User Manual

M.Lankhorst

University of Twente
Faculty of Science and Technology
Netherlands
8-2019

Contents

Chapter 1

release notes

1.1 current version

library	version
JJAsim_2D_network	1.1
JJAsim_2D_visualize	1.1
JJAsim_2D_lattice	in progress
JJAsim_2D_inductance	in progress

1.2 version history

1.2.1 2D_network v1.1 changes

- Replaced terminology `island` with `node`. Variable `Nis` renamed to `Nn`.
- `nodeRemove` and `junctionRemove` function now have extra output variables indicating the new number that are assigned to the nodes and the junctions.
- `stationairyState` has n removed as input. It only served as a check to see if the output state had this configuration. One must now manually check if the output vortex configuration is desired.

1.2.2 2D_visualize v1.1 changes

- Replaced terminology `island` with `node`. Variable `Nis` renamed to `Nn`.
- `circuit` can now display values of the network components (i.e. resistance, critical currents and betaCs)
- `snapshot` and `movie` now have an option to display the junctions as a grid.

Chapter 2

What is JJAsim?

JJAsim is a Josephson circuit simulator especially designed for large regular arrays. The library is written in Matlab and CUDA.

If arrays are lattices, it utilizes a specialized algorithm that scales much better for large arrays and works by exploiting the translational invariance of the array. This algorithm is also well suited for a GPU techniques and a CUDA implementation is provided with state-of-the-art performance.

The library is split into two parts, `JJAsim_network` and `JJAsim_lattice`. The `JJAsim_network` is a conventional Josephson circuit simulator like JSPICE or JSIM, whereas `JJAsim_lattice` is used for arrays that are lattices only.

The JJAsim package has the following features:

- JJAsim can find stationairy solutions and time evolutions.
- It supports passive components like resistors, capacitors and inductors.
- Lattices can have arbitrary unit cells and non-periodic, semi-periodic or periodic boundaries.
- One can create current sources and apply external magnetic fields. Current sources can be time-dependent to model Shapiro steps.
- Can compute Josephson vortices and self-fields.
- One can include thermal fluctuations to model temperature.
- Includes built-in parameter sweeps to compute for example IV curves.
- Provides extensive visualization tools.

Chapter 3

installation guide

Source code can be found at: https://github.com/martijnLankhorst/JJAsim

- Download the folders 2D and method and all its contents.
- Add the folders and all subfolders to the matlab path.
- Examples can be found in 2D\network\examples

Chapter 4

Creating Josephson circuits

4.1 definition of a Josephson junction array

Josephson junction arrays are modeled as electrical circuits containing an extra component type, called a Josephson element. JJAsim uses only one type of instance, called the basic instance and it is has two terminals. It is shown in figure 4.1. An array contains of a set of nodes, a set basic instances and the external current base.
A basic instance has the parameters: the critical current I_c, the shunt resistance R, the shunt capacitance C and the set of self inductance and mutuals inductances with all other instances $\{L_j\}$.
A node is defined with its coordinates in space and the external current base (IExtBase), which lists for each node the proportion of the external current that is injected or ejected from that node. Note that current and voltage sources as instances are not supported.

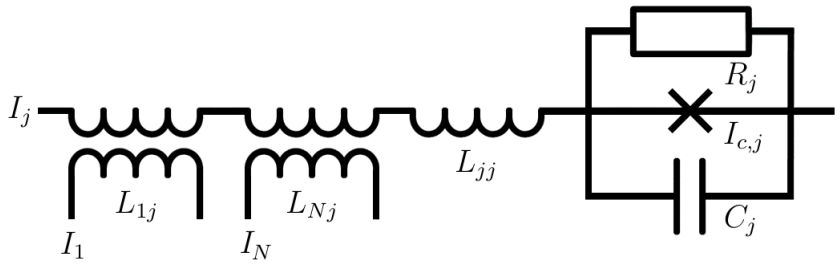

Figure 4.1: Basic instance in the JJAsim package containing a Josephson element with shunt resistor and capacitor in series with an inductor and mutual inductance couplings.

4.2 dimensionless quantities

JJAsim only used dimensionless quantities. The normalizations are shown in table 4.2. The quantity $\Phi_0 = 2.068 \times 10^{-15}$Wb is the magnetic flux quantum and $k_B = 1.381 \times 10^{-23}$J/K is the boltzmann constant. The scalars a, R_0 and I_0 are scaling quantities that can be freely chosen. The dimensionless capacitance and inductance are called `betaC` and `betaL` respectively, the other dimensionless quantities don't get a special name.

Physical quantity	variables	normalization
Current	I Ic IExtBase	$I =$II_0
Voltage	V	$V =$V$I_0 R_0$
Resistance	Rn	$R_n =$RnR_0
coordinate,length	x pathArea	$x =$xa $A =$pathArea$\,a^2$
Magnetic Flux	Phi	$\Phi =$PhiΦ_0
Capacitance	betaC	$C =$betaC$\Phi_0/(2\pi I_0 R_0^2)$
Inductance	betaL	$L =$betaL$\Phi_0/(2\pi I_0)$
time	t	$t =$t$\Phi_0/(2\pi I_0 R_0)$
temperature	T	$T =$T$\Phi_0 I_0/(2\pi k_B)$
Energy	E	$E =$E$2\pi/(\Phi_0 I_0)$

Table 4.1: Normalization table

4.3 Network versus Lattice

A network can be either 2-dimensional or 3-dimensional. A 2D network is created with the JJAsim_2D_network package. However, if the array is lattice, one can use JJAsim_2D_lattice package, which can improve performance. A lattice is a network that can be defined as a repeated unit cell. This section describes how to define 2D arrays, but it generalized trivially to 3D arrays.

4.3.1 create networks

A network is created with the function JJAsim_2D_network_create. Below a basic example is shown:

```
nodePosition = [0,0;0,1;1,1;1,0];
junctionNodes = [1,2;2,3;3,4;4,1];
IExtBase = [-1;-1;1;1];
Rn = 2;
Ic = [1;3;3;2];
array = JJAsim_2D_network_create(nodePosition,junctionNodes,...
      IExtBase,'Rn',Rn,'Ic',Ic);
JJAsim_2D_visualize_circuit(array,'FontSize',30);
```

The resulting network is shown in figure 4.2. Some comments:

- The required fields are nodePosition, junctionNodes and IExtBase. The physical quantities Rn, Ic, betaC, betaL and customcp are optional.
- Physical quantities are displayed in blue. If a physical quantity is a scalar rather than an array, all elements will get the same value. The number on top of the josephson elements is the critical current Ic.

- The external current base is displayed on the nodes. For example, if in a simulation one sets `IExt = 0.2`, then a current of 0.4 will be injected in the bottomleft node and 0.2 will be ejected from each of the right nodes.

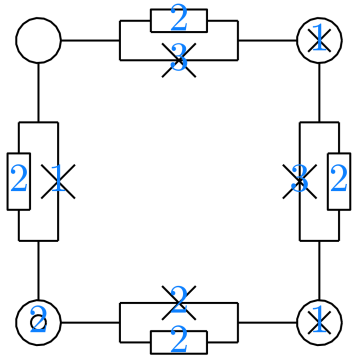

Figure 4.2: Basic network example.

4.3.2 create lattices

not yet implemented

4.4 create standard lattices

Specialized functions exist to easily create standard lattices both in the network and lattice packages. For two dimensions they are:

- JJAsim_2D_network_square
- JJAsim_2D_network_honeycomb
- JJAsim_2D_network_triangular
- JJAsim_2D_lattice_square
- JJAsim_2D_lattice_honeycomb
- JJAsim_2D_lattice_triangular

One can call `array = JJAsim_2D_network_triangular(Nx,Ny)`, which will create a triangular array with `Nx` by `Ny` unit cells and junctions of unit length. One can furthermore set stretch factors in all cardinal directions and choose an external current direction with `'x'` or `'y'`. An example:

```
Nx = 4;
Ny = 3;
ax = 1.5;
ay = 1;
currentDirection = 'x';
array = JJAsim_2D_network_triangular(Nx,Ny,ax,ay,currentDirection);
```

```
JJAsim_2D_visualize_circuit(array,'showQuantitiesQ',false);
```

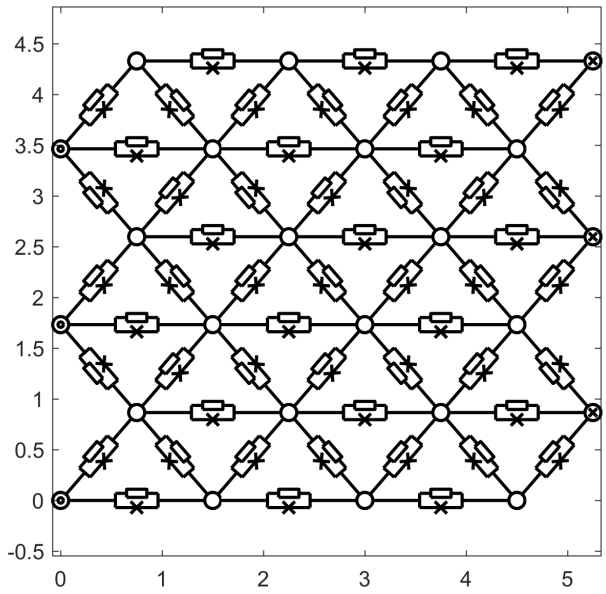

Figure 4.3: Standard lattice example.

