Lab Session #5

Computational Neurophysiology [E010620A]

Dept of Electronics and Informatics (VUB) and Dept of Information Technology (UGent)

Jorne Laton, Matthias Inghels, Talis Vertriest, Jeroen Van Schependom, Sarah Verhulst

Student names and IDs: Constantijn Coppers (02010771)

Academic Year: 2023-2024

Module 3: Brunel network

In this practical, you will simulate a network of sparsely connected identical Leaky-Integrate-and-Fire neurons.

This model is based on the paper by Brunel 2000 and is also discussed in Neuronal Dynamics, see eg. Figure 13.7.

We will show that a large network of neurons consisting of a group of excitatory and a group of inhibitory neurons that receive external input is capable of producing a rich dynamics. The parameters that we will vary to trigger different states are (1) the driving frequency of the external population and (2) the relative strength of inhibition vs excitation (g).

To do so, we will follow two strategies both relying on the BRIAN package: the first strategy is to create our own network from scratch. This will allow you to get acquainted with the BRIAN package and will prepare you for the second strategy. Here, you will use a pre-defined spiking network (included in the BRIAN package), but you will have to make sure you understand its input and outputs in order to induce the correct behaviour.

We start by ensuring all packages are loaded. Make sure to have installed the Brian Package

In [1]:
```
import brian2 as b2
from brian2 import NeuronGroup, Synapses, PoissonInput
from brian2.monitors import StateMonitor, SpikeMonitor, PopulationRateMonitor
from random import sample
from neurodynex3.tools import plot_tools, spike_tools
from numpy import random
import matplotlib.pyplot as plt
import numpy as np
```

Step 1.1. Initialising the NeuronGroup instance

We start by defining a NeuronGroup instance. You can find the documentation here

A NeuronGroup instance takes as input the number of neurons, the dynamics of a single neuron 'model', the length of the refractory period and the integration method. Note that initialising the NeuronGroup will not yet run the simulation. Running the simulation will be done in the upcoming blocks once we have added all details to the network.

The input parameters of the NeuronGroup instance should be:

- number of neurons: N_Excit (4000) +N_Inhib (1000)
- the model should model the LIF dynamics
- the model should reset at a threshold v = v_reset (10 mV)
- the neurons should have an absolute refractory period (2ms)
- integration method should be linear

What is the default firing threshold?

In [2]:
```
# Import neurodynex3 LIF dynamics & check default values
from neurodynex3.brunel_model import LIF_spiking_network

b2.start_scope()

# Define the network parameters
N_Excit = 4000
N_Inhib = 1000

# Define the single neuron model parameters
v_rest = 0*b2.mV
v_reset = 10*b2.mV
abs_refractory_period = 2*b2.ms

firing_threshold = LIF_spiking_network.FIRING_THRESHOLD
membrane_time_scale = LIF_spiking_network.MEMBRANE_TIME_SCALE
# b2.defaultclock.dt = 0.05*b2.ms

# ... and its dynamics
lif_dynamics = """
dv/dt = (v_rest - v)/membrane_time_scale : volt
"""
```

In [3]:
```
# Print parameters
print('Default firing threshold:\t', firing_threshold)
print('Default membrane timescale:\t', membrane_time_scale)
# print('Default defaultclok:\t\t', b2.defaultclock.dt)
```

```
Default firing threshold:        20. mV
Default membrane timescale:      20. ms
```

In [4]:
```
# Define the network
network = NeuronGroup(
    N_Excit + N_Inhib, model = lif_dynamics,
    threshold = 'v > firing_threshold', refractory = abs_refractory_period,
    method = 'linear', reset = 'v = v_reset')

print(network)
print(network.v)
```

```
NeuronGroup(clock=Clock(dt=50. * usecond, name='defaultclock'), when=start, order=0,
name='neurongroup')
<neurongroup.v: array([0., 0., 0., ..., 0., 0., 0.]) * volt>
```

A1.1 Answer

The default firing threshold is 20 mV.

Step 1.2. Add network structure

Now that we have created a group of neurons, we will define how they are connected. In order to connect neurons to each other, we first need to distinguish the two neuronal populations (the excitatory and inhibitory population). In order to define the first N_Excit neurons to be excitatory and the remaining part to be inhibitory, you can simply use the following code snippet.

In [5]:
```
# Split up the network into excitatory and inhibitory populations
excitatory_population = network[:N_Excit]
inhibitory_population = network[N_Excit:]
```

Now, we need to define two types of synapses: excitatory and inhibitory synapses. Allow for a synaptic delay of 1.5*b2.ms and use a random connection probability of 0.1. You can find the documentation here In order to follow the notations in the book, please use following notation and default values:

What is the "target" network?

In [6]:
```
# Connection Parameters
w0 = 0.1 * b2.mV
g = 4.0
J_excit = w0
J_inhib = - g * w0
synaptic_delay = 1.5 * b2.ms
connection_probability = 0.1
```

In [8]:
```
# Excitatory Synapses
exc_synapses = Synapses(excitatory_population, target = network, on_pre = "v += J_
exc_synapses.connect(p = connection_probability)
```

```
# Inhibitory Synapses
inhib_synapses = Synapses(inhibitory_population, target = network, on_pre = "v +=
inhib_synapses.connect(p = connection_probability)
```

In [9]:
```
help(Synapses)
```

```
Help on class Synapses in module brian2.synapses.synapses:

class Synapses(brian2.groups.group.Group)
 |  Synapses(source, target=None, model=None, on_pre=None, pre=None, on_post=None,
post=None, connect=None, delay=None, on_event='spike', multisynaptic_index=None, n
amespace=None, dtype=None, codeobj_class=None, dt=None, clock=None, order=0, metho
d=('exact', 'euler', 'heun'), method_options=None, name='synapses*')
 |
 |  Class representing synaptic connections.
 |
 |  Creating a new `Synapses` object does by default not create any synapses,
 |  you have to call the `Synapses.connect` method for that.
 |
 |  Parameters
 |  ----------
 |
 |  source : `SpikeSource`
 |      The source of spikes, e.g. a `NeuronGroup`.
 |  target : `Group`, optional
 |      The target of the spikes, typically a `NeuronGroup`. If none is given,
 |      the same as `source`
 |  model : `str`, `Equations`, optional
 |      The model equations for the synapses.
 |  on_pre : str, dict, optional
 |      The code that will be executed after every pre-synaptic spike. Can be
 |      either a single (possibly multi-line) string, or a dictionary mapping
 |      pathway names to code strings. In the first case, the pathway will be
 |      called ``pre`` and made available as an attribute of the same name.
 |      In the latter case, the given names will be used as the
 |      pathway/attribute names. Each pathway has its own code and its own
 |      delays.
 |  pre : str, dict, optional
 |      Deprecated. Use ``on_pre`` instead.
 |  on_post : str, dict, optional
 |      The code that will be executed after every post-synaptic spike. Same
 |      conventions as for `on_pre``, the default name for the pathway is
 |      ``post``.
 |  post : str, dict, optional
 |      Deprecated. Use ``on_post`` instead.
 |  delay : `Quantity`, dict, optional
 |      The delay for the "pre" pathway (same for all synapses) or a dictionary
 |      mapping pathway names to delays. If a delay is specified in this way
 |      for a pathway, it is stored as a single scalar value. It can still
 |      be changed afterwards, but only to a single scalar value. If you want
 |      to have delays that vary across synapses, do not use the keyword
 |      argument, but instead set the delays via the attribute of the pathway,
 |      e.g. ``S.pre.delay = ...`` (or ``S.delay = ...`` as an abbreviation),
 |      ``S.post.delay = ...``, etc.
 |  on_event : str or dict, optional
 |      Define the events which trigger the pre and post pathways. By default,
 |      both pathways are triggered by the ``'spike'`` event, i.e. the event
 |      that is triggered by the ``threshold`` condition in the connected
 |      groups.
 |  multisynaptic_index : str, optional
 |      The name of a variable (which will be automatically created) that stores
 |      the "synapse number". This number enumerates all synapses between the
```

```
 |      same source and target so that they can be distinguished. For models
 |      where each source-target pair has only a single connection, this number
 |      only wastes memory (it would always default to 0), it is therefore not
 |      stored by default. Defaults to ``None`` (no variable).
 |  namespace : dict, optional
 |      A dictionary mapping identifier names to objects. If not given, the
 |      namespace will be filled in at the time of the call of `Network.run`,
 |      with either the values from the ``namespace`` argument of the
 |      `Network.run` method or from the local context, if no such argument is
 |      given.
 |  dtype : `dtype`, dict, optional
 |      The `numpy.dtype` that will be used to store the values, or a
 |      dictionary specifying the type for variable names. If a value is not
 |      provided for a variable (or no value is provided at all), the preference
 |      setting `core.default_float_dtype` is used.
 |  codeobj_class : class, optional
 |      The `CodeObject` class to use to run code.
 |  dt : `Quantity`, optional
 |      The time step to be used for the update of the state variables.
 |      Cannot be combined with the `clock` argument.
 |  clock : `Clock`, optional
 |      The update clock to be used. If neither a clock, nor the `dt` argument
 |      is specified, the `defaultclock` will be used.
 |  order : int, optional
 |      The priority of of this group for operations occurring at the same time
 |      step and in the same scheduling slot. Defaults to 0.
 |  method : str, `StateUpdateMethod`, optional
 |      The numerical integration method to use. If none is given, an
 |      appropriate one is automatically determined.
 |  name : str, optional
 |      The name for this object. If none is given, a unique name of the form
 |      ``synapses``, ``synapses_1``, etc. will be automatically chosen.
 |
 |  Method resolution order:
 |      Synapses
 |      brian2.groups.group.Group
 |      brian2.groups.group.VariableOwner
 |      brian2.core.base.BrianObject
 |      brian2.core.names.Nameable
 |      brian2.core.tracking.Trackable
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __getitem__(self, item)
 |
 |  __init__(self, source, target=None, model=None, on_pre=None, pre=None, on_post
=None, post=None, connect=None, delay=None, on_event='spike', multisynaptic_index=
None, namespace=None, dtype=None, codeobj_class=None, dt=None, clock=None, order=
0, method=('exact', 'euler', 'heun'), method_options=None, name='synapses*')
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  before_run(self, run_namespace)
 |      Optional method to prepare the object before a run.
 |
 |      Called by `Network.after_run` before the main simulation loop starts.
```

```
|
|  check_variable_write(self, variable)
|      Checks that `Synapses.connect` has been called before setting a
|      synaptic variable.
|
|      Parameters
|      ----------
|      variable : `Variable`
|          The variable that the user attempts to set.
|
|      Raises
|      ------
|      TypeError
|          If `Synapses.connect` has not been called yet.
|
|  connect(self, condition=None, i=None, j=None, p=1.0, n=1, skip_if_invalid=Fals
e, namespace=None, level=0)
|      Add synapses.
|
|      See :doc:`/user/synapses` for details.
|
|      Parameters
|      ----------
|      condition : str, bool, optional
|          A boolean or string expression that evaluates to a boolean.
|          The expression can depend on indices ``i`` and ``j`` and on
|          pre- and post-synaptic variables. Can be combined with
|          arguments ``n``, and ``p`` but not ``i`` or ``j``.
|      i : int, ndarray of int, str, optional
|          The presynaptic neuron indices  It can be an index or array of
|          indices if combined with the ``j`` argument, or it can be a string
|          generator expression.
|      j : int, ndarray of int, str, optional
|          The postsynaptic neuron indices. It can be an index or array of
|          indices if combined with the ``i`` argument, or it can be a string
|          generator expression.
|      p : float, str, optional
|          The probability to create ``n`` synapses wherever the ``condition``
|          evaluates to true. Cannot be used with generator syntax for ``j``.
|      n : int, str, optional
|          The number of synapses to create per pre/post connection pair.
|          Defaults to 1.
|      skip_if_invalid : bool, optional
|          If set to True, rather than raising an error if you try to
|          create an invalid/out of range pair (i, j) it will just
|          quietly skip those synapses.
|      namespace : dict-like, optional
|          A namespace that will be used in addition to the group-specific
|          namespaces (if defined). If not specified, the locals
|          and globals around the run function will be used.
|      level : int, optional
|          How deep to go up the stack frame to look for the locals/global
|          (see ``namespace`` argument).
|
|      Examples
|      --------
```

```
 |      >>> from brian2 import *
 |      >>> import numpy as np
 |      >>> G = NeuronGroup(10, 'dv/dt = -v / tau : 1', threshold='v>1', reset='v=
0')
 |      >>> S = Synapses(G, G, 'w:1', on_pre='v+=w')
 |      >>> S.connect(condition='i != j') # all-to-all but no self-connections
 |      >>> S.connect(i=0, j=0) # connect neuron 0 to itself
 |      >>> S.connect(i=np.array([1, 2]), j=np.array([2, 1])) # connect 1->2 and 2
->1
 |      >>> S.connect() # connect all-to-all
 |      >>> S.connect(condition='i != j', p=0.1)  # Connect neurons with 10% proba
bility, exclude self-connections
 |      >>> S.connect(j='i', n=2)  # Connect all neurons to themselves with 2 syna
pses
 |      >>> S.connect(j='k for k in range(i+1)') # Connect neuron i to all j with
0<=j<=i
 |      >>> S.connect(j='i+(-1)**k for k in range(2) if i>0 and i<N_pre-1') # conn
ect neuron i to its neighbours if it has both neighbours
 |      >>> S.connect(j='k for k in sample(N_post, p=i*1.0/(N_pre-1))') # neuron i
connects to j with probability i/(N-1)
 |      >>> S.connect(j='k for k in sample(N_post, size=i//2)') # Each neuron conn
ects to i//2 other neurons (chosen randomly)
 |
 |  register_variable(self, variable)
 |      Register a `DynamicArray` to be automatically resized when the size of
 |      the indices change. Called automatically when a `SynapticArrayVariable`
 |      specifier is created.
 |
 |  unregister_variable(self, variable)
 |      Unregister a `DynamicArray` from the automatic resizing mechanism.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  verify_dependencies(eq, eq_type, deps, should_not_depend_on, should_not_depend
_on_name)
 |      Helper function to verify that event-driven equations do not depend
 |      on clock-driven equations and the other way round.
 |
 |      Parameters
 |      ----------
 |      eq : `SingleEquation`
 |          The equation to verify
 |      eq_type : str
 |          The type of the equation (for the error message)
 |      deps : list
 |          A list of dependencies
 |      should_not_depend_on : list
 |          A list of equations to verify against the dependencies
 |      should_not_depend_on_name : str
 |          The name of the list of equations (for the error message)
 |
 |      Raises
 |      ------
 |      `EquationError`
 |          If the given equation depends on something in the other set of
```

```
 |          equations.
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties defined here:
 |
 |  N_incoming_post
 |      The number of incoming synapses for each neuron in the post-synaptic grou
p.
 |
 |  N_outgoing_pre
 |      The number of outgoing synapses for each neuron in the pre-synaptic group.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  delay
 |      The presynaptic delay (if a pre-synaptic pathway exists).
 |
 |  delay_
 |      The presynaptic delay without unit information (if apre-synaptic pathway e
xists).
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  add_to_magic_network = True
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from brian2.groups.group.Group:
 |
 |  custom_operation(self, *args, **kwds)
 |
 |  resolve_all(self, identifiers, run_namespace, user_identifiers=None, additiona
l_variables=None)
 |      Resolve a list of identifiers. Calls `Group._resolve` for each
 |      identifier.
 |
 |      Parameters
 |      ----------
 |      identifiers : iterable of str
 |          The names to look up.
 |      run_namespace : dict-like, optional
 |          An additional namespace that is used for variable lookup (if not
 |          defined, the implicit namespace of local variables is used).
 |      user_identifiers : iterable of str, optional
 |          The names in ``identifiers`` that were provided by the user (i.e.
 |          are part of user-specified equations, abstract code, etc.). Will
 |          be used to determine when to issue namespace conflict warnings. If
 |          not specified, will be assumed to be identical to ``identifiers``.
 |      additional_variables : dict-like, optional
 |          An additional mapping of names to `Variable` objects that will be
 |          checked before `Group.variables`.
 |
 |      Returns
 |      -------
 |      variables : dict of `Variable` or `Function`
```

```
 |          A mapping from name to `Variable`/`Function` object for each of the
 |          names given in `identifiers`
 |
 |      Raises
 |      ------
 |      KeyError
 |          If one of the names in `identifier` cannot be resolved
 |
 |  run_regularly(self, code, dt=None, clock=None, when='start', order=0, name=Non
e, codeobj_class=None)
 |      Run abstract code in the group's namespace. The created `CodeRunner`
 |      object will be automatically added to the group, it therefore does not
 |      need to be added to the network manually. However, a reference to the
 |      object will be returned, which can be used to later remove it from the
 |      group or to set it to inactive.
 |
 |      Parameters
 |      ----------
 |      code : str
 |          The abstract code to run.
 |      dt : `Quantity`, optional
 |          The time step to use for this custom operation. Cannot be combined
 |          with the `clock` argument.
 |      clock : `Clock`, optional
 |          The update clock to use for this operation. If neither a clock nor
 |          the `dt` argument is specified, defaults to the clock of the group.
 |      when : str, optional
 |          When to run within a time step, defaults to the ``'start'`` slot.
 |          See :ref:`scheduling` for possible values.
 |      name : str, optional
 |          A unique name, if non is given the name of the group appended with
 |          'run_regularly', 'run_regularly_1', etc. will be used. If a
 |          name is given explicitly, it will be used as given (i.e. the group
 |          name will not be prepended automatically).
 |      codeobj_class : class, optional
 |          The `CodeObject` class to run code with. If not specified, defaults
 |          to the `group`'s ``codeobj_class`` attribute.
 |
 |      Returns
 |      -------
 |      obj : `CodeRunner`
 |          A reference to the object that will be run.
 |
 |  runner(self, *args, **kwds)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from brian2.groups.group.VariableOwner:
 |
 |  __getattr__(self, name)
 |
 |  __len__(self)
 |
 |  __setattr__(self, name, val, level=0)
 |      Implement setattr(self, name, value).
 |
 |  add_attribute(self, name)
```

```
 |      Add a new attribute to this group. Using this method instead of simply
 |      assigning to the new attribute name is necessary because Brian will
 |      raise an error in that case, to avoid bugs passing unnoticed
 |      (misspelled state variable name, un-declared state variable, ...).
 |
 |      Parameters
 |      ----------
 |      name : str
 |          The name of the new attribute
 |
 |      Raises
 |      ------
 |      AttributeError
 |          If the name already exists as an attribute or a state variable.
 |
 |  get_states(self, vars=None, units=True, format='dict', subexpressions=False, r
ead_only_variables=True, level=0)
 |      Return a copy of the current state variable values. The returned arrays
 |      are copies of the actual arrays that store the state variable values,
 |      therefore changing the values in the returned dictionary will not affect
 |      the state variables.
 |
 |      Parameters
 |      ----------
 |      vars : list of str, optional
 |          The names of the variables to extract. If not specified, extract
 |          all state variables (except for internal variables, i.e. names that
 |          start with ``'_'``). If the ``subexpressions`` argument is ``True``,
 |          the current values of all subexpressions are returned as well.
 |      units : bool, optional
 |          Whether to include the physical units in the return value. Defaults
 |          to ``True``.
 |      format : str, optional
 |          The output format. Defaults to ``'dict'``.
 |      subexpressions: bool, optional
 |          Whether to return subexpressions when no list of variable names
 |          is given. Defaults to ``False``. This argument is ignored if an
 |          explicit list of variable names is given in ``vars``.
 |      read_only_variables : bool, optional
 |          Whether to return read-only variables (e.g. the number of neurons,
 |          the time, etc.). Setting it to ``False`` will assure that the
 |          returned state can later be used with `set_states`. Defaults to
 |          ``True``.
 |      level : int, optional
 |          How much higher to go up the stack to resolve external variables.
 |          Only relevant if extracting subexpressions that refer to external
 |          variables.
 |
 |      Returns
 |      -------
 |      values : dict or specified format
 |          The variables specified in ``vars``, in the specified ``format``.
 |
 |  set_states(self, values, units=True, format='dict', level=0)
 |      Set the state variables.
 |
```

```
|      Parameters
|      ----------
|      values : depends on ``format``
|          The values according to ``format``.
|      units : bool, optional
|          Whether the ``values`` include physical units. Defaults to ``True``.
|      format : str, optional
|          The format of ``values``. Defaults to ``'dict'``
|      level : int, optional
|          How much higher to go up the stack to resolve external variables.
|          Only relevant when using string expressions to set values.
|
|  state(self, name, use_units=True, level=0)
|      Return the state variable in a way that properly supports indexing in
|      the context of this group
|
|      Parameters
|      ----------
|      name : str
|          The name of the state variable
|      use_units : bool, optional
|          Whether to use the state variable's unit.
|      level : int, optional
|          How much farther to go down in the stack to find the namespace.
|      Returns
|      -------
|      var : `VariableView` or scalar value
|          The state variable's value that can be indexed (for non-scalar
|          values).
|
|  ----------------------------------------------------------------------
|  Methods inherited from brian2.core.base.BrianObject:
|
|  __repr__(self)
|      Return repr(self).
|
|  add_dependency(self, obj)
|      Add an object to the list of dependencies. Takes care of handling
|      subgroups correctly (i.e., adds its parent object).
|
|      Parameters
|      ----------
|      obj : `BrianObject`
|          The object that this object depends on.
|
|  after_run(self)
|      Optional method to do work after a run is finished.
|
|      Called by `Network.after_run` after the main simulation loop terminated.
|
|  run(self)
|
|  ----------------------------------------------------------------------
|  Readonly properties inherited from brian2.core.base.BrianObject:
|
|  clock
```

```
|      The `Clock` determining when the object should be updated.
|
|      Note that this cannot be changed after the object is
|      created.
|
|  code_objects
|      The list of `CodeObject` contained within the `BrianObject`.
|
|      TODO: more details.
|
|      Note that this attribute cannot be set directly, you need to modify
|      the underlying list, e.g. ``obj.code_objects.extend([A, B])``.
|
|  contained_objects
|      The list of objects contained within the `BrianObject`.
|
|      When a `BrianObject` is added to a `Network`, its contained objects will
|      be added as well. This allows for compound objects which contain
|      a mini-network structure.
|
|      Note that this attribute cannot be set directly, you need to modify
|      the underlying list, e.g. ``obj.contained_objects.extend([A, B])``.
|
|  name
|      The unique name for this object.
|
|      Used when generating code. Should be an acceptable
|      variable name, i.e. starting with a letter
|      character and followed by alphanumeric characters and
|      ``_``.
|
|  updaters
|      The list of `Updater` that define the runtime behaviour of this object.
|
|      TODO: more details.
|
|      Note that this attribute cannot be set directly, you need to modify
|      the underlying list, e.g. ``obj.updaters.extend([A, B])``.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from brian2.core.base.BrianObject:
|
|  active
|      Whether or not the object should be run.
|
|      Inactive objects will not have their `update`
|      method called in `Network.run`. Note that setting or
|      unsetting the `active` attribute will set or unset
|      it for all `contained_objects`.
|
|  ----------------------------------------------------------------------
|  Data and other attributes inherited from brian2.core.base.BrianObject:
|
|  invalidates_magic_network = True
|
|  ----------------------------------------------------------------------
```

