 3)$ | diverged | yes |

## 4.3 stability

To determine if some static configuration $x_0$ is dynamically stable, one wants to write the time-dependent equations as $\dot{x} = f(x)$ at $x_0$ (such that $f(x_0) = 0$) and compute the eigenvalues of the Jaccobian of $f$ at $x_0$ . The time dependent system reads:

$$\begin{bmatrix} \mathbf{M} \\ \mathbf{Al} \end{bmatrix} \left( \mathbf{r}^{-1}\dot{\theta} + \mathbf{c}\ddot{\theta} \right) = - \begin{bmatrix} \mathbf{M}\left(\mathbf{i_c}\sin\theta - i_s\right) \\ \mathbf{A}\left(\mathbf{li_c}\sin\theta + \theta - g\right) \end{bmatrix} \tag{25}$$

Or shorthand $\mathbf{m}(\mathbf{r}^{-1}\dot{\theta} + \mathbf{c}\ddot{\theta}) = -F(\theta)$. It turns out this system is not symmetric. It is symetrized as follows using $\theta \equiv \mathbf{m}^T\theta'$:

$$\mathbf{m}\mathbf{r}^{-1}\mathbf{m}^T\dot{\theta}' + \mathbf{m}\mathbf{c}\mathbf{m}^T\ddot{\theta}' = -F(\mathbf{m}^T\theta') \tag{26}$$

$$\nabla_{\theta'}F(\mathbf{m}^T\theta') = \mathrm{J}(\mathbf{m}^T\theta')\mathbf{m}^T = \mathbf{m}\mathbf{i_c}\cos\boldsymbol{\theta}\mathbf{m}^T + \begin{bmatrix} 0 & 0 \\ 0 & \mathbf{A}\mathbf{l}\mathbf{A}^T \end{bmatrix} \equiv \mathbf{J}'(\theta) \tag{27}$$

Stable at $\theta_0$ obeying $F(\theta_0) = 0$:

$$\text{stable if } \begin{bmatrix} \mathbf{m}\mathbf{r}^{-1}\mathbf{m}^T & \mathbf{m}\mathbf{c}\mathbf{m}^T \\ \mathbf{I} & 0 \end{bmatrix}^{-1} \begin{bmatrix} -\mathbf{J}'(\theta_0) & 0 \\ 0 & \mathbf{I} \end{bmatrix} \text{ is negative definite} \tag{28}$$

This can be reduced to a quadratic eigenvalue problem:

$$\left[\mathbf{J}'(\theta_0) + \mathbf{m}\mathbf{r}^{-1}\mathbf{m}^T\lambda + \mathbf{m}\mathbf{c}\mathbf{m}^T\lambda^2\right] v = 0 \tag{29}$$

Note that the resistance and capacitance terms are positive definite. According to [] this means that the system is stable if $\mathbf{J}'(\theta_0)$ is positive definite, and the resistances and capacitances at the junctions do not matter for stability. Equally, the system is stable iff $\mathbf{J}(\theta_0)$ is positive definite. One can use any of the following variants:

$$\text{stable if } \mathbf{J}'(\theta_0) \text{ is positive definite} \tag{30a}$$

$$\text{stable if } \mathbf{i_c}\cos\boldsymbol{\theta_0} + \mathbf{A}^T\left[\mathbf{AlA}^T\right]^{-1}\mathbf{A} \text{ is positive definite} \tag{30b}$$

$$\text{stable if } \mathbf{Mi_c}\cos\boldsymbol{\theta_0}\mathbf{M}^T \text{ is positive definite and } \mathbf{l} = 0 \tag{30c}$$

This is done in pyjjasim by checking if the lowest eigenvalue exceeds zero. In pyjjasim equation 30**a** is used. It also removes all rows of $\mathbf{m}$ that are zero from $\mathbf{J}'(\theta_0)$ (and removes the same columns to keep it symmetric).

## 4.4 parameter optimization

The goal of parameter optimization is to find the maximum value of the scalar $\lambda$ such that the following system:

$$\mathbf{M}\left(\mathbf{i_c}\sin\theta - i_s(\lambda)\right) = 0 \tag{31a}$$

$$\mathbf{A}\left(\mathbf{li_c}\sin\theta + \theta\right) + 2\pi f_{\text{ext}}(\lambda) = 0 \tag{31b}$$

$$\mathbf{A}\left(\mathbf{li_c}\sin\theta + \text{pv}(\theta)\right) + 2\pi f_{\text{ext}}(\lambda) = 2\pi n \tag{31c}$$

has a solution (stability can optionally be required). This requires one to give an estimate for the upperbound of $\lambda$ `upper_estimate`. The method returns a lower-bound ($\lambda_-$) and an upper-bound ($\lambda_+$) for $\lambda$, 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 $\lambda_-$. The algorithm looks as follows:

```
def maximal_parameter(..., i_s(λ), f_ext(λ), init,
                          upper_estimate, accept_tol):
    solution, status = get_solution(..., i_s(0), f_ext(0), init)
    if status ≠ 0
        return nan
    λ = upper_estimate
    Δλ = upper_estimate
    init = solution
    tol = 1
    while tol > accept_tol:
        solution, status = get_solution(..., i_s(λ), f_ext(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
        tol = Δλ/λ
    λ_ = λ − Δλ
    λ+ = λ
    solution_at_lower = init
    return λ_, λ+, solution_at_lower
```