4.5 adjusting arrays

One can change physical quantities of an array with the function `JAsim_2D_network_adjust`. One can also remove nodes and junctions with the functions `JJAsim_2D_network_removeNodes` and `JJAsim_2D_network_removeJunctions`. For example:

```
 %create array
nodePosition = [0,0;0,1;1,1;1,0];
junctionNodes = [1,2;2,3;3,4;4,1];
IExtBase = [2;0;-1;-1];
Rn = 2;
Ic = [1;3;3;2];
array = JJAsim_2D_network_create(nodePosition,junctionNodes,...);
    IExtBase,'Rn',Rn,'Ic',Ic);
JJAsim_2D_visualize_circuit(array,'FontSize',30);

%adjust array
RnAdjusted = [1;2;3;4];
removeJunctions = 3;
array = JJAsim_2D_network_adjust(array,'Rn',RnAdjusted);
```

```
array = JJAsim_2D_network_removeJunctions(array,removeJunctions);
figure; JJAsim_2D_visualize_circuit(array,'FontSize',30);
```

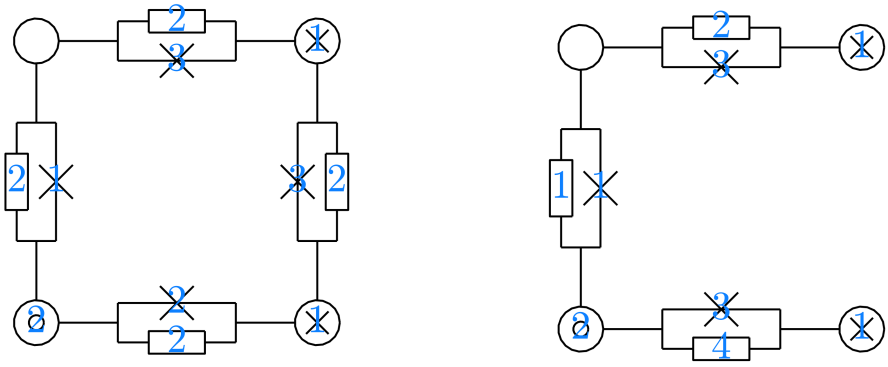

Figure 4.4: Adjusted array example.

Chapter 5

Computing stationairy states and time evolutions

JJAsim can compute two types of problems, stationairy states and time evolutions. Furthermore, approximate stationairy states can be computed, which can be used as an initial guess or initial condition. This chapter gives a quick introduction with examples. Note that all physical quantities are dimensionless, the normalizations are specified in table 4.2.

5.1 stationairy states

A stationairy state can be computed with `JJAsim_2D_network_stationairyState`. It requires an initial guess `th1` and uses newtons iteration to find a nearby solution in parameter space. The physical quantity θ is the gauge invariant phase difference, which is a quantity associated with each junction. Section 5.1.1 explains how to get an initial condition. Furthermore, some external parameters can be set. For example the external current `IExt`, the frustration factor `f` and the phase zone `z`. One can choose what physical quantities to output, for example the current in each junction `I`, the vortex configuration `n` or the gauge invariant phase differences `th`. One can use the fuction `JJAsim_2D_visualize_snapshot` to visualize the output.

To compute Kirchhoff's voltage rule, a set of closed paths is required. This is automatically computed when defining an array. One can access this for example with the fields `array.pathNodes`, which lists the nodes that each path contains, and `array.Np`, which gives the number of paths (the circuit rank). The frustration factor is the normalized magnetic flux per path, and can be specified for each path separately in an array to compute an inhomogeneous external magnetic field. Similarly, the phase zone can be specified for each path separately.

Another important quantity is the amount of Josephson vortices. Each path contains a number of Josephson vortices `n`. The number of vortices associated with a state (`th`,`z`) and can be found with `n = JJAsim_2D_network_method_getn(array,th,z)`. However, one cannot specify the number of Josephson vortices as input. Rather, one can specify an approximate initial guess which encodes the desired vortex configuration and the function `JJAsim_2D_network_stationairyState` will generally succeed to find a stationary state with the same vortex configuration, if it exists.

5.1.1 initial guess

An initial guess can be computed with `JJAsim_2D_network_stationairyState_approx_arctan` or `JJAsim_2D_network_stationairyState_approx_london`. They require as input the array, the frustration factor and the vortex configuration. They differ in two ways, firstly how to specify the vortex configuration and secondly in the phase zone in which the output resides.

First lets explain what the phase zone is. The phase zone `z` is a set of integers, one associated with each path. It determines in what phase zone any state `th` resides. With this is meant that, because `th` are angles, one can freely add multiples of 2π without changing the physical state. Changing the phase zone will result the same physical state, but some entries of `th` will differ by a multiple of 2π. One can change the phase zone of a state (`th_old,z_old`) to (`th_new,z_new`) with the function `th_new = JJAsim_2D_network_method_changePhaseZone(th_old,z_old,z_new)`. For periodic lattices, it is required that `z = n`. Otherwise any phase zone can be used.

`approx_arctan` specifies vortices as a list of coordinate and vorticity pairs, and allows vortices to be placed anywhere throughout the array. The vorticity of a path will then be the sum of the vorticities of all vortices enclosed in that path. However, it can happen that this rule does not hold, for example if the vorticity exceeds twice the number of nodes in a path. If this happens, the function will give a warning. The output resides in the phase zone `z = 0` for all paths. If one uses the output `th` as an initial guess for `JJAsim_2D_network_stationairyState`, one must make sure to set the input `z = 0`.

With `approx_london` one specifies the vorticity of each path with `n`. Again, the output `th` might not encode this vortex configuration, in which case a warning is given. The output resides in the phase zone `z = n`. If one uses the output `th` as an initial guess for `JJAsim_2D_network_stationairyState`, one must make sure to set the input `z = n`.

In the following code snippet, an initial guess computed with both `approx_arctan` and `approx_london` and visualized. One can see it results in the same physical solution.

```
 %set array and frustration factor
array = JJAsim_2D_network_square(4,4);
f = 0;

%compute initial guess with arctan approximation
n_arctan = 1;
x_arctan = 1.5;
y_arctan = 1.5;
[th_arctan,I_arctan,n_arctan] = ...
      JJAsim_2D_network_stationairyState_approx_arctan(...
      array,x_arctan,y_arctan,n_arctan,f);

%compute initial guess with london approximation
n_london = zeros(array.Np,1);
```

```
n_london(5) = 1;
[th_london,I_london] = ...
      JJAsim_2D_network_stationairyState_approx_london(...
      array,n_london,f);

%visualize initial guesses
JJAsim_2D_visualize_snapshot(array,n_arctan,I_arctan);
figure;
JJAsim_2D_visualize_snapshot(array,n_london,I_london);
```

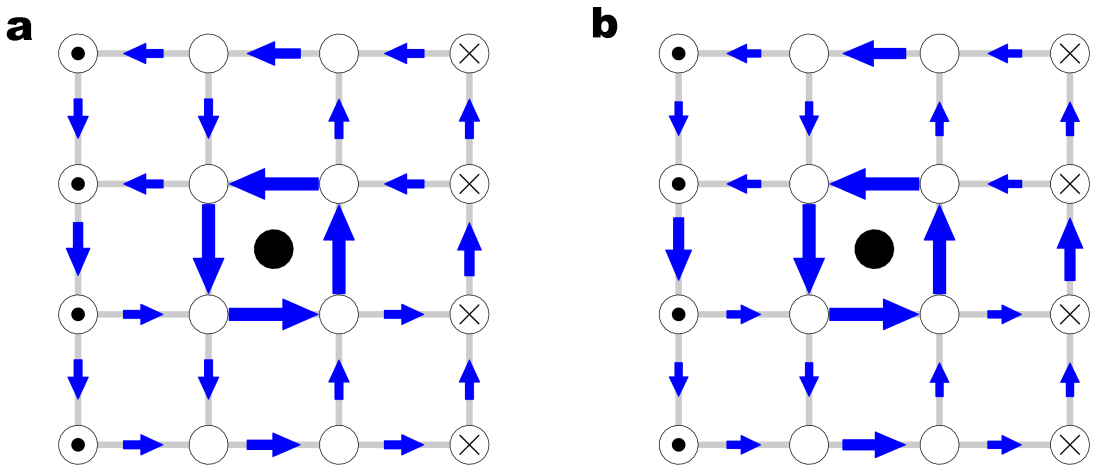

Figure 5.1: **a** Approximate state computed with the arctan approximation, **b** Approximate state computed with the london approximation. Note that they are not exactly the same, and that both do not obey Kirchhoffs current rules as they are approximate solutions.

5.1.2 exact stationairy states

To compute an exact staionairy state, one must specify the array, the external current `IExt`, the frustration factor `f`, the phase zone `z` and an initial guess `th1`. One must also specify `inputMode`, but this is only relevant if one wants to compute several problems in one function call, this is discussed in section 5.3. For now, it is set to `'sweep'`. Some optional parameters can be set regarding the newton-iteration. The output is a struct whos fields include physical parameters like `th`, `I` and `E`. One can disable output quantities to reduce memory consumption if needed. Lastly, the output struct contains the fields `solutionQ`, which is true if a valid stationairy state is found, and `nrOfSteps` which shows the number of newton iterations that were executed. The following programming example shows this in action. The resulting state is very close to the initial guess.

```
%set array and frustration factor
array = JJAsim_2D_network_square(4,4);
f = 0;

%compute initial guess
n = zeros(array.Np,1);
n(5) = 1;
th1 = JJAsim_2D_network_stationairyState_approx_london(...
     array,n,f);
z = n;

%find exact stationairy state
inputMode = 'sweep';
IExt = 0;
out = JJAsim_2D_network_stationairyState(array,inputMode,...
     IExt,f,z,th1);

%display if a valid solution is found
if out.solutionQ
   disp(['valid solution found in ',num2str(out.nrOfSteps),...
      '  iterations']);
else
   disp('no valid solution found');
end

%visualize stationairy state
JJAsim_2D_visualize_snapshot(array,out.n,out.I);
```

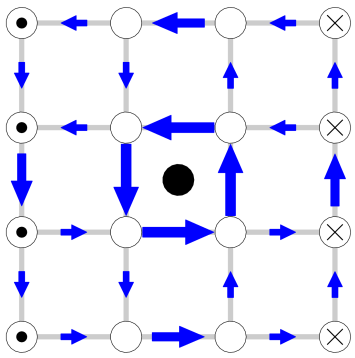

Figure 5.2: Exact stationairy state solution.

5.2 time evolution

Time evolutions can be computed with `JJAsim_2D_network_simulate`. It requires an initial condition `th1` and the time vector `t` which is a `Nt` by `1` array containing the time points to simulate over. Furthermore it requires the same input as `JJAsim_2D_network_stationairyState`, so `array`, `inputMode`, `IExt`, `f` and `z`. On top of that one must also specify the temperature `T`. Note that `IExt` and `T` can be specified for each timepoint, and `f` and `z` can be specified for each path. Lastly, if the array contains capacitors, a second initial condition `th2` must be specified.

The possible output quantities are the gauge invariant phase differences `th`, the junction current `I`, the junction voltage drop `V`, the total array voltage drop `Vtot` and the array total energy `E`. One can restrict the output to reduce memory consumption in two ways. One can choose if one wants to include quantity `X` in the output with `storeXQ`, and one can choose specific timepoints to output with `XTimePoints`, which is an `Nt` by `1` logical array. Time evolutions can be visualized with a movie with the function `JJAsim_2D_visualize_movie`.