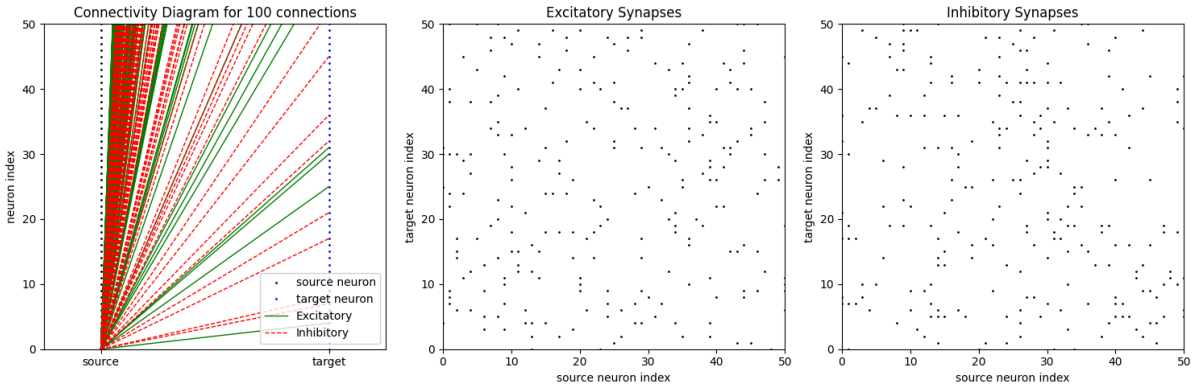
```
|  Methods inherited from brian2.core.names.Nameable:
|
|  assign_id(self)
|      Assign a new id to this object. Under most circumstances, this method
|      should only be called once at the creation of the object to generate a
|      unique id. In the case of the `MagicNetwork`, however, the id should
|      change when a new, independent set of objects is simulated.
|
|  ----------------------------------------------------------------------
|  Readonly properties inherited from brian2.core.names.Nameable:
|
|  id
|      A unique id for this object.
|
|      In contrast to names, which may be reused, the id stays
|      unique. This is used in the dependency checking to not
|      have to deal with the chore of comparing weak
|      references, weak proxies and strong references.
|
|  ----------------------------------------------------------------------
|  Class methods inherited from brian2.core.tracking.Trackable:
|
|  __instances__()
|
|  ----------------------------------------------------------------------
|  Static methods inherited from brian2.core.tracking.Trackable:
|
|  __new__(typ, *args, **kw)
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from brian2.core.tracking.Trackable:
|
|  __dict__
|      dictionary for instance variables
|
|  __weakref__
|      list of weak references to the object
|
|  ----------------------------------------------------------------------
|  Data and other attributes inherited from brian2.core.tracking.Trackable:
|
|  __instancefollower__ = <brian2.core.tracking.InstanceFollower object>
```

In [10]:

```
def show_connections(synaps, ax, lcolor = 'black', ls = '-', label = 'Excitatory',
    # get the source and target neurons
    source, target = synaps.i, synaps.j

    counter = 0
    for i, j in zip(source, target):
        line, = ax.plot([0, 1], [i, j], ls = ls, lw = 1, color = lcolor, label = l
        counter += 1

        if counter > N_max:
```

```
            break
    return line
```

In [11]:
```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize = (5*3, 5))


# plot source and target neuron indices
ax1.set_title('Connectivity Diagram for 100 connections')
line_source, = ax1.plot(np.zeros(N_Excit + N_Inhib), np.arange(N_Excit + N_Inhib),
line_target, = ax1.plot(np.ones(N_Excit + N_Inhib), np.arange(N_Excit + N_Inhib),

# plot the connections
line_ex = show_connections(exc_synapses, ax1, lcolor = 'green')
line_inhib = show_connections(inhib_synapses, ax1, lcolor = 'red', label = 'Inhibi

ax1.set_xticks([0, 1], ['source', 'target'])
ax1.set_ylabel('neuron index')
ax1.legend(handles = [line_source, line_target, line_ex, line_inhib], loc = 'lower

# plot the relation excitatory sources --> target
ax2.set_title('Excitatory Synapses')
i_ex, j_ex = exc_synapses.i, exc_synapses.j
ax2.plot(i_ex, j_ex, ls = '', marker = 'o', markersize = 1, color = 'black')

# plot the relation inhibitory sources --> target
ax3.set_title('Inhibitory Synapses')
i_inhib, j_inhib = inhib_synapses.i, inhib_synapses.j
ax3.plot(np.array(i_inhib), np.array(j_inhib), ls = '', marker = 'o', markersize =

ax1.set_xlim((-0.25, 1.25))
ax1.set_ylim((0, 50))

for ax in (ax2, ax3):
    ax.set_xlim((0, 50))
    ax.set_ylim((0, 50))

    ax.set_xlabel('source neuron index')
    ax.set_ylabel('target neuron index')

plt.tight_layout()
plt.show()
```



A1.2 Answer

The target network is the group of neurons (`NeuronGroup` object) that are targetted by the source neurons (source of spikes). In our case, we have two sources: the excitory neurons and inhibitory neurons which both target the whole network (`network`).

Step 1.3. Enter external Poisson input

Next, you can excite the network through an externally applied Poisson input, by using Poisson Input. Start with N = 1000 external Poisson neurons at an input rate of 13 Hz with a connectivity strength w = w0.

In [12]:
```
# Define PoissonInput parameters
poisson_input_rate = 13 * b2.Hz
N_extern = 1000
w_external = w0
```

In [13]:
```
# Initialize PoissonInput instance
external_poisson_input = PoissonInput(target = network, target_var = 'v', N = N_
```

Step 1.4. Add monitors

In the final step before running the simulation we will add some monitors that allows us to assess the simulated network once the simulation has finished. In order to do so, we will monitor a random selection of 100 neurons and the use the PopulationRateMonitor, SpikeMonitor and StateMonitor.

Sample 200 of all neurons involved.