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

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

# 5 Time Evolution Algorithms

$$\begin{bmatrix}\mathbf{M}\\ \mathbf{Al}\end{bmatrix}\left(\mathbf{r}^{-1}\dot{\theta}+\mathbf{c}\ddot{\theta}+\mathbf{i_c}\sin\theta(\tau)+\bar{\eta}(\tau)\right)=-\begin{bmatrix}\mathbf{M}\left(-i_s(\tau)\right)\\ \mathbf{A}\left(\theta(\tau)+\int_0^\tau V_s(\tau')d\tau'\right)+2\pi f_{\mathrm{ext},n}\end{bmatrix} \tag{32}$$

## 5.1 finite difference scheme

Time discretization $\tau_n = n\Delta\tau$:

$$\theta_n^V \equiv n\Delta\tau\sum_{m=0}^{n-1} V_s(\tau_m) \tag{33}$$

$$\mathbf{r}^{-1}\dot{\theta_n}+\mathbf{c}\ddot{\theta_n}\approx\mathbf{C}_-\theta_{n-1}+\mathbf{C}_\circ\theta_n+\mathbf{C}_+\theta_{n+1} \tag{34}$$

Here $\dot{\theta}$ is approximated with a forward Euler scheme and $\ddot{\theta_n}$ with a central difference scheme. Explicit methods are chosen because implicit methods are orders of magnitude slower per step (for implicit methods the matrix has to be factorized at each Newton step, for explicit methods the same matrix factorization can be used for all timesteps and compute multiple problems simultaneously). Two algorithms are implemented.

## 5.2 first algorithm

$$y_n \equiv \mathbf{C}_-\theta_{n-1}+\mathbf{C}_\circ\theta_n+\mathbf{i_c}\sin\theta_n+\bar{\eta}_n \tag{35}$$

$$\begin{bmatrix}\mathbf{MC}_+\\ \mathbf{AlC}_+\end{bmatrix}\theta_{n+1}=-\begin{bmatrix}\mathbf{M}\left(-i_{s,n}+y_n\right)\\ \mathbf{A}\left(\theta_n+\theta_n^V+\mathbf{l}y_n\right)+2\pi f_{\mathrm{ext},n}\end{bmatrix} \tag{36}$$

This only works if $\mathbf{Al}\neq 0$, meaning that in every closed loop some inductance must occur. To generalize this to "mixed" inductance, one defines the loops surrounding faces with no inductance: $\overline{x}$, and all other loops surrounding faces with $\overline{\overline{x}}$ (so $\overline{\mathbf{Al}}=0$). In the system 36 the equations leading with $\overline{\mathbf{Al}}$ drop out and must be replaced with $\overline{\mathbf{A}}(\theta_{n+1}+\theta_{n+1}^V)+2\pi\overline{f_{\mathrm{ext},n}}=0$

$$\begin{bmatrix}\mathbf{MC}_+\\ \overline{\overline{\mathbf{A}}}\mathbf{lC}_+\\ \overline{\mathbf{A}}\end{bmatrix}\theta_{n+1}=-\begin{bmatrix}\mathbf{M}\left(-i_{s,n}+y_n\right)\\ \overline{\overline{\mathbf{A}}}\left(\theta_n+\theta_n^V+\mathbf{l}y_n\right)+2\pi\overline{\overline{f_{\mathrm{ext},n}}}\\ \overline{\mathbf{A}}\theta_n^V+2\pi\overline{f_{\mathrm{ext},n}}\end{bmatrix} \tag{37}$$

## 5.3 second algorithm

For the case $\beta_L = 0$:

$$x_n = \mathbf{C}_+^{-1}(y_n - i_{s,n}) \tag{38}$$

$$\theta_{n+1}=-x_n-\mathbf{C}_+^{-1}\mathbf{A}^T\left(\mathbf{A}\mathbf{C}_+^{-1}\mathbf{A}^T\right)^{-1}\mathbf{A}(\theta_n^V+2\pi f_{\mathrm{ext},n}-x_n) \tag{39}$$

For the case $\beta_L \neq 0$:

$$\theta_{n+1} = -\mathbf{C}_+^{-1}\left[y_n - i_{s,n} + \mathbf{A}^T\left(\mathbf{A}\mathbf{L}\mathbf{A}^T\right)^{-1}\left(\mathbf{A}\left(\theta_n + \theta_n^V\right) + 2\pi f_{\text{ext},n}\right)\right] \quad (40)$$

The second algorithm does not support mixed inductance.

## 5.4 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 $\eta$ represent a Wiener process. This is implemented as:

$$\bar{\eta}_n = \sqrt{2\Delta\tau^{-1}\mathbf{r}^{-1}T_n'}Z_n \text{ where at each } n,\ Z_n \text{ comes from } N(0,1) \quad (41)$$

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

## 5.5 Stability of finite difference scheme

Both algorithms are unstable $\Delta\tau < \Delta\tau_{\max}$ where $\Delta\tau_{\max}$ is small if $\mathbf{l}$ is small, but reasonably large if $\mathbf{l} = 0$. This is work in progress, future releases may provide more stable schemes and more concrete stability criteria.