One important parameter is the timestep. For accurate simulation, one must obey $\Delta t \ll 1$ if no inductance is present. In the case of a constant self inductance, $\Delta t \ll \beta_L$. The next coding example shows a time evolution in action:

```
%set array and frustration factor
array = JJAsim_2D_network_square(6,6);
f = 0;

%compute initial guess
n = zeros(array.Np,1);
n(18) = 1;
th1 = JJAsim_2D_network_stationairyState_approx_london(array,n,f);
z = n;

%compute time evolution
t = (0:0.05:30)';
T = 0;
inputMode = 'sweep';
IExt = 0.3;
outSim = JJAsim_2D_network_simulate(array,t,inputMode,IExt,T,f,z,th1);

%visualize time evolution
nSim = JJAsim_2D_network_method_getn(array,outSim.th,z);
JJAsim_2D_visualize_movie(array,t,nSim,outSim.I);
```

In this time evolution a vortex is pushed out of the array by the external current, which exerts a force on the vortex perpendicular to the direction of the current. Figure 5.3**a** shows the first snapshot of the time evolution, the black arrow indicates the direction in which the vortex is pushed. Figure 5.3**c** shows

the 2π phase slips that occur in the junctions (labeled in 5.3**b**) when a vortex moves over it. Figure 5.3**d** shows that whenever a phase slip occurs, a voltage spike is observed. Note that if the external current would sufficently small (`IExt` $\lesssim 0.1$), the vortex would not move and the system would reach a stable stationairy state.

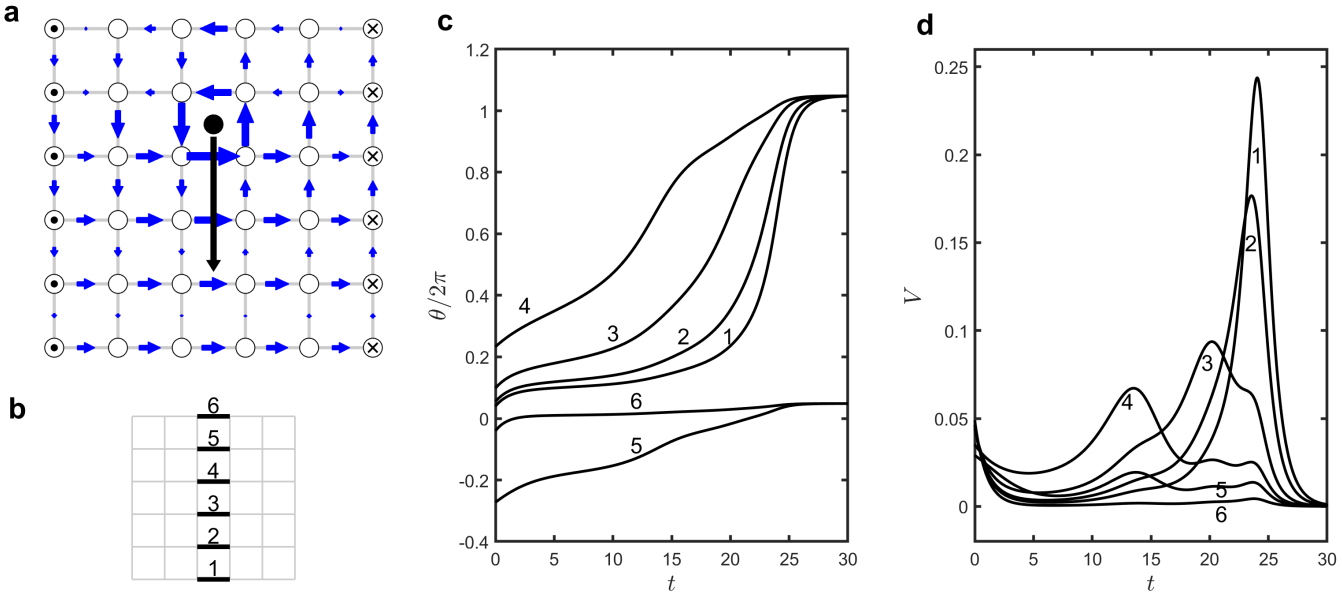

Figure 5.3: Example of a time evolution where a vortex is pushed out of the array by the external current. The gauge invariant phase difference and the voltage is shown for a selection of junctions to visualize 2π phase slips.

5.3 multiple problems and parallel processing

For both `JJAsim_2D_network_stationairyState` and `JJAsim_2D_network_simulate` one can compute multiple problems in one function call. This will generally be more efficient than computing the problems one at a time. To enable parallel processing, these problems can be partitioned, where each partition can be computed on its own core (CPU) or device(GPU). This section described how to speficy multiple problems for `JJAsim_2D_network_simulate` as an example, but the ideas are the same for stationairy states.

There are two modes for multi-problems, specified with `inputMode`. The options are `'sweep'` and `'list'`. The relevant input variables for multi-problems are `IExt`, `T`, `f`, `z` and `th1` (and `th2`). Table 5.3 shows the required sizes for these inputs in the two input modes.

If `inputMode = 'list'`, the number of problems is `W`, and all quantities must have length `1` or `W` in the `W`-direction. If the length is `1` while `W` is not, the specified value holds for all problems.

The mode `inputMode = 'sweep'` is designed to do parameter sweeps with `IExt`, `T` and `f`. These quantities have their own list of problems, and all possible combinations of problems are computed such

inputMode	'sweep'	'list'
IExt	WI by Nt*	W* by Nt*
T	WT by Nt*	W* by Nt*
f	Np* by Wf*	Np* by W*
z	Np* by Wf*	Np* by W*
th1 (and th2)	Nj* by W*	Nj* by W*

that the total number of problems `W = WI*WT*Wf`. Note that `f` and `z` share a problem list (they can't have a different (non-unity) number of problems), and that the initial conditions are tied to the total problem list. The sweep-order for the total problem list is (`IExt`,`T`,`f`).

One can repeat the entire problem set multiple times with the field `repetitions`. The total number of problems `Wtot = W*repetitions`.

The option `parallelQ` must be set to `true` to enable parallel processing, where the number of cores (CPU) or devices(GPU) are set with `cores`. If `cores = 0` (the default value), the maximum amount of available recourses are allocated. However, to do parallel processing, one must define multiple partitions. The partitions are distributed over the cores as the cores become available. One can set the number of partitions with `computePartitions`.

Chapter 6

Example code

6.1 example 1: single vortex

The example code 1 computes a stationairy vortex state in a square array with and without self inductance. In the case of self inductance, the magnitude of the current will drop off faster the further one moves away from the vortex, this is why inductance is sometimes also called screening. The magnitude of the currents is also smaller. This is because the gauge invariant phase differences are closer to zero, as a portion of the phase winding is relieved by inducing magnetic flux in the vortex site. In the limit $\beta_L \to \infty$ the currents will tend to zero and the magnetic flux tends to Φ_0 at the vortex site.

```
% - this example shows how to compute a stationairy single vortex state
%   in a square array.
% - It does this in an array with and without an inductance parameter.

%array size
Nx = 10; %nr of islands in x-direction
Ny = 10; %nr of islands in y-direction
ax = 1; %island spacing in x-direction
ay = 1; %island spacing in y-direction

%vortex configuration
x_n = 4.5; %(x_n,y_n) is the coordinate for the placed vortex
y_n = 4.5;
n = 1; %n is the vorticity of the placed vortex

%other parameters
f = 0;                %frustration factor
inputMode = 'sweep'; %input mode
IExtDirection = 'x'; %external current direction, either 'x' or 'y'
IExt = 0;             %External current
betaLList = [0,3];    %inductance parameter.  Each junction has a self
```

```
                    %inductance of L = betaL*Phi0/(2*pi*I0)

%compute stationairy states
for i = 1:2
   betaL = betaLList(i);

   %create square array
   array = JJAsim_2D_network_square(Nx,Ny,ax,ay,IExtDirection,...
      'betaL',betaL);

   %make initial guess
   [th,I] = JJAsim_2D_network_stationairyState_approx_arctan(array,...
      x_n,y_n,n,f);
   z = 0; %When using approx_arctan, the phase zone z must be zero.

     %compute exact stationairy state
   out = JJAsim_2D_network_stationairyState(array,inputMode,IExt,f,z,th);

   %vortex configuration of stationairy state
   nOut = JJAsim_2D_network_method_getn(array,out.th,z);

   %compute island phases
   phiOut = mod(JJAsim_2D_network_method_getphi(array,out.th),2*pi);

   %display stationairy state
   figure;
   if i == 1
     title('no inductance parameter')
   else
     title(['inductance parameter $ \beta_L = $',num2str(betaL)])
   end
   JJAsim_2D_visualize_snapshot(array,nOut,out.I,'showNodeQuantityQ',...
      true,'nodeQuantity',phiOut,'nodeColorLimits',[0,2*pi],...
      'nodeQuantityLabel','$\phi$','showIExtBaseQ',false,...
      'nodeDiameter',0.35,'FontSize',20);
end
```

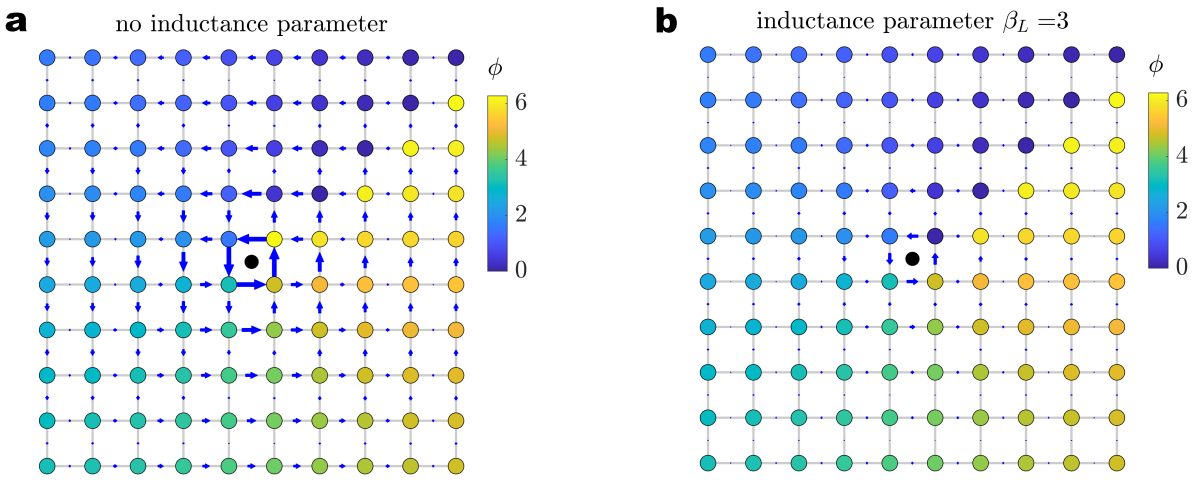

Figure 6.1: **a** Vortex configuration without inductance. **b** vortex configuration when all junctions have a self inductance parameter $\beta_L = 3$.