In [14]:
```
# set random seed
# random.seed(123)

# select random subset of neurons to be monitored
monitored_subset_size = 200
idx_monitored_neurons = sample(range(N_Excit + N_Inhib), monitored_subset_size)
```

In [15]:
```
# rate_monitor: records instantaneous firing rates, averaged across the neurons
rate_monitor = PopulationRateMonitor(network)

# spike_monitor: records spikes from the NeuronGroup
spike_monitor = SpikeMonitor(network, record = idx_monitored_neurons)

# voltage_monitor: records values of the state variable v during a simulation
voltage_monitor = StateMonitor(network, 'v', record = idx_monitored_neurons)
```

Step 1.5. Run the simulation

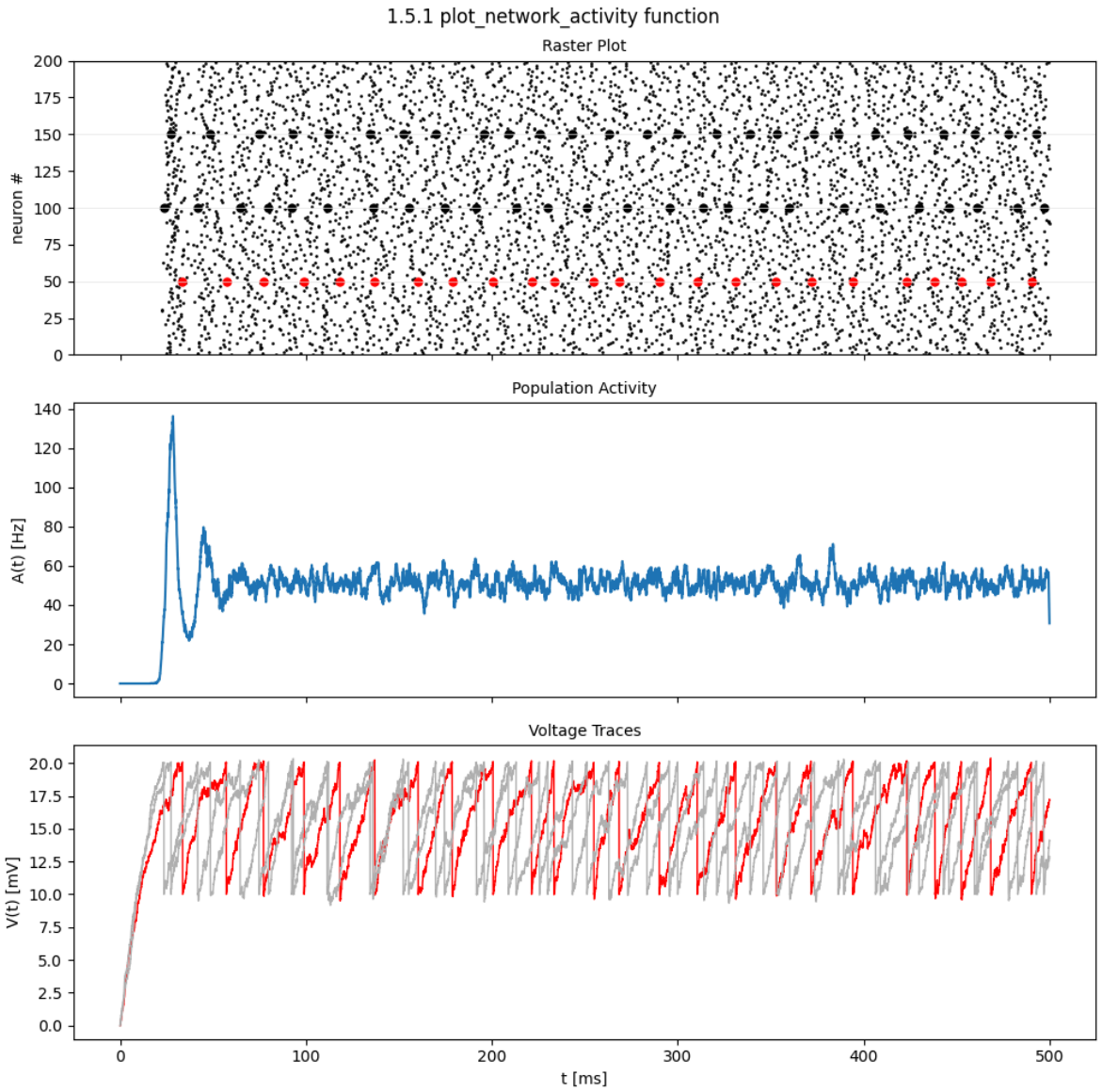
Run the simulation for a total simulated time of 500 ms using the following line of code. Describe what is plotted.

In [16]:
```
sim_time = 500.*b2.ms
b2.run(sim_time)
```

```
WARNING    Came across an abstract code block that may not be well-defined: the o
utcome may depend on the order of execution. You can ignore this warning if you a
re sure that the order of operations does not matter. Abstract code: 'v += J_exci
t * (rand() < connection_probability) (in-place)'
 [brian2.codegen.generators.base]
WARNING    Came across an abstract code block that may not be well-defined: the o
utcome may depend on the order of execution. You can ignore this warning if you a
re sure that the order of operations does not matter. Abstract code: 'v += J_inhi
b * (rand() < connection_probability) (in-place)'
 [brian2.codegen.generators.base]
```

In [17]:
```
plot_tools.plot_network_activity(rate_monitor, spike_monitor, \
                                 voltage_monitor, spike_train_idx_list = idx_mon
                                 t_min = 0 * b2.ms, t_max = sim_time,
                                 figure_size=(10, 10))

# plt.xlim(0, 500)
plt.suptitle(r'1.5.1 plot_network_activity function')
plt.tight_layout()
plt.show()
```

```
INFO       width adjusted from 1. ms to 1.05 ms [brian2.monitors.ratemonitor.adju
sted_width]
```



Access the data in rate_monitor, spike_monitor, voltage_monitor

First, describe what is described in rate_monitor.rate. Use the smooth_rate function (flat window) to the outputted rates. Make sure to only include the last 150 ms of your simulation. Plot the smoothed_rate in function of time and calculate the mean of the rate across this time window.

You can the time axis from rate_monitor.t

In [18]:
```
ts = rate_monitor.t / b2.ms
t_min = 350; t_max = 500
```

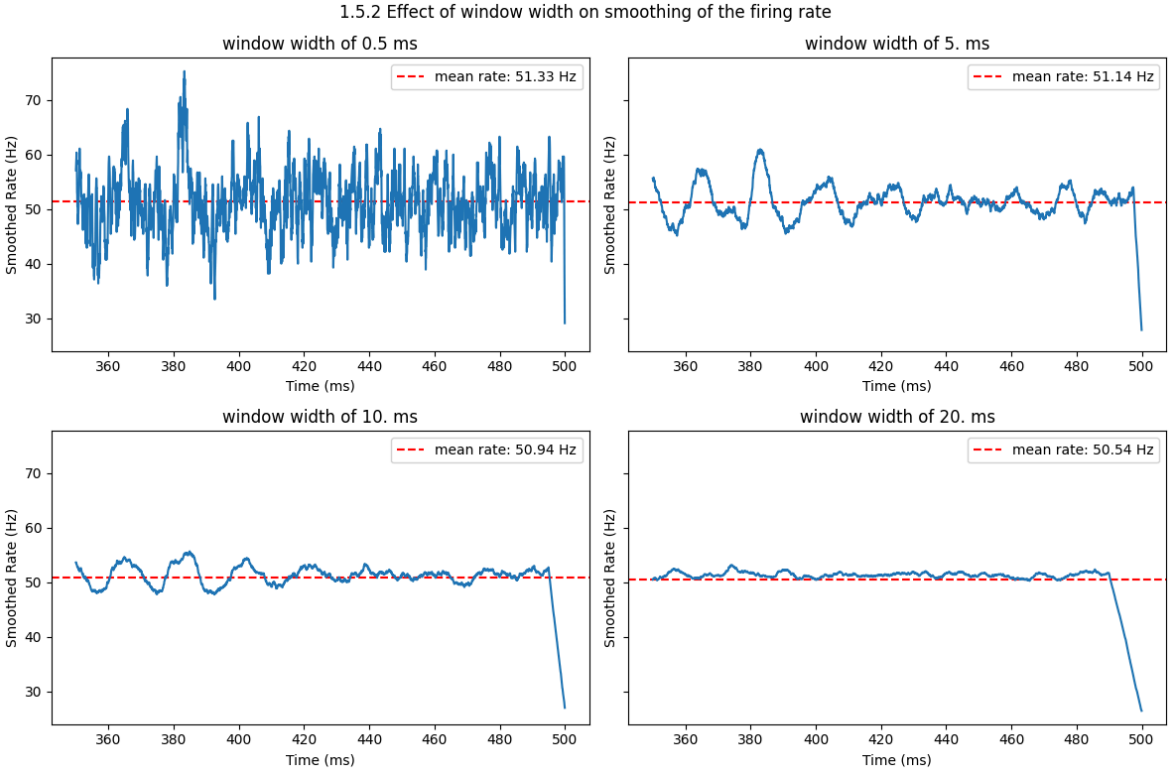
In [19]:
```
idx_rate = np.where((t_min <= ts) & (ts < t_max))
print(rate_monitor.rate)
```

```
<ratemonitor.rate: array([ 0.,  0.,  0., ..., 64., 68., 40.]) * hertz>
```

Now, vary the window width across which the rate is averaged and write down your observations (do plot!).

In [20]:
```
window_widths = [0.5, 5, 10, 20] * b2.ms

fig, axs = plt.subplots(2, 2, figsize = (12, 4*2), sharey = True)
fig.suptitle('1.5.2 Effect of window width on smoothing of the firing rate')

for i in range(0, 4):
    smoothed_rates = rate_monitor.smooth_rate(window = "flat", width = window_wi

    mean_rate = np.mean(smoothed_rates[idx_rate])
    axs[i//2, i%2].axhline(y = mean_rate, color = 'red', ls = '--', label = f'me
    axs[i//2, i%2].plot(ts[idx_rate], smoothed_rates[idx_rate])

    axs[i//2, i%2].set_xlabel('Time (ms)')
    axs[i//2, i%2].set_ylabel('Smoothed Rate (Hz)')
    axs[i//2, i%2].set_title(f'window width of {window_widths[i]}')

    axs[i//2, i%2].legend()

plt.tight_layout()
plt.show()
```

```
INFO       width adjusted from 0.5 ms to 0.55 ms [brian2.monitors.ratemonitor.adj
usted_width]
INFO       width adjusted from 5. ms to 5.05 ms [brian2.monitors.ratemonitor.adju
sted_width]
INFO       width adjusted from 10. ms to 10.05 ms [brian2.monitors.ratemonitor.ad
justed_width]
INFO       width adjusted from 20. ms to 20.05 ms [brian2.monitors.ratemonitor.ad
justed_width]
```



Now, study what is saved in spike_monitor. Start by creating eventtrains. What is the mean time between two subsequent spikes?

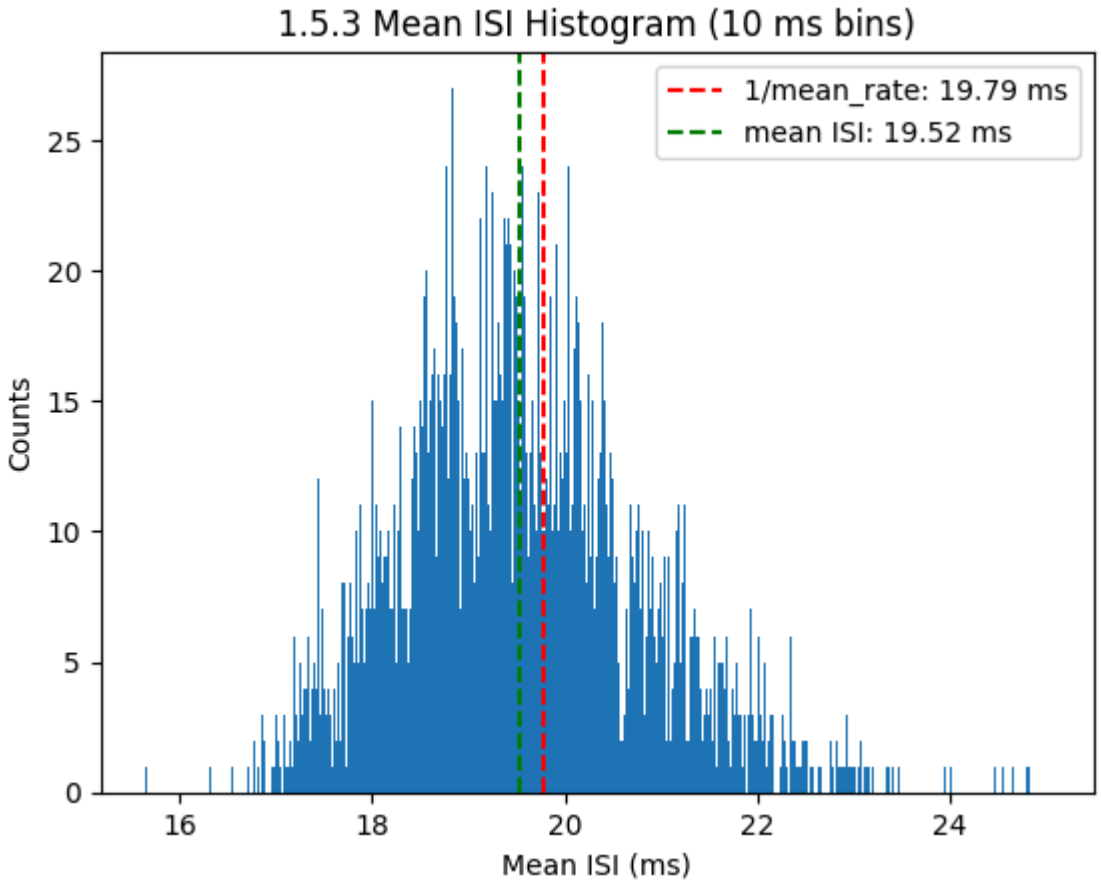
In [21]:
```
eventtrains = spike_monitor.event_trains()

ISIs = []
for train in eventtrains.values():
    ISI = np.diff(train)
    if len(ISI) > 1:
        ISIs.append(np.array(ISI / b2.ms).mean())

ISIs = np.array(ISIs)
print(len(ISIs))
```

```
5000
```

In [22]:
```
mean = ISIs.mean()
print(f"Mean time between two subsequent spikes: {mean:.2f} ms")
```

```
Mean time between two subsequent spikes: 19.52 ms
```

In [23]:
```
dt = 1e-2
bin_edges = np.arange(ISIs.min(), np.percentile(ISIs, 100), dt)

plt.figure()

plt.hist(ISIs, bin_edges)

plt.axvline(x = 1 / mean_rate * 1e3, color = 'red', ls = '--', label = f'1/mean_
plt.axvline(x = mean, color = 'green', ls = '--', label = f'mean ISI: {mean:.2f}

plt.ylabel('Counts')
plt.xlabel('Mean ISI (ms)')
plt.title('1.5.3 Mean ISI Histogram (10 ms bins)')

plt.legend()
plt.show()
```



A1.5 Answer

The function `plot_network_activity()` offers a visualization of the simulation results, focusing on the 200 monitored neurons within the network. Plot 1.5.1 displays a dot-raster plot illustrating the spike trains of the monitored neurons, the network activity $A(t)$, and voltage traces of selected neurons. Notice that the red color in the bottom plot and top panel corresponds to the same neuron, as does the relationship between black dots and gray lines.

The `rate_monitor.rate` instance provides insight into the averaged instantaneous firing rate across all neurons in the network for each time point during the simulation.

Plot 1.5.2 showcases the averaged firing rate using varying time windows: 0.5, 5, 10, and 20 ms. Analysis reveals that a 0.5 ms window width exhibits numerous high-frequency oscillations, hindering the observation of a clear pattern in the firing rate. Conversely, setting the window width to 5 ms and 10 ms reveals clear oscillatory behavior around the mean firing rate, enabling us to see a clear pattern in the temporal evolution of the firing rate. Increasing the window width to 20 ms yields a relatively straight line, indicating the mean firing rate. The mean firing rate is approximately 51 Hz.

The selection of a window width involves a trade-off between capturing high-frequency information (small window width) and discerning an overall trend in the signal (larger window width). As the window width increases, high-amplitude values are averaged out, leading to a decrease in overall amplitude.

The spike_monitor object records spikes from the monitored neurons, accessible via the `spike_trains()` -method, which returns a dictionary mapping neuron indices to spike trains. These spike trains are further utilized to calculate the mean ISI.

Figure 1.5.3 presents the distribution of mean ISIs, confirming that the calculated mean ISI aligns closely with the inverse of the mean firing frequency, as expected.

2. The pre-implemented Brunel network

Import the LIF_spiking_network function from neurodynex3.brunel_model and use this function to simulate a network consisting of 10000 excitatory neurons, 2500 inhibitory neurons and 1000 external neurons.

Further,

- w0 = 0.1 mV
- total simulated time of 500 ms
- the membrane time scale can be put to default (LIF_spiking_network.MEMBRANE_TIME_SCALE)
- the same goes for the firing threshold (LIF_spiking_network.FIRING_THRESHOLD)
- monitored_subset = 50
- synaptic delay is 1.5ms

We will vary two parameters: g (an input parameter to simulate_brunel_network) and, second, the firing frequency of the external neurons ν_{extern}. The latter should be expressed as a ratio multiplied with $\nu_{threshold}$. The frequency $\nu_{threshold}$ is the minimal poisson rate in the external neuronal population required to elicit firing in the network in the absence of any feedback.

According to Brunel (2000), $\nu_{threshold}$ can be calculated as:

$$\nu_{threshold} = \frac{\theta}{N_{extern} w_0 \tau_m} \tag{1}$$

Does this expression make sense?

For starters, you can use g=6 and $\nu_{extern} = 4\nu_{threshold}$

Calculate $\nu_{threshold}$ in function of the parameters mentioned above and the corresponding ν_{extern}. And run the simulation. Make sure to output rate_monitor,

spike_monitor, voltage_monitor and monitored_spike_idx.

Make sure to start each simulation with a "b2.start_scope()" statement.

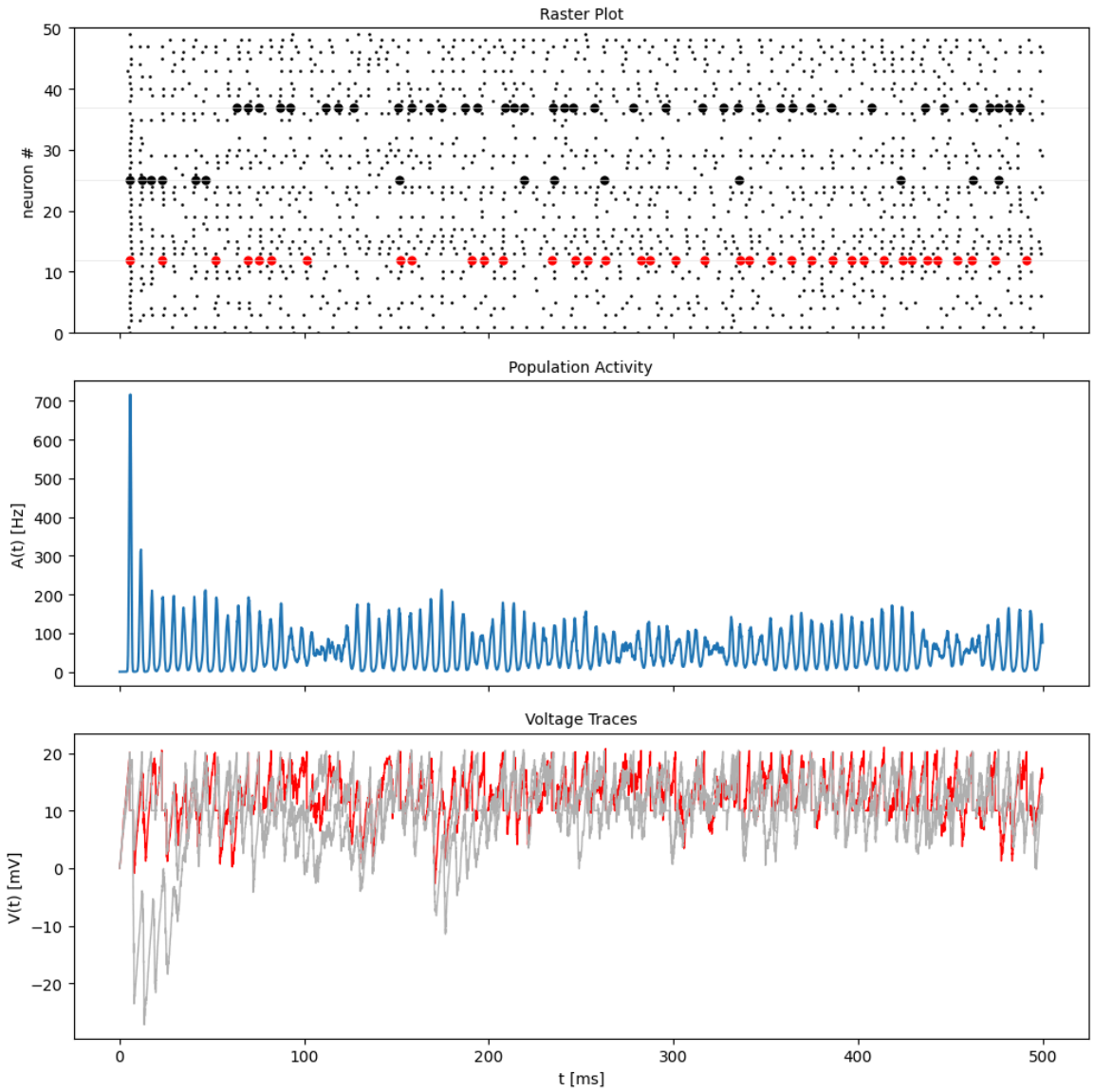
In [24]:
```
from neurodynex3.brunel_model import LIF_spiking_network
b2.start_scope()

N_Excit = 10000
N_Inhib = 2500
N_extern = 1000

w0 = 0.1 * b2.mV
sim_time = 500 * b2.ms
monitored_subset_size = 50
SYNAPTIC_DELAY = 1.5 * b2.ms
g = 6

nu_threshold = LIF_spiking_network.FIRING_THRESHOLD / (N_extern * w0 * LIF_spik
nu_extern = 4 * nu_threshold

print(f"nu_threshold =\t{nu_threshold} \nnu_extern =\t{nu_extern}")
```

```
nu_threshold =  10. Hz
nu_extern =     40. Hz
```

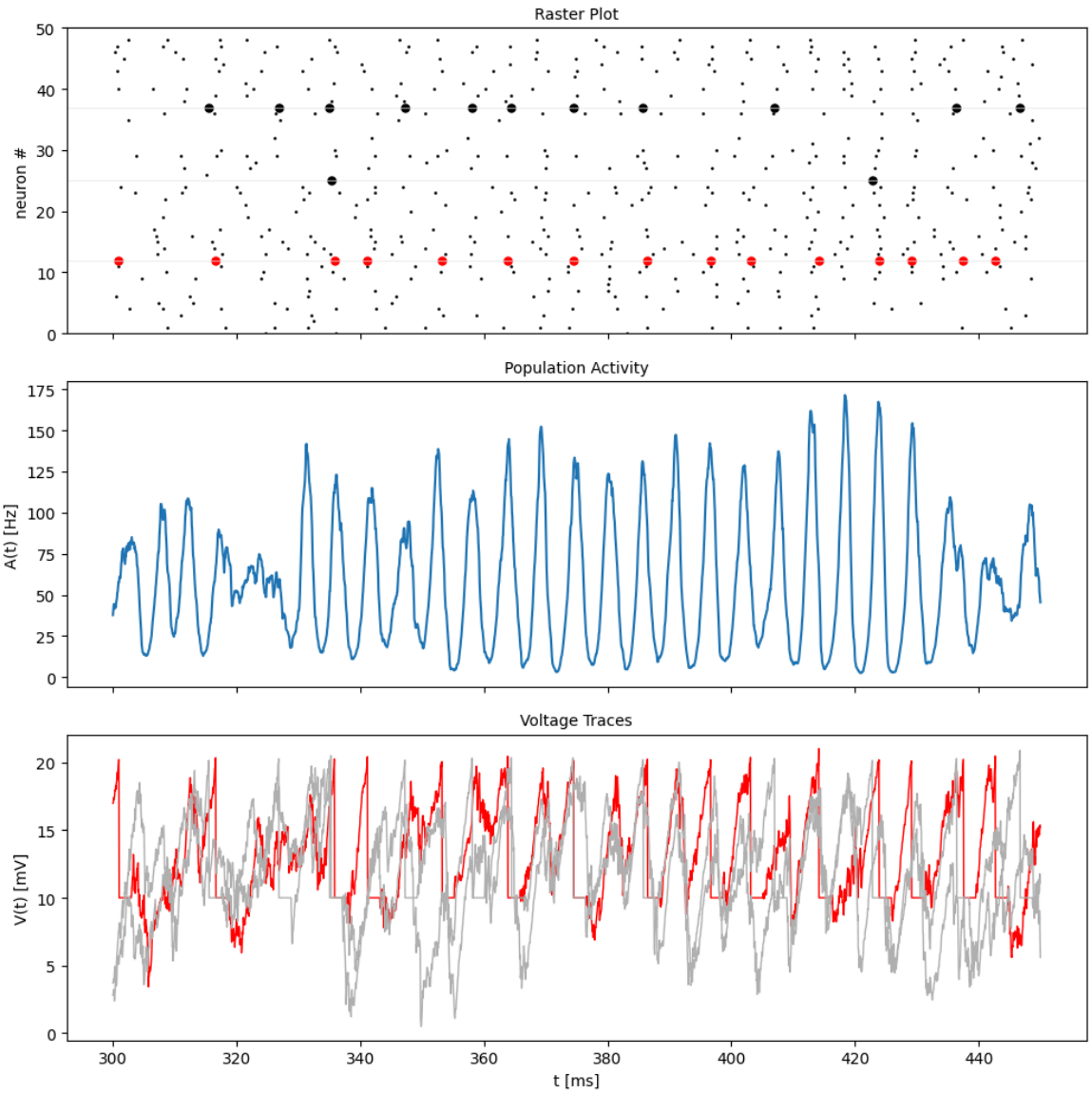
In [25]:
```
rate_monitor, spike_monitor, voltage_monitor, monitored_spike_idx = \
    LIF_spiking_network.simulate_brunel_network(N_Excit = N_Excit, N_Inhib = N_
                                                monitored_subset_size = monitor
```

Plot the output using "plot_network_activity", make sure to add the "t_min" and "t_max" parameter and increase the figure size to figure_size=(10,10).

In [26]:
```
plot_tools.plot_network_activity(rate_monitor, spike_monitor, \
                                 voltage_monitor, spike_train_idx_list = monitc
                                 t_min = 0 * b2.ms, t_max = sim_time,\
                                 figure_size=(10, 10))

plt.tight_layout()
plt.show()
```



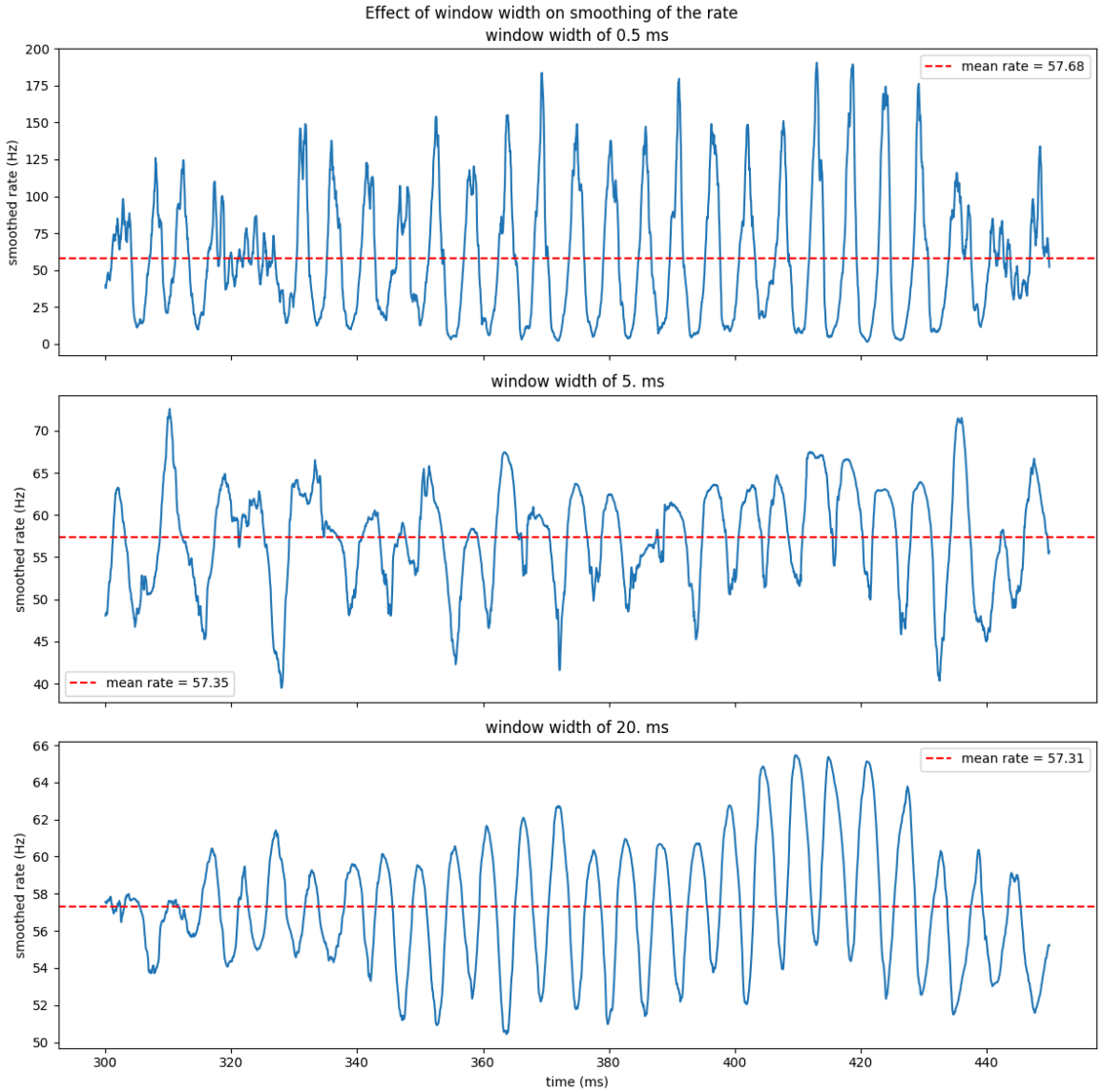
In [27]:
```
plot_tools.plot_network_activity(rate_monitor, spike_monitor, \
                                 voltage_monitor, spike_train_idx_list=monitore
                                 t_min = 300 * b2.ms, t_max = 450 * b2.ms,\
                                 figure_size=(10, 10))

plt.tight_layout()
plt.show()
```



The first output to analyse is the rate_monitor. Plot - for different choices of window width (0.5, 5, 20) ms - the smoothed_rates and calculate the mean of smoothed_rate across a time window (e.g. between 300 and 450 ms). How does this compare to Table 1 in Brunel (2000)?