6.2 example 2: annealing

Example code 2 shows how one can find low-energy states with simulated annealing. This involves doing a time simulation with a initial state `th1` far from equilibrium, and allowing the system to find lower energy states by providing the system with thermal fluctuations induced by the temperature. The temperature profile is a linear quench to zero in this example.

```
% - this example shows how to find low energy stationairy states with
%   simulated annealing.
% - Done in the presence of an external magnetic field, which is
%   represented by the frustration factor f.

%array size
Nx = 10; %nr of islands in x-direction
Ny = 10; %nr of islands in y-direction
ax = 1;  %island spacing in x-direction
ay = 1;  %island spacing in y-direction

%other parameters
f = 1/3;                %frustration factor
inputMode = 'sweep';    %input mode
IExt = 0;               %External current
t = (0:0.5:20000)';     %time points
Nt = length(t);
z = 0;                  %phase zone
th1 = 0;                %initial condition
T = linspace(0.15,0,Nt); %annealing temperature profile

%create square array
array = JJAsim_2D_network_square(Nx,Ny);

%time evolution
out = JJAsim_2D_network_simulate(array,t,inputMode,IExt,T,f,z,th1);
nOut = JJAsim_2D_network_method_getn(array,out.th,z);

%visualize annealing process
selectedTimePoints = false(Nt,1);
selectedTimePoints(1:20:end) = true;
JJAsim_2D_visualize_movie(array,t,zeros(array.Np,Nt),out.I,...
  'showPathQuantityQ',true,'pathQuantity',nOut,'selectedTimePoints',...
  selectedTimePoints,'showIslandsQ',false,'arrowColor',[0,0,0],...
  'pathQuantityLabel','n');
```

```
E = out.E(end)/array.Nj;
disp(['energy of final state (per junction):  ',num2str(E)])
```

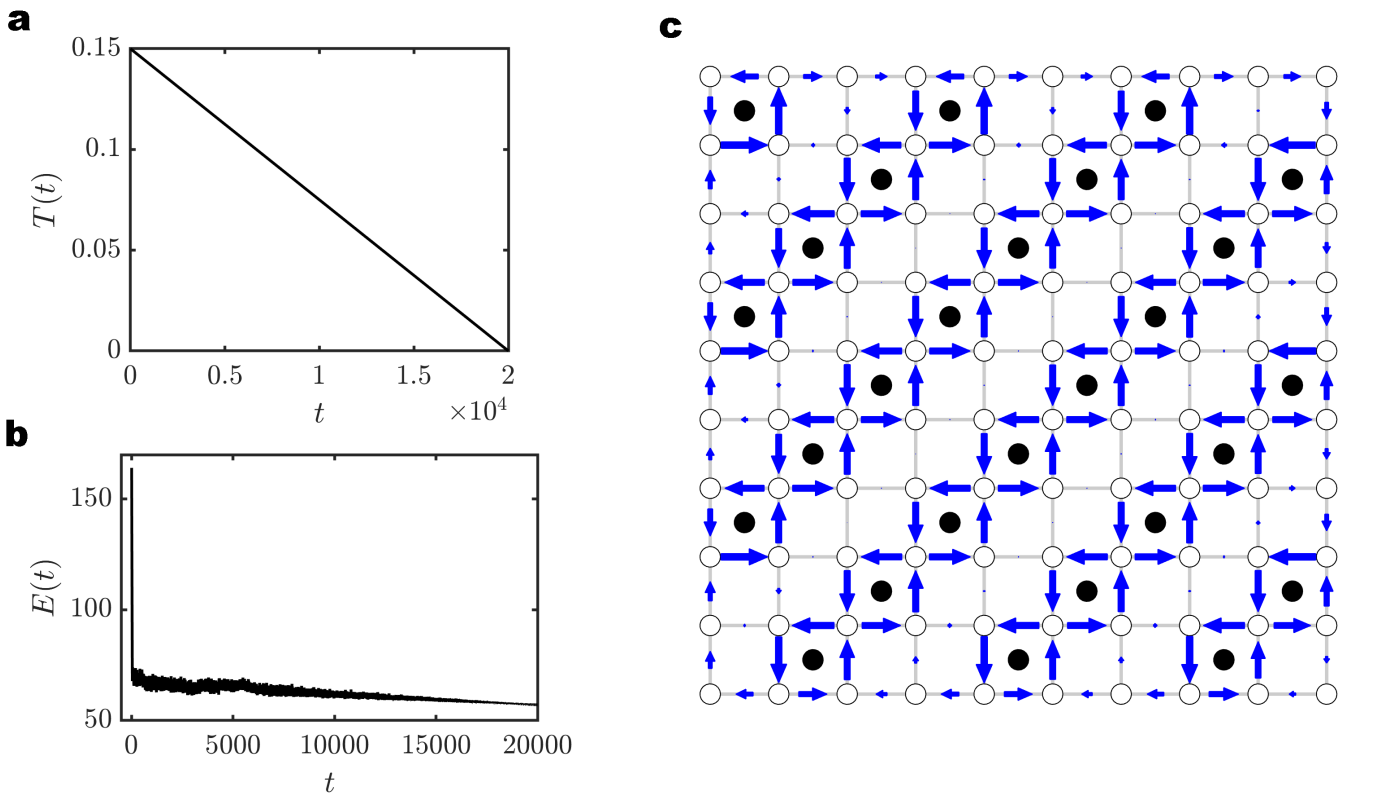

Figure 6.2: **a** Annealing temperature profile. **b** Total energy in array as a function of time. **c** Final vortex configuration. This annealing attempt succeeded in finding the ground state but it will sometimes get stuck in a higher energy state.

6.3 example 3: biassed disordered array

Example code 3 computes the time evolution of a square array that has a fraction of the node removed and is biassed with an external current. Furthermore, an external magnetic field is applied such that `f = 0.2`. Note that the `IExtBase` is automatically rescaled such that it satisfies `sum(IExtBase) = 0`. The external current applies a Lorentz force on the vortices in the negative y-direction which moves the vortices. Secondly, if the vorticity of a path is larger than 1, it is displayed with a symbol where the number of rings is equal to the vorticity. Anti-vortices are displayed in red.

```
% - this example shows the time evolution on a current-biassed
%   disordered square array where some islands are removed.  For
%   reference it also shows a biassed normal square array.

%array size
Nx = 16; %array size
Ny = 16;
nodeRemoveFraction = 0.2; %fraction of nodes to remove

%other parameters
f = 0.2;              %frustration factor
inputMode = 'sweep'; %input mode
IExt = 0.5;          %External current
t = (0:0.1:100)';     %time points
Nt = length(t);
th1 = 0;              %initial condition
z = 0;                %phase zone
T = 0;                %temperature

%create disordered square array
array = JJAsim_2D_network_square(Nx,Ny);
removeNodes = find(rand(array.Nn,1) < nodeRemoveFraction);
array = JJAsim_2D_network_removeNodes(array,removeNodes);

%time evolution
out = JJAsim_2D_network_simulate(array,t,inputMode,IExt,T,f,z,th1);

%compute output vortex configuration
nOut = JJAsim_2D_network_method_getn(array,out.th,z);

%visualize annealing process
selectedTimePoints = false(Nt,1);
selectedTimePoints(1:2:end) = true;
JJAsim_2D_visualize_movie(array,t,nOut,out.I,'selectedTimePoints',...
```

```
    selectedTimePoints,'arrowWidth',0.6);
```