In [28]:
```
# Get time axis
t = rate_monitor.t / b2.ms
t_min, t_max = 300, 450
idx_rate = (t_min <= ts) * (ts <= t_max)

# Investigate the effect window width on smoothing of the rate
window_widths = [0.5, 5, 20]*b2.ms

fig, axs = plt.subplots(3, 1, figsize = (12, 4 * len(window_widths)), sharex =
fig.suptitle('Effect of window width on smoothing of the rate')

for i, window in enumerate(window_widths):
    smoothed_rates = rate_monitor.smooth_rate(window="flat", width=window)/b2.H
```

```
    axs[i].plot(ts[idx_rate], smoothed_rates[idx_rate])

    mean_rate = np.mean(smoothed_rates[idx_rate])
    axs[i].axhline(y = mean_rate, color = 'red', ls = '--', label = f'mean rate

    axs[i].set_ylabel('smoothed rate (Hz)')
    axs[i].set_title(f'window width of {window}')

    axs[i].legend()

axs[-1].set_xlabel('time (ms)')

plt.tight_layout()
plt.show()
```



Next, we want to assess the global frequency content in rate_monitor. We know that if we have a time signal of 0.5 seconds, the maximal frequency resolution is 2 Hz. Be aware that Welch' method as implemented will chop the data in different pieces (to enable the averaging of the spectrum) yielding a smaller frequency resolution.

Start by smoothing the rate with a window of 0.5 ms and calculate the spectrum using signal.welch. What is the sampling frequency fs?

Plot the spectrum. What is the frequency resolution (look at np.diff(f)[0]). Can you explain how signal.welch arrives at this frequency resolution if you know that it - by default - includes 256 samples and if you check the integration time constant (typically 0.05 ms, but check b2.defaultclock.dt) that you used for the simulation?

What happens if you impose nperseg = 256*8?

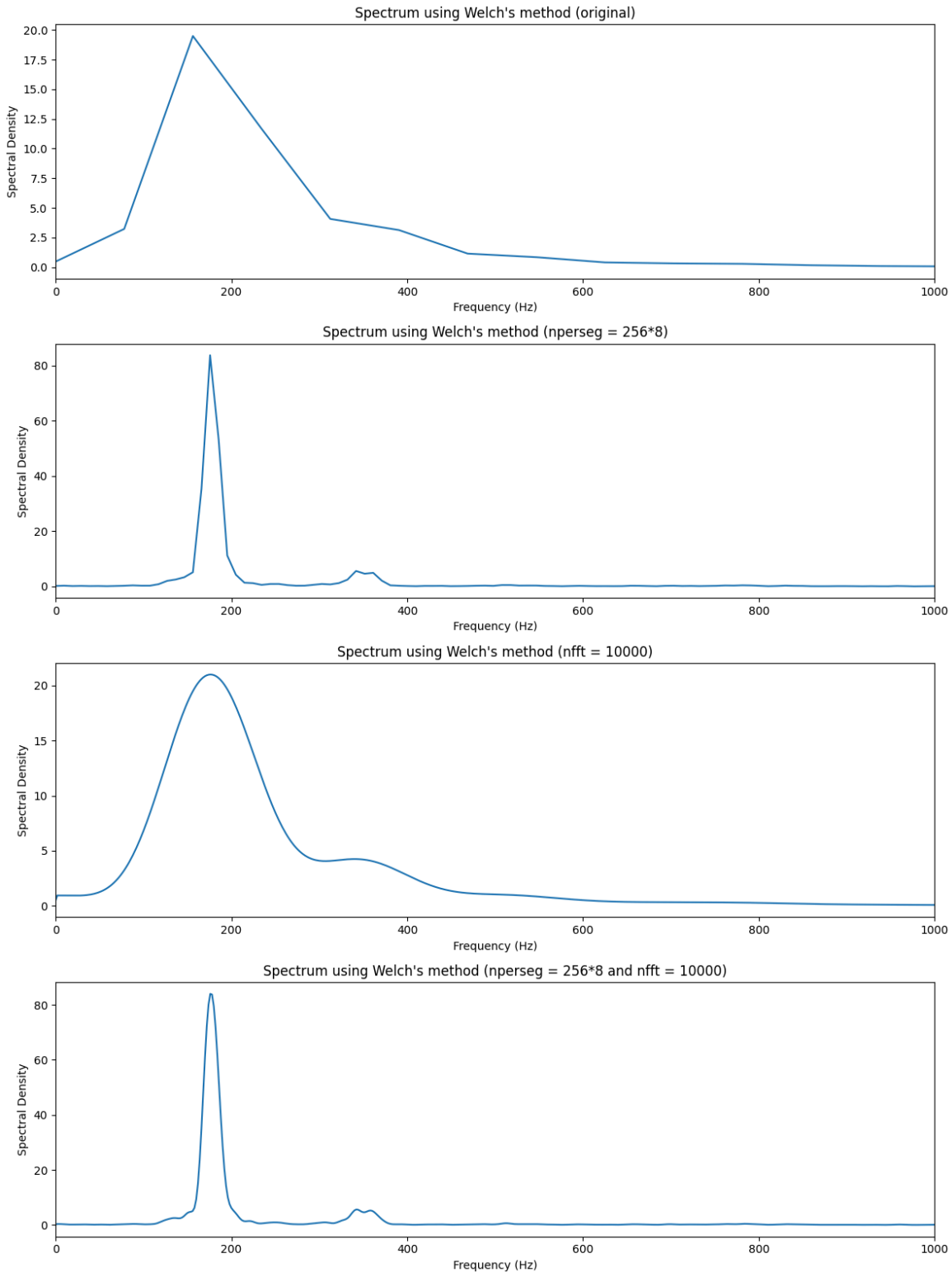
What happens if you impose nfft = 10000 ? (Answer: you interpolate the spectrum).

In [29]:
```
# Smoothing the rate
smoothed_rates = rate_monitor.smooth_rate(window = "flat", width = 0.5 * b2.ms)

# calculating the spectra
from scipy import signal
fs = 1 / (np.diff(ts)[0] * 1e-3)
f, X = signal.welch(smoothed_rates, fs = fs) #nperseg = 256*8, nfft = 10000)
f2, X2 = signal.welch(smoothed_rates, fs = fs, nperseg = 256*8) #, nfft = 10000
f3, X3 = signal.welch(smoothed_rates, fs = fs, nfft = 10000)
f4, X4 = signal.welch(smoothed_rates, fs = fs, nperseg = 256*8, nfft = 10000)

# check integration time constant
print(f"Integration time constant: {b2.defaultclock.dt}")

# sampling frequencies
print(f"Sampling frequency: {fs:.2f} Hz")

# frequency resolutions
f_res = [np.diff(fi)[0] for fi in [f, f2, f3, f4]]
print(f"Frequency resolution: {f_res} Hz")

vals = [(f, X), (f2, X2), (f3, X3), (f4, X4)]
```

```
Integration time constant: 50. us
Sampling frequency: 20000.00 Hz
Frequency resolution: [78.125, 9.765625, 2.0, 2.0] Hz
```

In [30]:
```
# plot the spectra
fig, ax = plt.subplots(4, 1, figsize = (12, 4 * 4))

labels = ['original', 'nperseg = 256*8', 'nfft = 10000', 'nperseg = 256*8 and n

for i, axi in enumerate(ax):
    axi.set_ylabel('Spectral Density')
    axi.set_xlabel('Frequency (Hz)')
    axi.set_title(f'Spectrum using Welch\'s method ({labels[i]})')

    axi.plot(vals[i][0], vals[i][1])

    axi.set_xlim((0, 1000))
```

```
plt.tight_layout()
plt.show()
```



We will now use the machinery provided by BRIAN to calculate and plot the power spectrum. BRIAN takes a slightly different approach. BRIAN starts from a desired frequency resolution, a desired number of windows over which the spectrum should be averaged and an initial time segment that should be ignored. Based on these desiredata, the minimal simulation length is calculated.

Example: if we want a frequency resolution of 5 Hz, we need time windows of 200 ms. If we assume non-overlapping time windows and a number of averages the total simulated signal should be k*200ms.

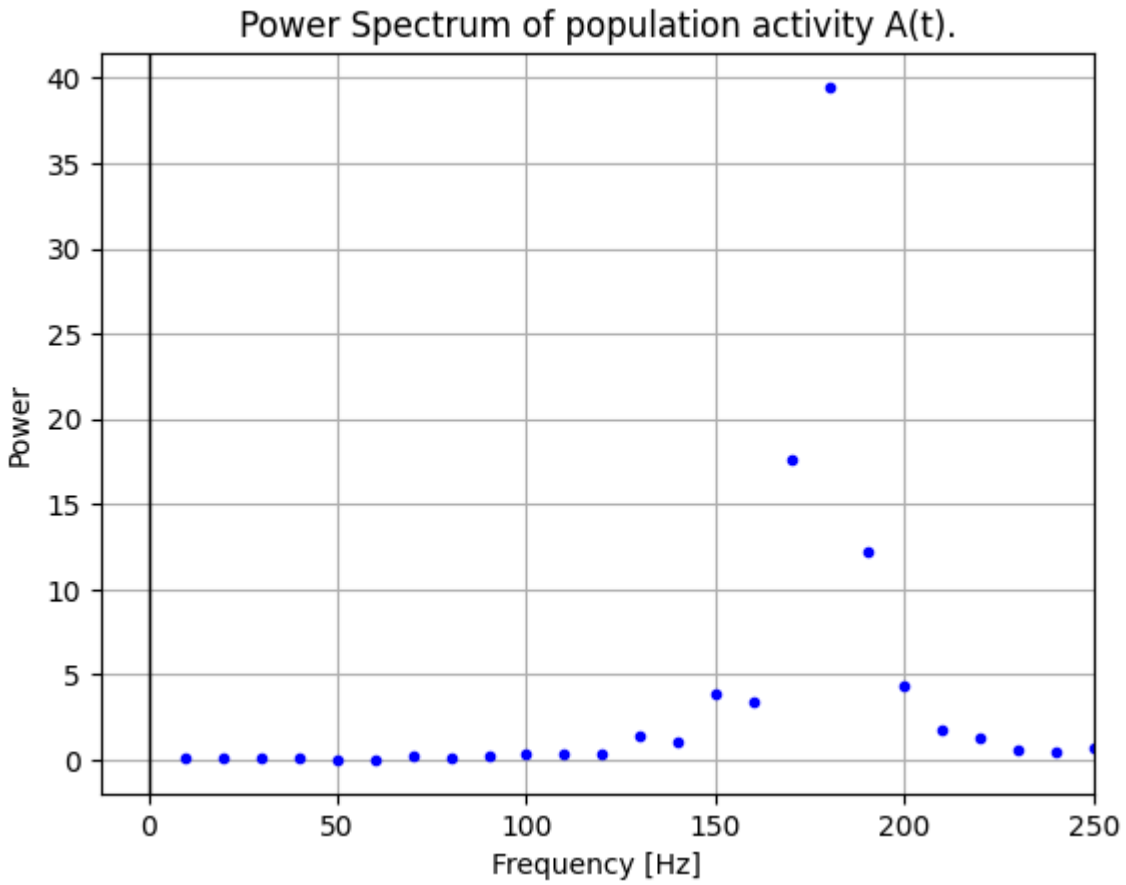
In this case, we have simulated 0.5 seconds of data. If we aim for an initial segment to be removed of 99ms and a frequency resolution of 10 Hz, what is then the maximal number of windows we can include?

In [31]:
```
# Adjusted version of (corrupted) get_population_activity_power_spectrum() meth
def get_population_activity_power_spectrum(
        rate_monitor, delta_f, k_repetitions, T_init=100*b2.ms, subtract_mean_a
    """
    Computes the power spectrum of the population activity A(t) (=rate_monitor.

    Args:
        rate_monitor (RateMonitor): Brian2 rate monitor. rate_monitor.rate is t
            analysed here. The temporal resolution is read from rate_monitor.cl
        delta_f (Quantity): The desired frequency resolution.
        k_repetitions (int): The data rate_monitor.rate is split into k_repetit
            independently and then averaged in frequency domain.
        T_init (Quantity): Rates in the time interval [0, T_init] are removed b
            Fourier transform. Use this parameter to ignore the initial transie
        subtract_mean_activity (bool): If true, the mean value of the signal is

    Returns:
        freqs, ps, average_population_rate
    """
    data = rate_monitor.rate/b2.Hz
    delta_t = rate_monitor.clock.dt
    f_max = 1./(2. * delta_t)
    N_signal = int(2 * f_max / delta_f)
    T_signal = N_signal * delta_t
    N_init = int(T_init/delta_t)
    N_required = k_repetitions * N_signal + N_init
    N_data = len(data)

    # print("N_data={}, N_required={}".format(N_data,N_required))
    if (N_data < N_required):
        err_msg = "Inconsistent parameters. k_repetitions require {} samples."
                  " rate_monitor.rate contains {} samples.".format(N_required,
        raise ValueError(err_msg)
    if N_data > N_required:
        # print("drop samples")
        data = data[:N_required]
    # print("length after dropping end:{}".format(len(data)))
    data = data[N_init:]
    # print("length after dropping init:{}".format(len(data)))
    average_population_rate = np.mean(data)
    if subtract_mean_activity:
        data = data - average_population_rate
    average_population_rate *= b2.Hz
    data = data.reshape(k_repetitions, N_signal)  # reshape into one row per re
    k_ps = np.abs(np.fft.fft(data))**2
    ps = np.mean(k_ps, 0)
```

```
    # normalize
    ps = ps * delta_t / N_signal  # TODO: verify: subtract 1 (N_signal-1)?
    freqs = np.fft.fftfreq(N_signal, delta_t)
    ps = ps[:int((N_signal/2))]
    freqs = freqs[:int((N_signal/2))]
    return freqs, ps, average_population_rate
```

In [32]:
```
delta_f = 10*b2.Hz
k = 4
T_init = 99*b2.ms + b2.defaultclock.dt;
pop_freqs, pop_ps, average_population_rate = get_population_activity_power_spec
```

In [33]:
```
# Plot population activity PS
plot_tools.plot_population_activity_power_spectrum(pop_freqs, pop_ps, 250 *b2.H
plt.show()
```



2 Answer

The expression for $\nu_{\text{threshold}}$ is logically as it establishes a relationship between the threshold firing rate $\nu_{\text{threshold}}$ of a neuron and the external input it receives. This relationship considers the number of external inputs N_{extern}, the strength of the synapses w_0, and the membrane time constant τ_m. The parameter θ represents the firing threshold. This derivation is based on the assumption that the neuron receives input from a large number of other neurons, and that the input is weak and uncorrelated.

Similar to the findings in the previous exercise, the network activity plots exhibit the expected behavior. The mean time between spikes appears to be slightly less than 20 ms, as can be visaully observed from the raster plot. A smoothed rate with a window width of 5 ms provides the most accurate results, for the same reasons mentioned in the previous exercise. During the investigated period of 300 to 450 ms, clear peaks in the rate are observed, with an average interval of 15 - 20 ms between each.

When comparing the results to those of the previous exercise, it is evident that the rate follows a less oscillatory shape in the simulation with the pre-implemented Brunel network (in contrast to the first approach).

To calculate the spectrum, we first smooth the rate with a window of 0.5 ms, and then utilize the `signal.welch` function to compute the power spectral density (PSD). The default value of `nperseg` is 256, indicating that the signal is segmented into 256 samples before computing the spectrum. Here, the integration time constant (`b2.defaultclock.dt`) used for the simulation is 0.05 ms, resulting in a sampling frequency of 1/0.05 = 20 000 Hz.

The frequency resolution of the spectrum can be calculated using,

$$\Delta f = \frac{f_s}{N}, \tag{1}$$

where N is the number of samples in each segment. Thus, for the default `nperseg` value of 256, the frequency resolution is

$$\Delta f = \frac{20000}{256} = 78.13/\mathrm{Hz}$$

Increasing nperseg to 256×8 would decrease Δf to 9.77 Hz, resulting in a higher frequency resolution.

Setting `nfft` to 10 000 would set the length of the FFT used to compute the spectrum to 10 000, which is larger than the default `nperseg` value. This would interpolate the spectrum (zero-padding), theoretically resulting in a lower frequency resolution of 2 Hz. However, this does not provide additional information about the spectrum compared to the first case (`nperseg` =256). Instead, the graphical representation is smoothed out to yield a visually appealing plot, enhancing the interpretability of the spectrum. Increasing `nfft` (i.e. zero-padding) does not add any information but improves the visualization.

To calculate the power spectrum using the brian package, we need to choose the number of windows or segments to chop the data into. In this case, we have

simulated 0.5 seconds of data. We aim for an initial segment to be removed of 99 ms and a frequency resolution of 10 Hz. Taking these parameters into account, the maximal number of windows is $k = 4$, based on the following reasoning:

The simulation time was originally 500 ms, but the first 99 ms are not used, leaving us with 401 ms of data. We want a frequency resolution of 10 Hz, which corresponds to time windows of 100 ms. The number of windows is calculated as:

$$k = \frac{\Delta t_{\text{sim}}}{\Delta t_{\text{window}}} = \frac{401}{100} \approx 4.$$

Therefore, to calculate the power spectrum with a frequency resolution of 10 Hz using the brian package, we need to chop the data into four windows of 100 ms each.

3 Explore the dynamics.

Repeat the steps in ##2 for the following set of parameters:

(A) g = 3, $\nu_{extern}/\nu_{threshold} = 2$

(B) g = 6, $\nu_{extern}/\nu_{threshold} = 2$

(C) g = 4.5, $\nu_{extern}/\nu_{threshold} = 0.95$

Describe what you observe.

In [34]:
```
from scipy.signal import welch

class DynamicsExplorer:

    def __init__(self, g, nu_extern):
        b2.start_scope()
        # Parameters to be changed
        self.g, self.nu_extern = g, nu_extern

        # Nuber of each neuron group
        self.N_Excit, self.N_inhib, self.N_extern = 10000, 2500, 1000

        # Model parameters
        self.w0 = 0.1 * b2.mV

        # Simulation parameters
        self.sim_time = 500 * b2.ms
        self.monitored_subset_size = 50

        self.window_widths = [0.5,5,20] * b2.ms

    def simulate(self):
        self.rate_monitor, self.spike_monitor, self.voltage_monitor, self.mon
```

```
        LIF_spiking_network.simulate_brunel_network(N_Excit = self.N_Excit, N_
                                                    w0 = self.w0, sim_time = s
                                                    monitored_subset_size = s
                                                    poisson_input_rate = self
    def show_simulation(self, t_min = 300*b2.ms, t_max = 450*b2.ms):
        plot_tools.plot_network_activity(self.rate_monitor, self.spike_monitor
                                         self.voltage_monitor, spike_train_idx_list
                                         t_min = t_min, t_max = t_max,\
                                         figure_size = (12, 4*3))
        plt.tight_layout()
        plt.show()


    def show_smoothed_rate(self, tmin = 300, tmax = 450 ):
        self.ts = self.rate_monitor.t / b2.ms
        idx_rate = (t_min <= ts) * (ts < t_max)


        fig, axs = plt.subplots(3, 1, figsize = (12, len(self.window_widths)*4
        fig.suptitle('Effect of window width on Firing Rate')

        for i, window in enumerate(self.window_widths):
            smoothed_rates = self.rate_monitor.smooth_rate(window = "flat", wi
            axs[i].plot(ts[idx_rate], smoothed_rates[idx_rate])

            mean_rate = np.mean(smoothed_rates[idx_rate])
            axs[i].axhline(y = mean_rate, color = 'red', ls = '--', label = f

            axs[i].set_ylabel('Firing Rate (Hz)')
            axs[i].set_title(f'window width of {window}')
            axs[i].legend()

        axs[-1].set_xlabel('Time (ms)')
        plt.tight_layout()
        plt.show()

    def get_spectrum(self, width = 0.5*b2.ms):
        # Smooth rate (window = 0.5 ms)
        self.smoothed_rates = self.rate_monitor.smooth_rate(window = "flat", w
        self.Fs = 1 / (self.ts[1] - self.ts[0]) * 1e3

        # Calculate spectrum (welch method)
        self.freqs, self.smoothed_rates_spectrum = welch(self.smoothed_rates,
        self.freq_res = np.diff(self.freqs)[0]

        self.freqs_nperseg, self.smoothed_rates_spectrum_nperseg = welch(self
        self.freq_res_nperseg = np.diff(self.freqs_nperseg)[0]

        self.freqs_nfft, self.smoothed_rates_spectrum_nfft = welch(self.smoothl
        self.freq_res_nfft = np.diff(self.freqs_nfft)[0]

        self.freqs_X, self.smoothed_rates_X = welch(self.smoothed_rates, fs =
        self.freq_res_X = np.diff(self.freqs_X)[0]

    def get_freqres(self):
         print(f"""
```

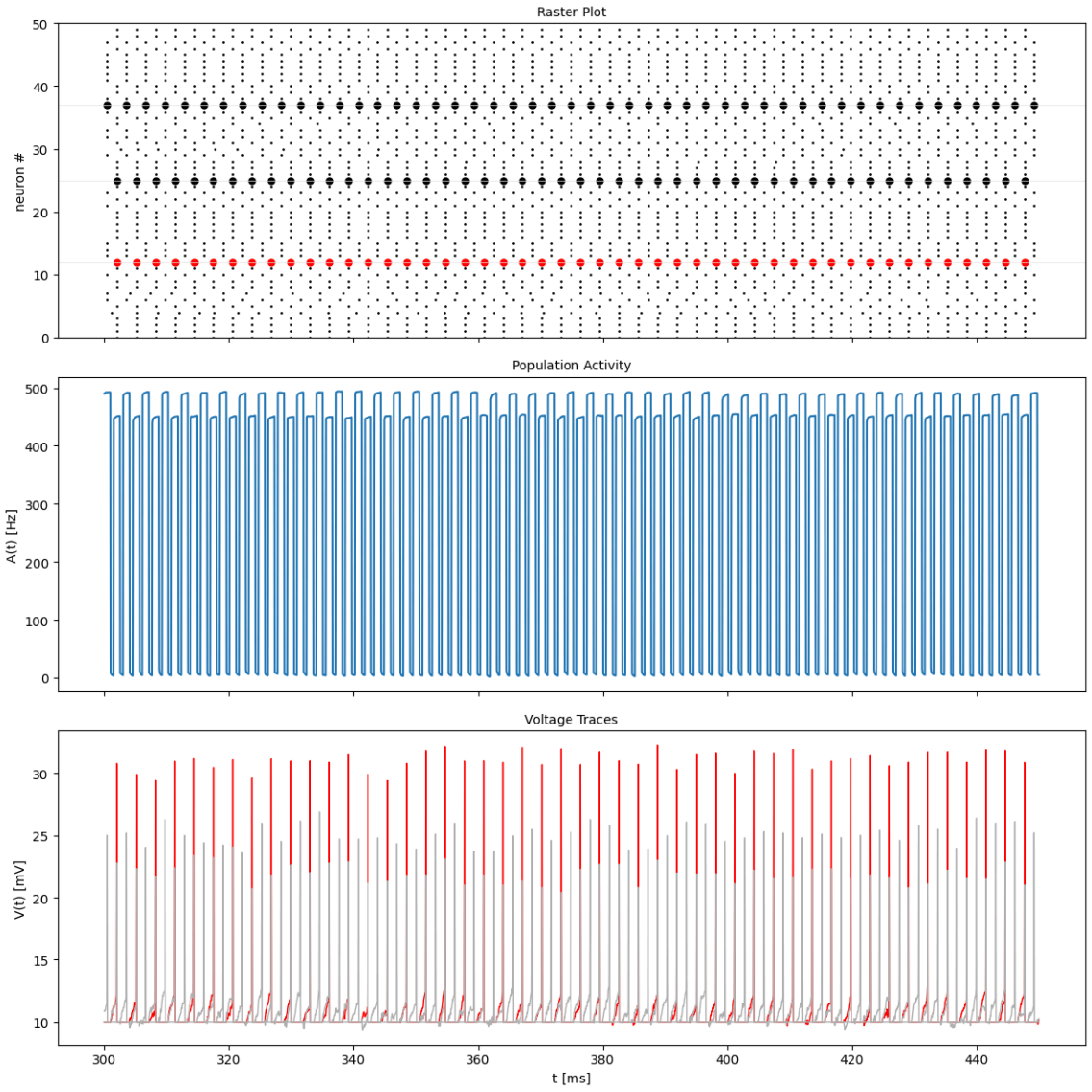
```
Sampling frequency fs\t\t= {self.Fs} Hz
Integration time constant\t= {b2.defaultclock.dt}
-------------------------------------------------------------
Welch frequency resolution\t\t\t= {self.freq_res} Hz
Welch frequency resolution (nperseg=256*8)\t= {self.freq_res_nperseg} Hz
Welch frequency 'resolution' (nfft=10000)\t= {self.freq_res_nfft} Hz
Welch frequency 'resolution' (nperseg=256*8 & nfft=10000)\t= {self.freq_res_X
""")

    def show_PSD(self, f_lim = 1000):
        # Plots: effect of changing parameters in welch method
        fig, axs = plt.subplots(4, 1, figsize = (12, 4 * 4), sharex = True)

        axs[0].plot(self.freqs, self.smoothed_rates_spectrum)
        axs[1].plot(self.freqs_nperseg, self.smoothed_rates_spectrum_nperseg)
        axs[2].plot(self.freqs_nfft, self.smoothed_rates_spectrum_nfft)
        axs[3].plot(self.freqs_X, self.smoothed_rates_X)

        fig.suptitle('Effect of changing parameters in Welch method on the Pow
        axs[-1].set_xlabel('Frequency (Hz)')

        axs[0].set_title('Standard Welch method')
        axs[1].set_title('nperseg = 256*8')
        axs[2].set_title('nfft = 10 000')
        axs[3].set_title('nperseg = 256*8 & nfft = 10 000')

        for ax in axs:
            ax.set_ylabel('PSD')
            ax.set_xlim((0, f_lim))

        plt.tight_layout()
        plt.show()

    def get_population_activity(self, f_lim = 600):
         # Compute population activity PSD
        delta_f = 10*b2.Hz
        k = 4
        T_init = 99 * b2.ms + b2.defaultclock.dt
        pop_freqs, pop_ps, average_population_rate = get_population_activity_p

        # Plot population activity
        plot_tools.plot_population_activity_power_spectrum(pop_freqs, pop_ps,
        plt.show()
```

A : g = 3, $\nu_{extern}/\nu_{threshold} = 2$

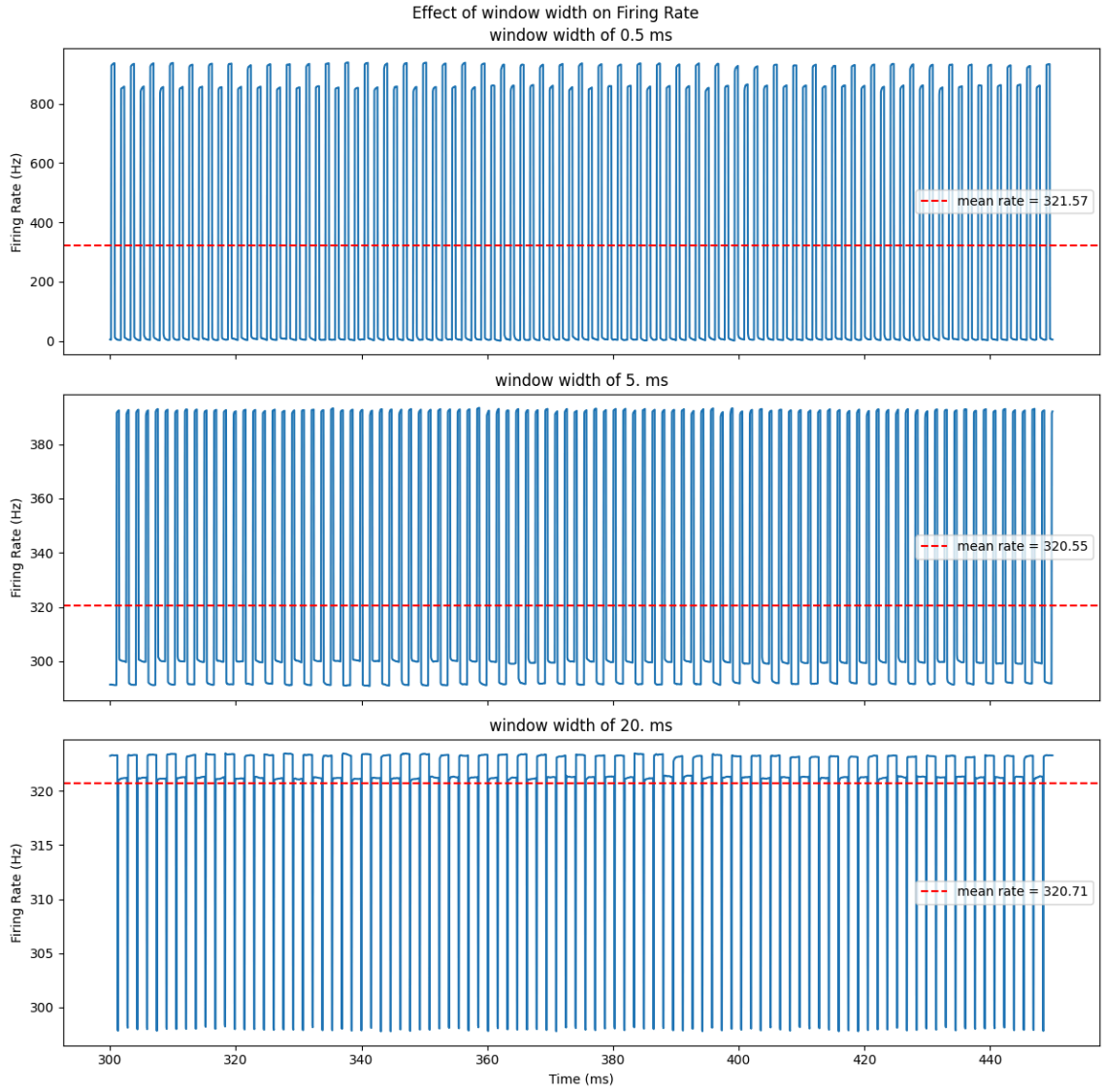
In [35]:

```
g = 3
nu_threshold = LIF_spiking_network.FIRING_THRESHOLD / (N_extern * w0 * LIF_spi
nu_extern = 2*nu_threshold

DEA = DynamicsExplorer(g, nu_extern)
DEA.simulate()
```