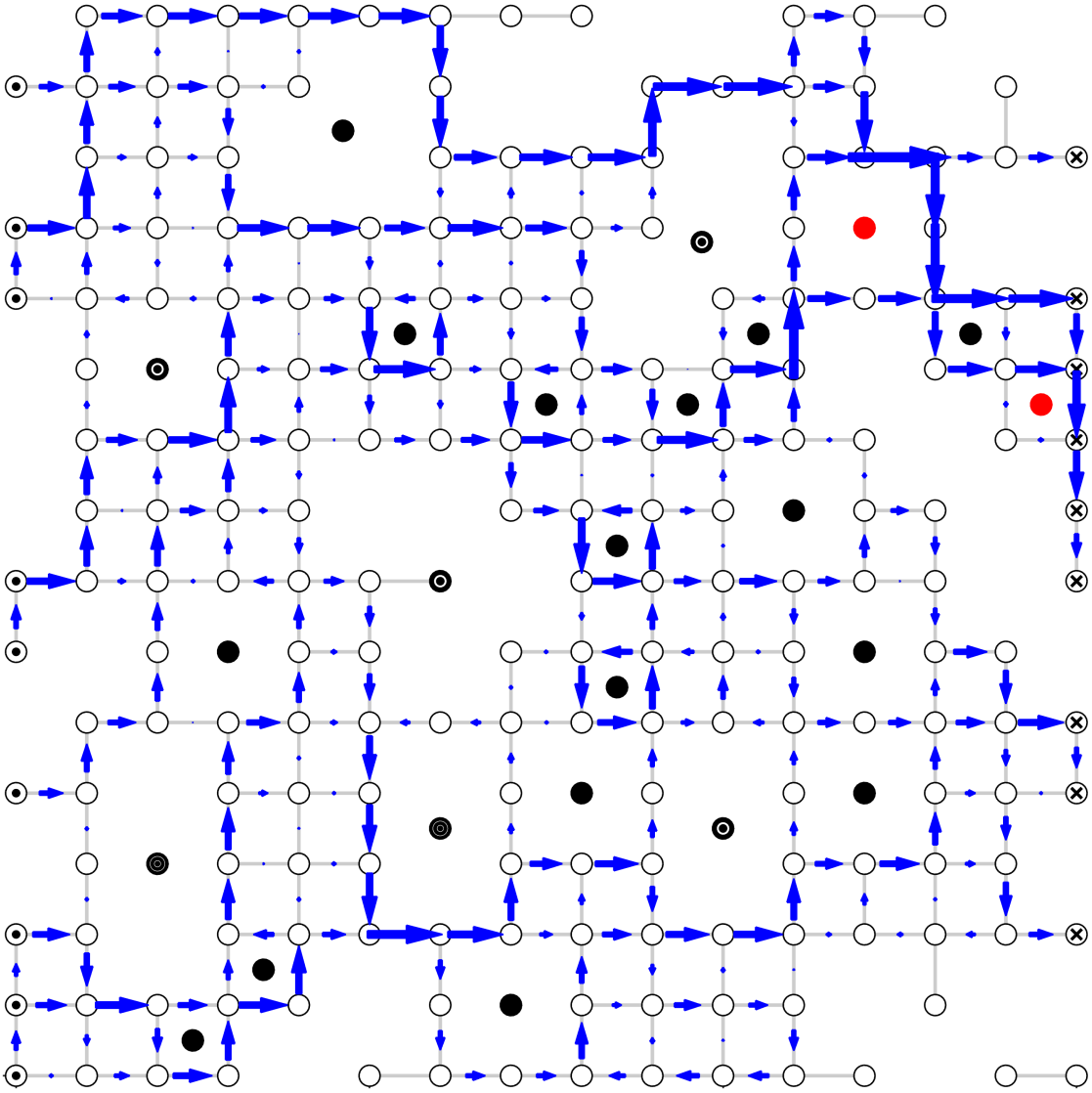

Figure 6.3: Snapshot of a disordered array at the last timestep.

6.4 Example 4: Resistance versus temperature plot

Example code 4 computes a RT (resistance versus temperature) curve. Uses the feature that multiple problems can be computed in a single function call. The resistance is based on the average of the total voltage over the last twothird of the time range.

```
% - Computes an RT curve (resistance versus temperature)

%array size
Nx = 20;
Ny = 20;

%create square array
array = JJAsim_2D_network_square(Nx,Ny);

%other parameters
f = 0; %frustration factor
inputMode = 'sweep'; %input mode
IExt = 0.1; %Bias current
t = (0:0.1:1000)'; %time points
Nt = length(t);
z = 0; %phase zone
th1 = 0; %initial condition
T = linspace(0,2,100)'; %temperature list

%time evolution (storethQ and storeIQ are set to false to reduce memory)
out = JJAsim_2D_network_simulate(array,t,inputMode,IExt,T,f,z,th1,...
'storethQ',false,'storeIQ',false,'storeVQ',false);

%compute array resistance
R = mean(out.Vtot(:,round(Nt/3):end),2)/IExt/(Nx-1);

%plot RT curve
plot(T,R)
xlabel('T')
ylabel('R')
```

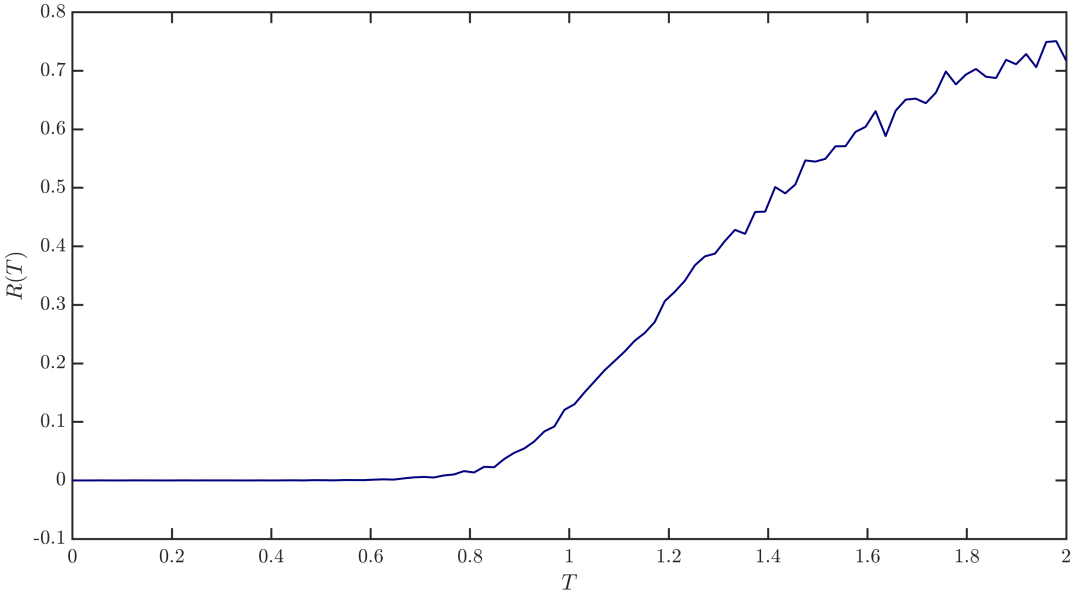

Figure 6.4: Resistance versus temperature curve for a square array.

6.5 Example 5: Magnetoresistance

Example code 5 computes the magnetoresistance for a square array. Note that the magnetoresistance is periodic in `f` with period 1.

```
% - magnetoresistance curves for several bias currents

%array size
Nx = 20;
Ny = 20;

%create square array
array = out = JJAsim_2D_network_square(Nx,Ny);

%other parameters
f = linspace(0,2,101);          %frustration factor
inputMode = 'sweep';            %input mode
IExt = [0.1,0.3,0.5,0.8,1.2]'; %Bias current
t = (0:0.1:500)';               %time points
Nt = length(t);
z = 0;                          %phase zone
th1 = 0;                        %initial condition
T = 0.1;                        %temperature
parallelQ = true;               %true to use multiple cores.
partitions = 4;

%time evolution (storeQs set to false to reduce memory)
out = JJAsim_2D_network_simulate(array,t,inputMode,IExt,T,f,...
     z,th1,'storethQ',false,'storeIQ',false,'storeVQ',false,...
     'parallelQ',parallelQ,'computePartitions',partitions);

%compute array resistance
V = mean(out.Vtot(:,round(Nt/3):end),2)/(Nx-1);
V = reshape(V,length(IExt),length(f));
R = V./IExt;

%plot RT curve
plot(f,R,'LineWidth',1.5)
xlabel('f')
ylabel('R')
```

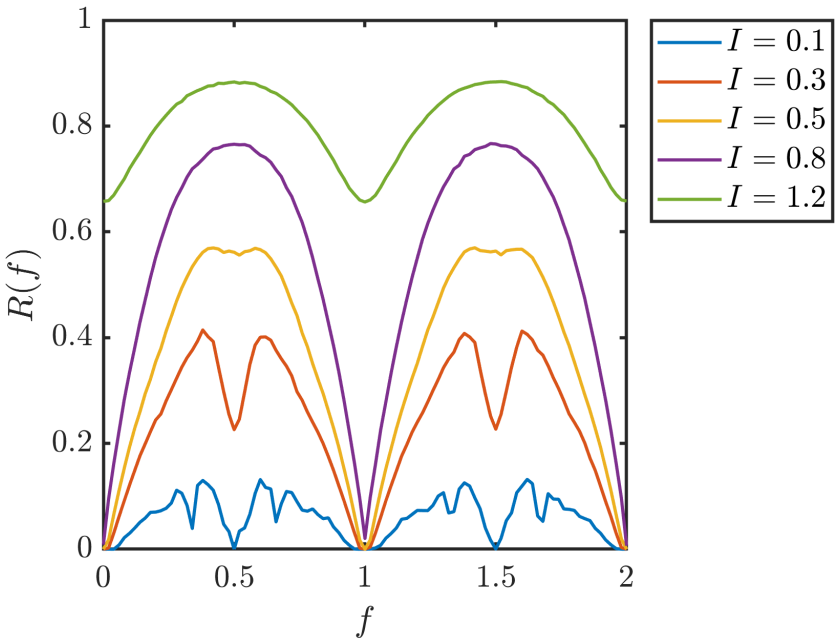

Figure 6.5: Magnetoresistance for a square array.

6.6 Example 6: Giant Shapiro steps

Example code 6 computes giant Shapiro steps in a square array. In the presence of electromagnetic radiation IV-curves of Josephson junctions exhibit steps, where the height of the steps is proportional to the frequency of the radiation. These steps are called Shapiro steps. For arrays, one gets larger steps which are proportional to the number of junctions resonating in series. In this example, there are 20 junctions and the angular frequency `freq = 0.25`, so one expects a step height of 5, which one indeed observes. In this example, the radiation is modeled by an added AC component to the external current inserted in the nodes. Two problems are computed in the same function call, one with an AC amplitude of `IAC1 = 0.5` and one with `IAC2 = 1`.

```
% - magnetoresistance curves for several bias currents

%array size
Nx = 21;
Ny = 21;

%create square array
array = JJAsim_2D_network_square(Nx,Ny);

%time
t = (0:0.1:300)';
Nt = length(t);

%bias current components
freq = 0.25;
Amp1 = 0.5;
Amp2 = 1;
IDC = linspace(0,2,201)';

IAC1 = Amp1*sin(freq*t)';
IAC2 = Amp2*sin(freq*t)';
IExt = [IDC + IAC1;IDC + IAC2];

%other parameters
f = 0;                %frustration factor
inputMode = 'sweep'; %input mode
z = 0;                %phase zone
th1 = 0;              %initial condition
T = 0.01;             %temperature

%time evolution (storeQs are set to false to reduce memory)
out = JJAsim_2D_network_simulate(array,t,inputMode,IExt,T,f,z,...
     th1,'storethQ',false,'storeIQ',false);
```

```
%compute array resistance
V = mean(out.Vtot(:,round(Nt/3):end),2);
V = reshape(V,length(IDC),2);

%plot IV curve
plot(IDC,V,'LineWidth',1.5)
xlabel('I')
ylabel('V')
legend(['Amplitude = ',num2str(Amp1)],['Amplitude = ',num2str(Amp2)],'Location
```

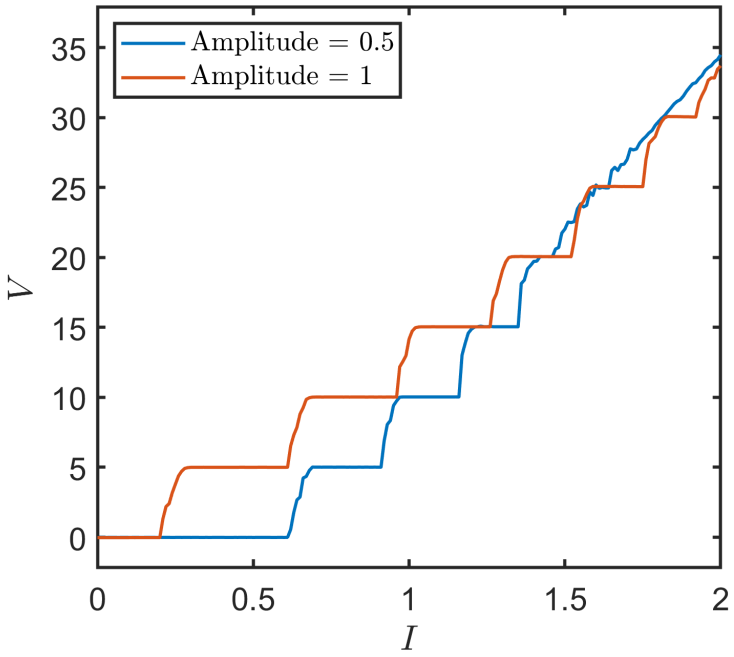

Figure 6.6: Giant shapiro steps in a square array.

Chapter 7

JJAsim_2D_visualize API (v1.1)

The JJAsim_2D_visualize API is used to visualize circuits, stationary states and time evolutions. It consists of the following functions:

- JJAsim_2D_visualize_circuit
- JJAsim_2D_visualize_snapshot
- JJAsim_2D_visualize_movie

Note that all variables are of type double if the type is not specified, or boolean if the variable end in Q.

7.1 JJAsim_2D_visualize_circuit

JJAsim_2D_visualize_circuit(array,varargin)

description

- Displays a circuit representation of an array with the symbols of components.
- Inductors are not shown.
- Nodes where external current is injected are shown with a dot and nodes where external current is ejected are displayed with a cross.

variable input

Variable	size	default	description
showQuantitiesQ	1 by 1	true	If true, Rn, Ic, betaC and IExtBase are visualized.
FontSize	1 by 1	18	Font size
textColor	1 by 3	[0.05,0.5,1]	RGB triplet for text color.
lineWidth	1 by 1	1.5	Width of all lines (in pixels)
nodeDiameter	1 by 1	0.16	Diameter of nodes (circuit nodes)
JLength	1 by 1	0.42	Length of all junctions (as a fraction of its wire)
JLength	1 by 1	0.14	Width of all junctions (as a fraction of its wire)
RWidth	1 by 1	0.08	Resistor is rectangle with this width (fraction of wire)
RLength	1 by 1	0.2	... and this length (fraction of wire)
JEsize	1 by 1	0.12	Width of Josephson element as a fraction of wire
CGap	1 by 1	0.04	Capcitor gap size as a fraction of wire
CWidth	1 by 1	0.14	Capacitor width as a fraction of wire

7.2 JJAsim_2D_visualize_snapshot

```
out = JJAsim_2D_visualize_snapshot(array,n,I,varargin)
```

description

- Displays a snapshot of a 2D josephson junction array with a current configuration I and a vortex configuration n.
- Currents are displayed with arrows where the current is proportional to the strength.
- Vortices are displayed as circles.
- Currents and vortices can be displayed with a 'normal' or 'fancy' draw mode. Normal is more efficient. Set with arrowType and vortexType.
- Optionally one can display node quantities with colored nodes. Can for example be used to display node phases or node potential.
- Optionally one can display path quantities with colored path area. Can for example be used to display path currents or self fields.
- Optionally one can display the nodes in which external current is injected and ejected with showIExtBaseQ. Injected nodes get a dot and ejection nodes get a cross.

fixed input

Variable	size	description
array	struct	information about Josephson junction array.
n	Np by 1	vortex configuration
I	Nj by 1	current configuration

variable input

Variable	size	default	description
figurePosition	empty	[]	automatic figure position
	1 by 1		figure is centered, between 0 and 1 where 1 = fullscreen.
	1 by 4		manual figure position.
FontName	string	'Arial'	font name
FontSize	1 by 1	12	font size
showVorticesQ	1 by 1	true	If true, vortices are displayed in the centre of paths.
vortexDiameter	1 by 1	0.3	Diameter of displayed vortices.
vortexColor	1 by 3	[0,0,0]	RGB triplet for vortex color
antiVortexColor	1 by 3	[1,0,0]	RGB triplet for antivortex color
vortexType	string	'fancy'	'normal' or 'fancy'. Draw mode for vortices.
showGridQ	1 by 1	true	draw a line grid over all junctions
gridWidth	1 by 1	0.05	width of gridlines

(continued)	size	default	description
gridColor	1 by 3	[.8,.8,.8]	RGB triplet for grid color
showCurrentQ	1 by 1	true	If true, current is displayed with arrows whose length is proportional to the magnitude of the current.
arrowWidth	1 by 1	1	width scale factor of current arrays.
arrowLength	1 by 1	1	length scale factor of current arrays.
arrowColor	1 by 3	[0,0,1]	RGB triplet for arrow color
arrowType	string	'fancy'	'normal' or 'fancy'. Draw mode for current arrows.
showNodesQ	1 by 1	true	If true, nodes are shown as circles.
nodeDiameter	1 by 1	0.3	Diameter of displayed nodes.
nodeColor	1 by 3	[1,1,1]	RGB triplet for color of nodes (ignored if showNodeQuantityQ)
showNodeQuantityQ	1 by 1	false	if true, nodeQuantity is displayed with colored nodes.
nodeQuantity	Nn by 1	[]	quantity displayed with colored nodes.
nodeColorLimits	empty 1 by 2	[] 	automatic color limits for nodeQuantity manual color limits for nodeQuantity
nodeQuantityLabel	string	”	Colorbar label for nodeQuantity
showPathQuantityQ	1 by 1	false	if true, pathQuantity is displayed by coloring the enclosed area of each path.
pathQuantity	Np by 1	[]	quantity displayed with colored path areas.
pathColorLimits	empty 1 by 2	[] 	automatic color limits for pathQuantity manual color limits for pathQuantity
pathQuantityLabel	string	”	Colorbar label for pathQuantity
pathQuantityAlpha	1 by 1	1	Transparancy of colored path areas.
showIExtBaseQ	1 by 1	false	Dispay the nodes where external current is in/ejected.
IExtBaseColor	1 by 3	[0,0,0]	RGB color triplet for symbols displaying external current.

output

Variable	size	description
out.figureHandle	handle	figure handle
out.axisHandle	handle	axis handle
out.colorbarHandle	handle	colorbar handle (if a colorbar exists)

7.3 JJAsim_2D_visualize_movie

JJAsim_2D_visualize_movie(array,t,n,I,varargin)

description

- Displays a movie of a time evolution simulation on a 2D josephson junction array.
- Currents are displayed with arrows where the current is proportional to the strength.
- Vortices are displayed as circles.
- Currents and vortices can be displayed with a 'normal' or 'fancy' draw mode. Normal is more efficient. Set with arrowType and vortexType.
- Optionally one can display node quantities with colored nodes. Can for example be used to display node phases or node potential.
- Optionally one can display path quantities with colored path area. Can for example be used to display path currents or self fields.
- Optionally one can display the nodes in which external current is injected and ejected with showIExtBaseQ. Injected nodes get a dot and ejection nodes get a cross.
- The movie can be recorded into an .avi file.

fixed input

Variable	size	description
array	struct	information about Josephson junction array.
t	Nt by 1	time points
n	Np by Nt	vortex configuration
I	Nj by Nt	current configuration

variable input

Variable	size	default	description
figurePosition	empty	[]	automatic figure position
	1 by 1		figure is centered, between 0 and 1 where 1 = fullscreen.
	1 by 4		manual figure position.
selectedTimePoints	Nt by 1	true(Nt,1)	The entries that are true are displayed in the movie.
FontName	string	'Arial'	font name
FontSize	1 by 1	12	font size
showVorticesQ	1 by 1	true	If true, vortices are displayed in the centre of paths.
vortexDiameter	1 by 1	0.3	Diameter of displayed vortices.
vortexColor	1 by 3	[0,0,0]	RGB triplet for vortex color
antiVortexColor	1 by 3	[1,0,0]	RGB triplet for antivortex color
vortexType	string	'fancy'	'normal' or 'fancy'. Draw mode for vortices.

(continued)	size	default	description
showGridQ	1 by 1	true	draw a line grid over all junctions
gridWidth	1 by 1	0.05	width of gridlines
gridColor	1 by 3	[.8,.8,.8]	RGB triplet for grid color
showCurrentQ	1 by 1	true	If true, current is displayed with arrows whose length is proportional to the magnitude of the current.
arrowWidth	1 by 1	1	width scale factor of current arrays.
arrowLength	1 by 1	1	length scale factor of current arrays.
arrowColor	1 by 3	[0,0,1]	RGB triplet for arrow color
arrowType	string	'fancy'	'normal' or 'fancy'. Draw mode for current arrows.
showNodesQ	1 by 1	true	If true, nodes are shown as circles.
nodeDiameter	1 by 1	0.3	Diameter of displayed nodes.
nodeColor	1 by 3	[1,1,1]	RGB triplet for color of nodes (ignored if showNodeQuantityQ)
showNodeQuantityQ	1 by 1	false	if true, nodeQuantity is displayed with colored nodes.
nodeQuantity	Nn by Nt	[]	quantity displayed with colored nodes.
nodeColorLimits	empty 1 by 2	[]	automatic color limits for nodeQuantity manual color limits for nodeQuantity
nodeQuantityLabel	string	”	Colorbar label for nodeQuantity
showPathQuantityQ	1 by 1	false	if true, pathQuantity is displayed by coloring the enclosed area of each path.
pathQuantity	Np by Nt	[]	quantity displayed with colored path areas.
pathColorLimits	empty 1 by 2	[]	automatic color limits for pathQuantity manual color limits for pathQuantity
pathQuantityLabel	string	”	Colorbar label for pathQuantity
pathQuantityAlpha	1 by 1	1	Transparancy of colored path areas.
showIExtBaseQ	1 by 1	false	Dispay the nodes where external current is in/ejected.
IExtBaseColor	1 by 3	[0,0,0]	RGB color triplet for symbols displaying external current.
framePause	1 by 1	0.005	extra time between frames
saveQ	1 by 1	false	true to generate a .avi file
filename	string	'JJAmovie'	filename of generated .avi file
framerate	1 by 1	24	framerate of generated .avi file
compression	1 by 1	0.5	compression factor of generated .avi file. Between 0 and 1. 1 means no compression.