In [36]:
```
DEA.show_simulation()
```



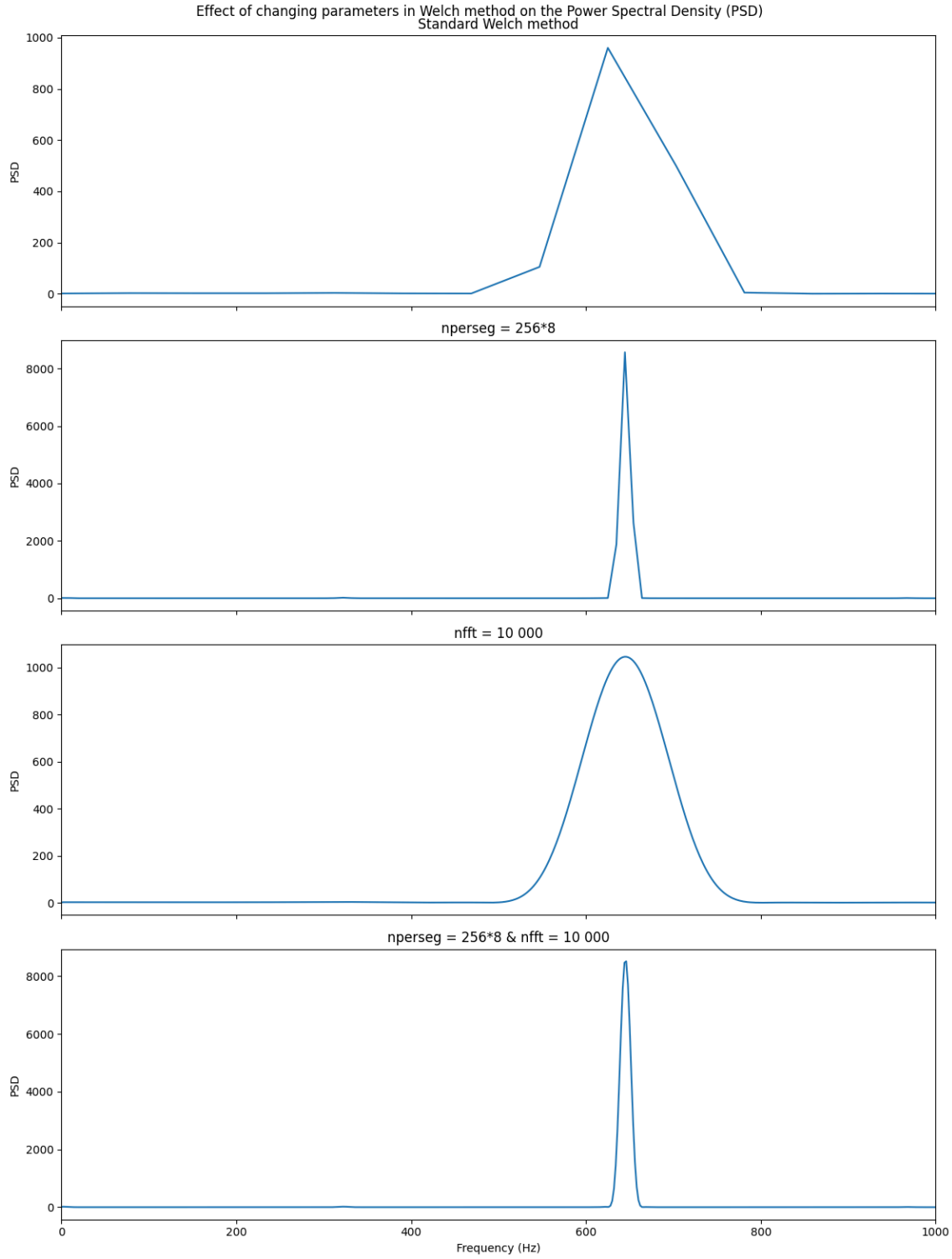
In [37]:
```
DEA.show_smoothed_rate()
```



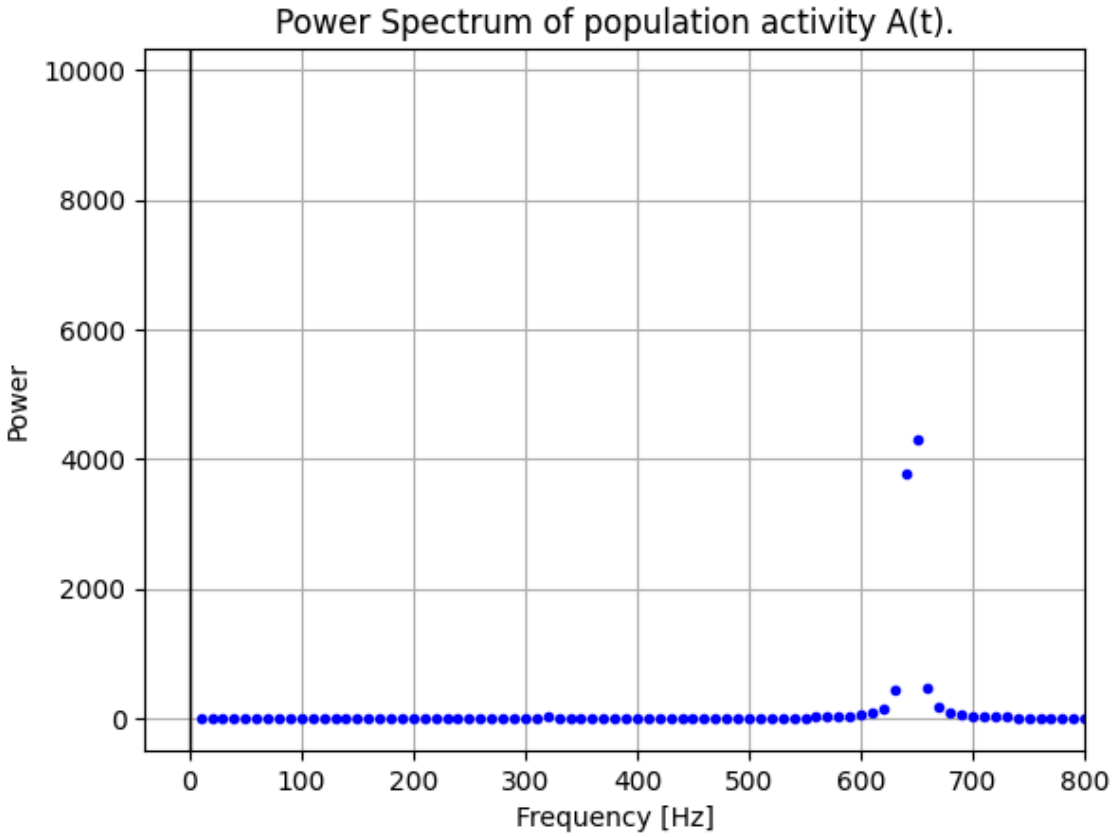
In [38]:
```
DEA.get_spectrum()
DEA.get_freqres()
```

```
Sampling frequency fs           = 20000.0 Hz
Integration time constant       = 50. us
------------------------------------------------------------
Welch frequency resolution                      = 78.125 Hz
Welch frequency resolution (nperseg=256*8)      = 9.765625 Hz
Welch frequency 'resolution' (nfft=10000)       = 2.0 Hz
Welch frequency 'resolution' (nperseg=256*8 & nfft=10000)       = 2.0 Hz
```

In [39]:
```
DEA.show_PSD()
```



In [40]:
```
DEA.get_population_activity(f_lim = 800)
```



3A Answer

The parameter g represents the inhibitory strength, while ν_{extern} denotes the firing frequency of the external neurons.

The network settles into a synchronous regular (SR) state when excitation dominates inhibition and synaptic time distributions are sharply peaked. In this state, neurons tend to synchronize almost fully into a few clusters, behaving as oscillators. The activity in this state appears to be very regular, with the group of excitatory and inhibitory neurons spiking in an alternating manner. The power spectrum also exhibits a sharp peak, occurring around 640 Hz, despite the mean firing rate being around 320 spikes/s. This is because two clusters, the inhibitory and excitatory, fire in an alternating pattern.

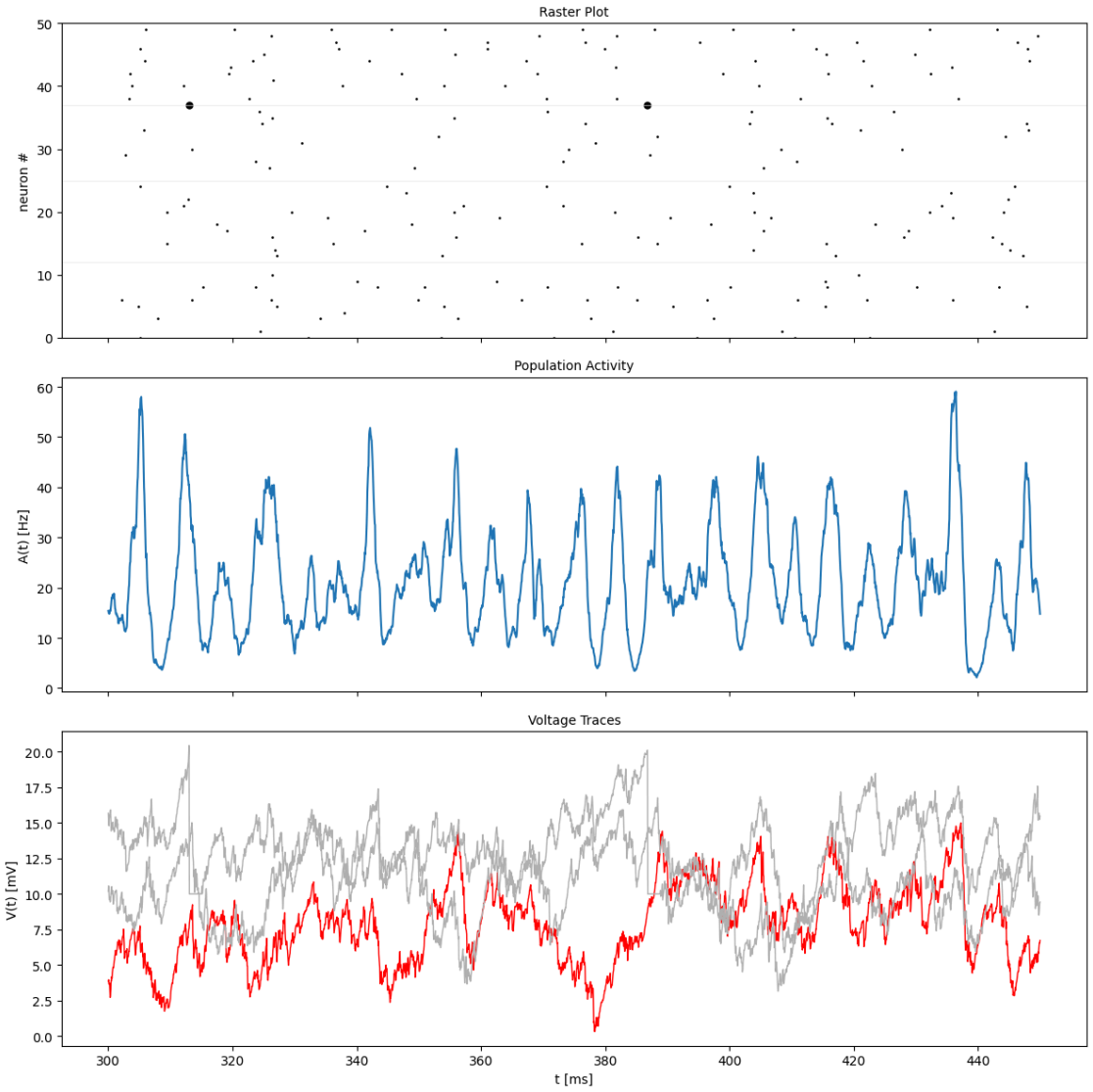
Overall, the network exhibits strong synchronization with regularly firing neurons when excitation dominates inhibition.