Chapter 8

JJAsim_2D_network API (v1.1)

The JJAsim_2D_network API consists of the following functions:

- array methods
  - JJAsim_2D_network_create
  - JJAsim_2D_network_adjust
  - JJAsim_2D_network_removeNodes
  - JJAsim_2D_network_removeJunctions
  - JJAsim_2D_network_square
  - JJAsim_2D_network_honeycomb
  - JJAsim_2D_network_triangular
- basic methods
  - JJAsim_2D_network_method_changePhaseZone
  - JJAsim_2D_network_method_getn
  - JJAsim_2D_network_method_getFlux
  - JJAsim_2D_network_method_getphi
  - JJAsim_2D_network_method_getU
  - JJAsim_2D_network_method_getJ
  - JJAsim_2D_network_method_getEJ
  - JJAsim_2D_network_method_getEM
  - JJAsim_2D_network_method_getEC
- stationairy state methods
  - JJAsim_2D_network_stationairyState
  - JJAsim_2D_network_stationairyState_approx_london
  - JJAsim_2D_network_stationairyState_approx_arctan
  - JJAsim_2D_network_stationairyState_stability
- time evolution methods
  - JJAsim_2D_network_simulate

Some general remarks:

- Variables are of type double if the type is not specified, or boolean if the variable end in Q.
- An asterisk behind a size implies that it can also be of length 1 in that dimension. In that case, all elements are assumed to have this value.

8.1 array methods

8.1.1 JJAsim_2D_network_create

```
array = JJAsim_2D_network_create(nodePosition,junctionNodes,...
  IExtBase,varargin)
```

description

- Generate an 2D electrical network of Josephson junctions with arbitrary network structure. The resulting struct array is used in the JJAsim package to do simulations.
- Includes geometrical information and physical quantities of the components.
- All physical quantities are dimensionless, see table 4.2 for normalizations.
- Note that for networks which have the structure of a lattice one can use JJAsim_lattice_2D instead, as this special type generally allows for significantly sped up computation.
- Inductances are specified as inductances between junctions (not between paths). Realistic values can be computed with the INDUCTsim package.

fixed input

Variable	size	description
nodePosition	Nn by 2	x and y coordinates of the nodes of the array
junctionNodes	Nj by 2	node numbers of endpoints of each junction
IExtBase	Nn by 1	relative external current injected per node

variable input

Variable	size	default	description
customcpQ	1 by 1	false	true if custom current-phase relation cp
cp	function _handle	@(Ic,th) Ic.*sin(th)	current phase relation @(Ic,th) cp(Ic,th)
dcp	function _handle	@(Ic,th) Ic.*cos(th)	derivative of current phase relation
icp	function _handle	@(Ic,th)Ic .*(1-cos(th))	integral of current phase relation
Ncp	1 by 1	1	number of c-p relation parameters
Ic	Nj* by Ncp	1	critical current of junctions
Rn	Nj* by 1	1	normal state resistance of junctions
betaC	Nj* by 1	0	capacitance of junctions
betaL (self)	Nj* by 1	0	junction self inductance.
(self+mutual)	Nj by Nj		junction inductance matrix. Can be sparse or dense.
cycleAlgorithm	string	'greedy'	Algorithm for cycle space ('greedy', 'spanningTree').

output

Variable	size	description
array	struct	container for all information of the array.

8.1.2 JJAsim_2D_network_adjust

```
array = JJAsim_2D_network_adjust(array,varargin)
```

description

- Adjust physical properties of 2D electrical network of Josephson junctions defined in array.
- Cannot add/remove nodes or junctions.

fixed input

Variable	size	description
array	struct	josephson junction array

variable input

Variable	size	default	description
nodePosition	Nn by 2	[]	x and y coordinates of the nodes of the array
IExtBase	Nn by 1	[]	relative external current injected per node
customcpQ	1 by 1	[]	true if custom current-phase relation cp
cp	function _handle	[]	current phase relation @(Ic,th) cp(Ic,th)
dcp	function _handle	[]	derivative of current phase relation
icp	function _handle	[]	integral of current phase relation
Ncp	1 by 1	[]	number of c-p relation parameters
Ic	Nj* by Ncp	[]	critical current of junctions
Rn	Nj* by 1	[]	normal state resistance of junctions
betaC	Nj* by 1	[]	capacitance of junctions
betaL (self) (self+mutual)	Nj* by 1 Nj by Nj	[]	junction self inductance. junction inductance matrix.
cycleAlgorithm	string	[]	Algorithm to find cycle space. 'greedy' or 'spanningTree'.

output

Variable	size	description
array	struct	adjusted array

8.1.3 JJAsim_2D_network_removeNodes

```
[array,newNodeNrs,newJunctionNrs] = JJAsim_2D_network_removeNodes(...
    array,nodeNrs)
```

description

- remove nodes from an existing 2D network of Josephson junctions.

fixed input

Variable	size	description
array	struct	josephson junction array
nodeNrs	N by 1	list of node numbers to remove

output

Variable	size	description
array	struct	adjusted array
newNodeNrs	Nn by 1	Lists for all original nodes what new number is assigned to it. 0 if removed.
newJunctionNrs	Nj by 1	Lists for all original junctions what new number is assigned to it. 0 if removed.

8.1.4 JJAsim_2D_network_removeJunctions

```
[array,newJunctionNrs] = JJAsim_2D_network_removeJunctions(array,junctionNrs)
```

description

- remove junctions from an existing 2D network of Josephson junctions.

fixed input

Variable	size	description
array	struct	josephson junction array
junctionNrs	N by 1	list of junction numbers to remove

output

Variable	size	description
array	struct	adjusted array
newJunctionNrs	Nj by 1	Lists for all original junctions what new number is assigned to it. 0 if removed.

8.1.5 JJAsim_2D_network_square

```
array = JJAsim_2D_network_square(Nx,Ny,ax,ay,...
  currentDirection,varargin)
```

description

- generates a Nx by Ny square array.
- Inherits optional input from JJAsim_2D_network_create.

fixed input

Variable	size	description
Nx	1 by 1	number of nodes in x-direction
Ny	1 by 1	number of nodes in y-direction
ax	1 by 1	node distance in x-direction
ay	1 by 1	node distance in y-direction
currentDirection	string	'x' or 'y'. Sets a homogeneous external current in the respective dimension

variable input

inherited from JJAsim_2D_network_create

output

Variable	size	description
array	struct	container for all information of the array.

8.1.6 JJAsim_2D_network_honeycomb

```
array = JJAsim_2D_network_honeycomb(Nx,Ny,ax,ay,...
  currentDirection,varargin)
```

description

- generates a Nx by Ny unit cells honeycomb array. A unit cell contains 4 nodes.
- Inherits optional input from JJAsim_2D_network_create.

fixed input

Variable	size	description
Nx	1 by 1	number of nodes in x-direction
Ny	1 by 1	number of nodes in y-direction
ax	1 by 1	node distance in x-direction
ay	1 by 1	node distance in y-direction
currentDirection	string	'x' or 'y'. Sets a homogeneous external current in the respective dimension

variable input

inherited from JJAsim_2D_network_create

output

Variable	size	description
array	struct	container for all information of the array.

8.1.7 JJAsim_2D_network_triangular

```
array = JJAsim_2D_network_triangular(Nx,Ny,ax,ay,...
  currentDirection,varargin)
```

description

- generates a Nx by Ny unit cells triangular array. A unit cell contains 2 nodes.
- Inherits optional input from JJAsim_2D_network_create.

fixed input

Variable	size	description
Nx	1 by 1	number of nodes in x-direction
Ny	1 by 1	number of nodes in y-direction
ax	1 by 1	node distance in x-direction
ay	1 by 1	node distance in y-direction
currentDirection	string	'x' or 'y'. Sets a homogeneous external current in the respective dimension

variable input

inherited from JJAsim_2D_network_create

output

Variable	size	description
array	struct	container for all information of the array.

8.2 basic methods

8.2.1 JJAsim_2D_network_method_getn

```
n = JJAsim_2D_network_method_getn(array,th,z)
```

description

- obtains the vortex configuration n from phase difference th and phase zone z.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
th	Nj* by W* by Nt	gauge invariant phase difference
z	Np* by W*	phase zone

output

Variable	size	description
n	Np by W by Nt	vortex configuration

8.2.2 JJAsim_2D_network_method_changePhaseZone

```
th_new = JJAsim_2D_network_method_changePhaseZone(array,th_old,z_old,z_new)
```

description

- Expresses th in a different phase zone.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
th_old	Nj* by W* by Nt	gauge invariant phase difference
z_old	Np* by W*	phase zone of th_old
z_new	Np* by W*	new phase zone to express th_old in

output

Variable	size	description
th_new	Nj by W by Nt	gauge invariant phase difference in new phase zone

8.2.3 JJAsim_2D_network_method_getFlux

```
Phi = JJAsim_2D_network_method_getFlux(array,I)
```

description

- obtains the induced magnetic flux Phi from junction current I.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
I	Nj* by W by Nt	junction current

output

Variable	size	description
Phi	Np by W by Nt	induced magnetic flux

8.2.4 JJAsim_2D_network_method_getphi

```
phi = JJAsim_2D_network_method_getphi(array,th)
```

description

- obtains the gauge dependent phase at each node from the gauge invariant phase difference th.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
th	Nj* by W by Nt	gauge invariant phase difference

output

Variable	size	description
phi	Np by W by Nt	gauge dependent phase at each node

8.2.5 JJAsim_2D_network_method_getU

```
U = JJAsim_2D_network_method_getU(array,V)
```

description

- obtains the node potential U from the junction voltage drop V.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
V	Nj* by W by Nt	junction voltage drop

output

Variable	size	description
U	Np by W by Nt	node potential

8.2.6 JJAsim_2D_network_method_getJ

```
J = JJAsim_2D_network_method_getJ(array,I)
```

description

- obtains the path current J from the junction current I.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
I	Nj* by W by Nt	junction current

output

Variable	size	description
J	Np by W by Nt	path current

8.2.7 JJAsim_2D_network_method_getEJ

```
[EJ,EJtot] = JJAsim_2D_network_method_getEJ(array,th)
```

description

- obtains energy stored in the Josephson elements, called the Josephson Energy EJ, from the gauge invariant phase difference th.
- EJ lists it for all junctions individually, whereas EJtot is the sum over all junctions.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
th	Nj by W by Nt	gauge invariant phase difference

output

Variable	size	description
EJ	Nj by W by Nt	Josephson energy
EJtot	W by Nt	total Josephson energy of whole array

8.2.8 JJAsim_2D_network_method_getEM

```
[EM,EMtot] = JJAsim_2D_network_method_getEM(array,I)
```

description

- obtains the energy stored in the magnetic self fields, called EM, from the junction current I.
- EM lists it for all junctions individually, whereas EMtot is the sum over all junctions.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
I	Nj by W by Nt	junction current

output

Variable	size	description
EM	Nj by W by Nt	magnetic energy
EMtot	W by Nt	total magnetic energy of whole array

8.2.9 JJAsim_2D_network_method_getEC

```
[EC,ECtot] = JJAsim_2D_network_method_getEC(array,V)
```

description

- obtains the energy stored in the capacitors, called EC, from the junction voltage V.
- EC lists it for all junctions individually, whereas ECtot is the sum over all junctions.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
V	Nj by W by Nt	junction voltage

output

Variable	size	description
EC	Nj by W by Nt	capacitance energy
ECtot	W by Nt	total capacitance energy of whole array

8.3 stationairy state methods

8.3.1 JJAsim_2D_network_stationairyState

```
out = JJAsim_2D_network_stationairyState(array,inputMode,IExt,f,...
  z,th1,varargin)
```

description

- Finds stationairy states for 2D josephson junction networks. The array can be an arbitrary 2D network. This is specified by array, which is created with JJAsim_2D_network_create.
- Requires an initial guess th1. A suitible initial guess can be generated with the function call JJAsim_2D_network_stationairyState_approx_london.
- A solution need not exist, or if it exits, it may not be found. solutionQ lists if a solution is found.
- To find a solution, Newtons iteration is used. The number of iterations is stored in nrOfSteps. The maximum number of steps and acceptance tolerance can be specified with newton_steps and newton_tol respectively.
- Several problems can be computed in parallel in one function call. inputMode determines the way to specify these; the options are 'sweep' and 'list'.
- All physical quantities are normalized, see table 4.2.

fixed input

Variable	size	description
array	struct	Geometric information of the network. Create with JJAsim_2D_network_create.
inputMode	string	'sweep' or 'list'. Determines how to specify multiple simultaneous problems. Effects IExt,f,z,n. For 'list' the number of problems is W, whereas for 'sweep', the number of problems W = WI*Wf.
IExt (sweep) (list)	WI by 1 W* by 1	External current
f (sweep) (list)	Np* by Wf* Np* by W*	Frustration factor
z (sweep) (list)	Np* by Wf* Np* by W*	Phase zone
th1	Nj* by W*	Initial guess for gauge invariant phase differences.

variable input

Variable	size	default	description
repetitions	1 by 1	1	Number of repetitions for the whole problem set
newton_steps	1 by 1	20	number of steps for newtons algorithm.
newton_tolerance	1 by 1	1E-7	acceptance tolerance for newtons algorithm.
storethQ	1 by 1	true	Determines if th is stored in output.
storenQ	1 by 1	true	Determines if n is stored in output.
storeIQ	1 by 1	true	Determines if I is stored in output.
storeEQ	1 by 1	true	Determines if E is stored in output.
computePartitions	1 by 1	1	Splits the problem set into computePartitions which are computed consecutively. Use to reduce memory usage or for parallel computation.
parallelQ	1 by 1	false	If true, the partitions are computed in parallel on multiple cores.
cores	1 by 1	0	Number of cores used by parallel computing. If 0, it is set to the amount of available cores.

output

Variable	size	description
out.solutionQ	W by 1	true if valid solution is found
out.nrOfSteps	W by 1	number of newton iterations done to find solution
out.th	Nj by W	gauge invariant phase difference of every junction
out.n	Np by W	vortex configuration
out.I	Nj by W	junction current
out.E	W by 1	total energy stored in stationairy state

8.3.2 JJAsim_2D_network_stationairyState_approx_london

```
[th,I] = JJAsim_2D_network_stationairyState_approx_london(array,n,f)
```

description

- Generate an approximate stationairy state for a 2D network with the within the London's gauge, which can be used as an initial guess for JJAsim_2D_network_stationairyState or initial condition for JJAsim_2D_network_simulate.
- The output is found the phase zone z = n. On can use JJAsim_2D_network_method_changePhaseZone to convert it to other phase zones.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
n	Np* by W	desired vortex configuration
f	Np* by W	frustration factor

output

Variable	size	description
th	Nj by W	approximate gauge invariant phase difference
I	Nj by W	approximate junction current

8.3.3 JJAsim_2D_network_stationairyState_approx_arctan

```
[th,I,n] = JJAsim_2D_network_stationairyState_approx_london(array,...
  x0,y0,n0,f)
```

description

- Generate an approximate stationairy state for a 2D network with the arctan approximation, which can be used as an initial guess for JJAsim_2D_network_stationairyState or initial condition for JJAsim_2D_network_simulate.
- The resulting th will encode the specified vortex configuration.
- A vortex configuration is specified with a list of coordinate and vorticity pairs, (x0,y0,n0).
- The output is in the phase zone z = 0. One can use the function JJAsim_2D_network_method_changePha to convert it to other phase zones.
- Also returns nInput, which is the vector representation of the input vortex configuration. This need not be equal to n, the vortex configuration of the output th. If they differ, a warning is given.

fixed input

Variable	size	description
array	struct	information about Josephson junction array
x0	L by W*	x coordinates of the L sites that contain vortices
y0	L by W*	y coordinates of the L sites that contain vortices
n0	L by W*	vorticity of the L sites that contain vortices. Must be integers.
f	Np* by W*	frustration factor

output

Variable	size	description
th	Nj by W	approximate gauge invariant phase difference
I	Nj by W	approximate junction current
n	Np by W	vortex configuration of approximate initial condition
nInput	Np by W	vector representation of input vortex configuration

8.3.4 JJAsim_2D_network_stationairyState_stability

```
[stableQ,eigv] = JJAsim_2D_network_stationairyState_stability(array,th)
```

description

- returns if a stationairy state th is dynamically stable, defined by the criterion that the maximum eigenvalue of the Jacobian of the system is non-positive.
- Can compute W problems in one function call.
- Also outputs the eigenvalues of the Jacobian.
- Does not explicitly check if th is a valid stationairy state.

fixed input

Variable	size	description
array	struct	array defining a network of Josephson junctions.
th	Nj by W	stationairy state to check the stability of.

output

Variable	size	description
stableQ	1 by W	true if stationairy state is stable
eigv	Nj by W	listing all eigenvalues of the Jacobian matrix at th

8.4 time evolution methods

8.4.1 JJAsim_2D_network_simulate

```
out = JJAsim_2D_network_simulate(array,t,inputMode,IExt,T,f,z,th1,varargin)
```

description

- Compute time evolution on 2D josephson junction networks. The array can be an arbitrary 2D network. This is specified by array, which is created with JJAsim_2D_network_create.
- Requires an initial condition th1. A reasonable initial condition can be generated with the JJAsim_2D_network_stationairyState_approx functions.
- several problems can be computed in one function call.
- if the array contains capacitance (i.e. array.capacitanceQ is true), the second timestep must also be specified with th2.
- All physical quantities are dimensionless, see documentation.

fixed input

Variable	size	description
array	struct	Geometric information of the network, Create with JJAsim_2D_network_create.
t	1 by Nt	simulated time points.
inputMode	string	'sweep' or 'list'. Determines how to specify multiply simultaneous problems. Effects IExt, f and T. For 'list', the number of parallel problems is W. For 'sweep', W = WI*Wf*WT.
IExt (sweep) (list)	WI by Nt* W* by Nt*	External current
T (sweep) (list)	WT by Nt* W* by Nt*	temperature parameter
f (sweep) (list)	Np* by Wf* Np* by W*	frustration factor
z (sweep) (list)	Np* by Wf* Np* by W*	phase zone
th1	Nj* by W*	Initial condition phase differences

variable input

Variable	size	default	description
th2	Nj* by W*	[]	phase differences at second timestep. Only if array.capacitanceQ. Equal to th1 if empty.
Vscheme	string	'forward'	Scheme of the voltage derivative dth/dt. Can be set to 'forward', 'central' or 'backward'. Only used if array.capacitanceQ = true.
storethQ	1 by 1	true	If true, phase difference th is stored in output.
storeIQ	1 by 1	true	If true, phase difference I is stored in output.
storeVQ	1 by 1	true	If true, phase difference V is stored in output.
storeVtotQ	1 by 1	true	If true, phase difference Vtot is stored in output.
storeEQ	1 by 1	true	If true, phase difference E is stored in output.
thTimePoints	Nt by 1	true(Nt,1)	Lists which for which timepoints th is stored.
ITimePoints	Nt by 1	true(Nt,1)	Lists which for which timepoints I is stored.
VTimePoints	Nt by 1	true(Nt,1)	Lists which for which timepoints V is stored.
VtotTimePoints	Nt by 1	true(Nt,1)	Lists which for which timepoints Vtot is stored.
ETimePoints	Nt by 1	true(Nt,1)	Lists which for which timepoints E is stored.
repetitions	1 by 1	1	Number of repetition of the whole problem set. rep for short.
computePartitions	1 by 1	1	Splits the problems into computePartitions which are computed separately. Use to reduce memory or for parallel computation.
parallelQ	1 by 1	false	If true, the partitions are computed in parallel on multiple cores.
cores	1 by 1	0	Number of cores used by parallel computing. If 0, it is set to the amount of available cores.

output

Variable	size	description
out.th	Nj by W*rep by Ntth	gauge invariant phase difference output
out.I	Nj by W*rep by NtI	junction current output
out.V	Nj by W*rep by NtV	junction voltage drop output
out.Vtot	W*rep by NtVtot	total voltage drop output
out.E	W*rep by NtE	total energy output