B : g = 6, $\nu_{extern}/\nu_{threshold} = 2$

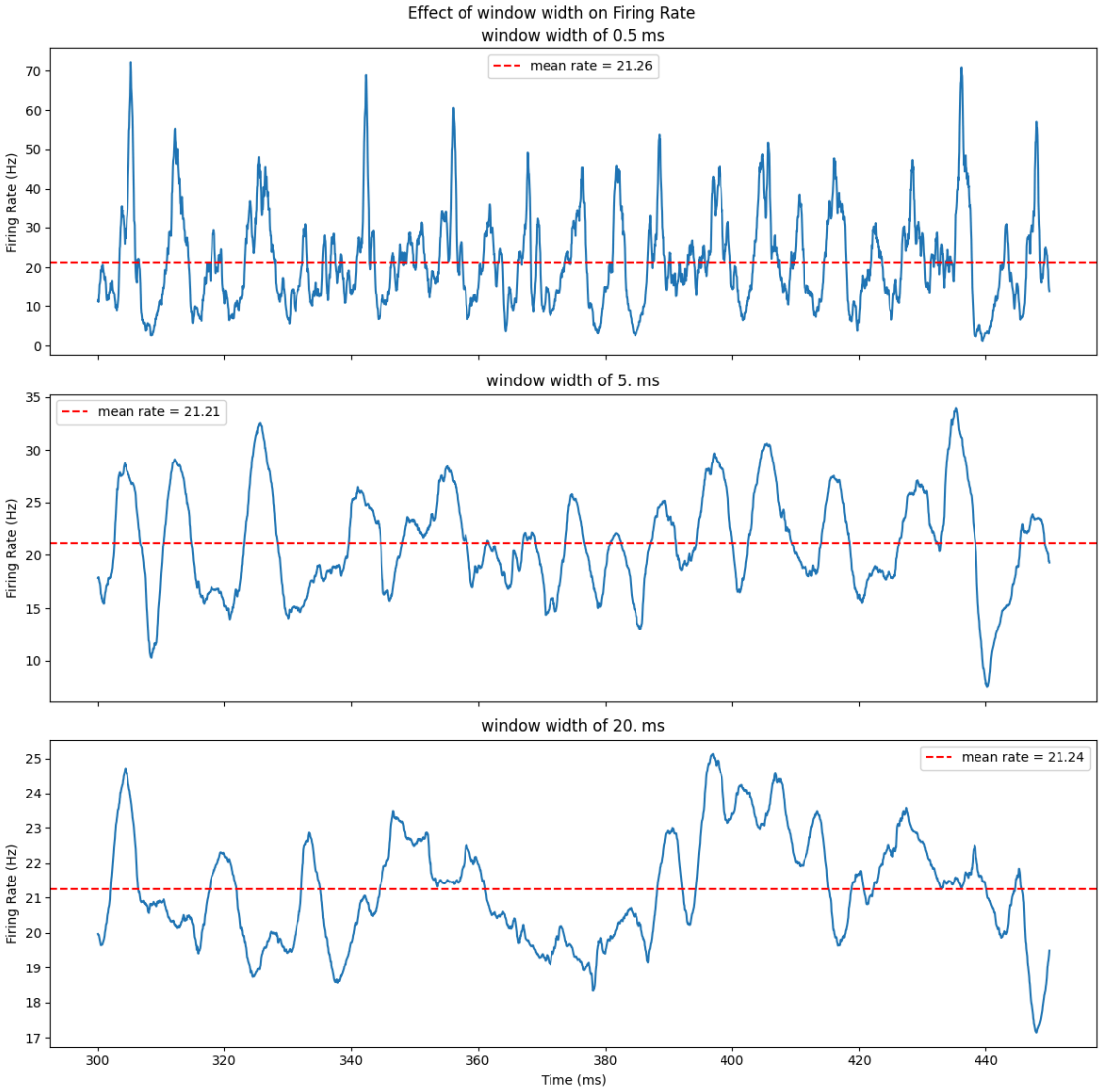
In [41]:
```
g = 6
nu_threshold = LIF_spiking_network.FIRING_THRESHOLD / (N_extern * w0 * LIF_s
nu_extern = 2*nu_threshold
```

```
DEB = DynamicsExplorer(g, nu_extern)
DEB.simulate()
```

In [42]:
```
DEB.show_simulation()
```



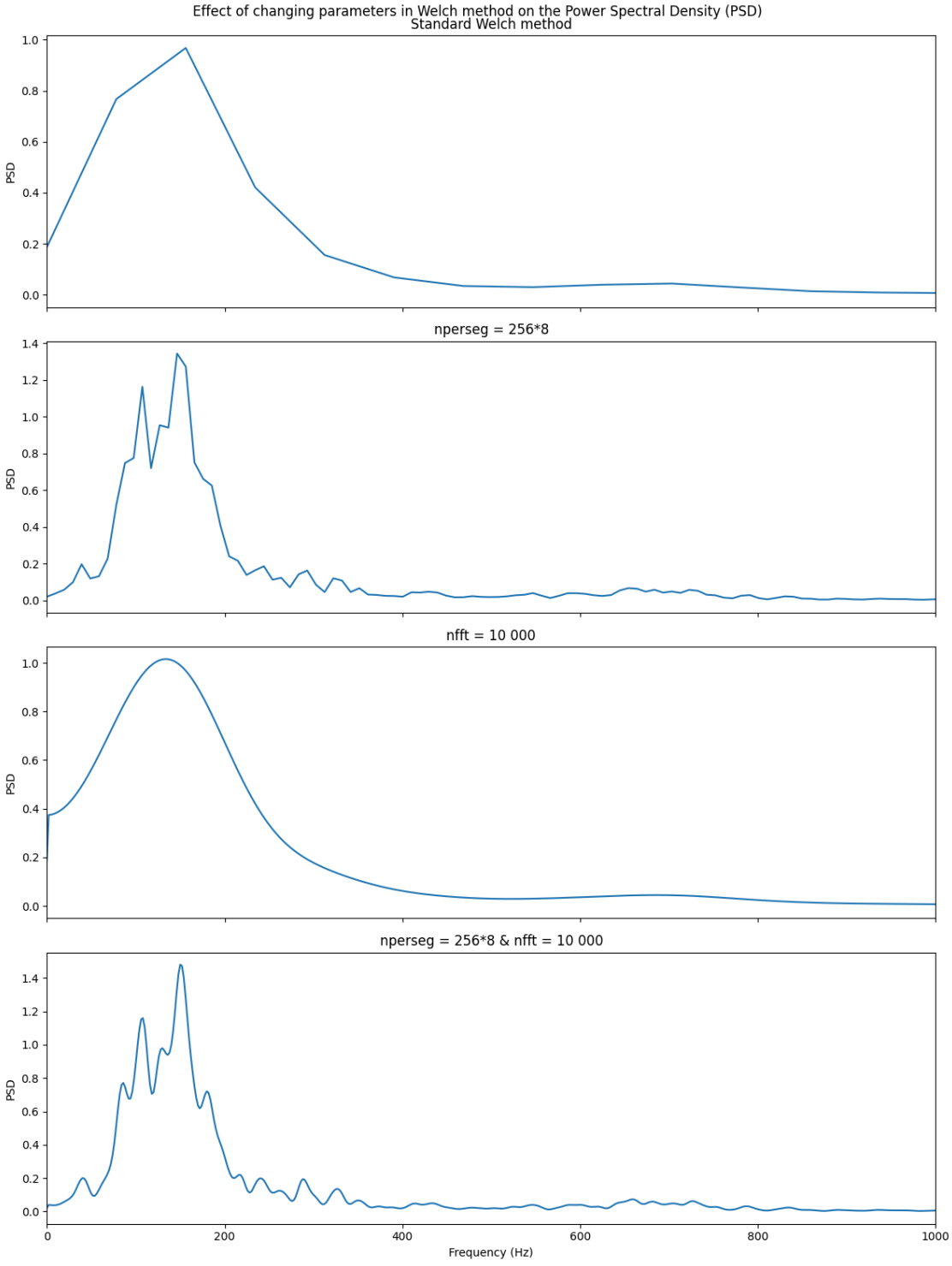
In [43]:
```
DEB.show_smoothed_rate()
```



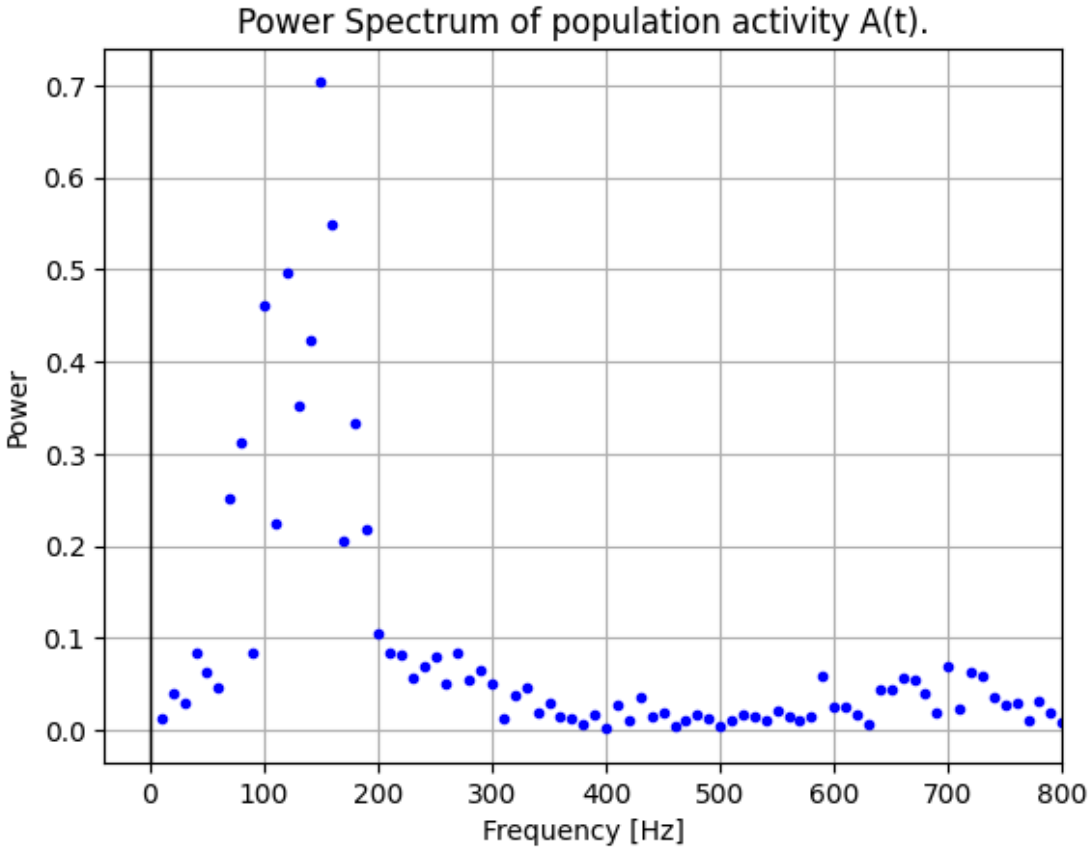
In [44]:
```
DEB.get_spectrum()
DEB.get_freqres()
```

```
Sampling frequency fs           = 20000.0 Hz
Integration time constant       = 50. us
-------------------------------------------------------------
Welch frequency resolution                          = 78.125 Hz
Welch frequency resolution (nperseg=256*8)          = 9.765625 Hz
Welch frequency 'resolution' (nfft=10000)           = 2.0 Hz
Welch frequency 'resolution' (nperseg=256*8 & nfft=10000)       = 2.0 Hz
```

In [45]:
```
DEB.show_PSD()
```



In [46]:
```
DEB.get_population_activity(f_lim = 800)
```



3B Answer

The results show a similar pattern to the parameters used in exercise 2, with a similarity observed among the spiking patterns of the different neurons. We thus conclude that this similarity probably depends on the value of g the most.

This simulation induces the neurons into an asynchronous irregular state. The oscillations of the power spectrum are strongly damped.

Spiking becomes irregular due to the increased importance of the inhibitory population (increasing g from 3 to 6).

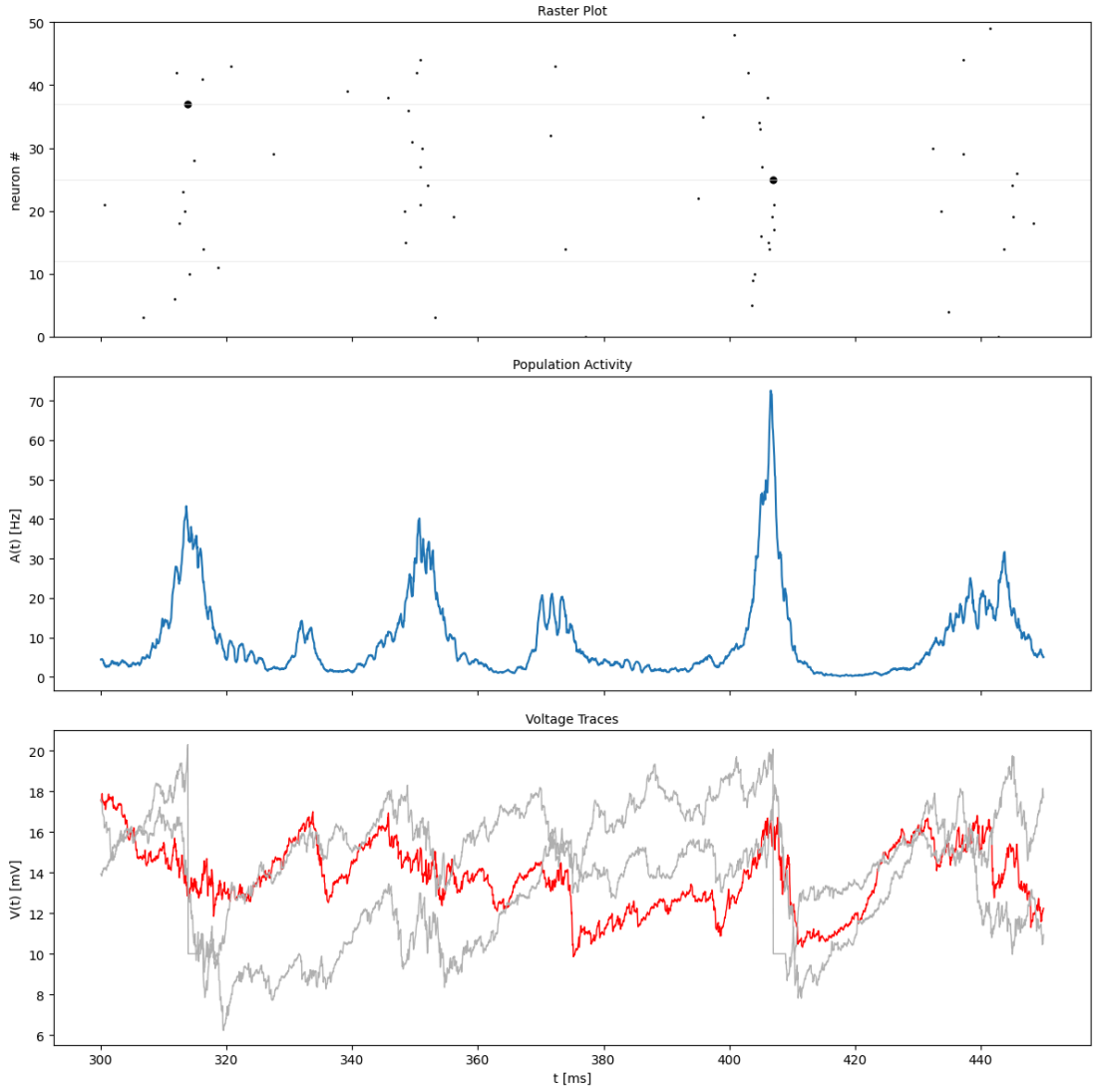
For this combination of parameters, the system settles into a state where global activity exhibits strongly damped oscillations, and neurons fire irregularly. Hence, inhibition dominates, and the external frequency is moderate. The mean rate is much lower than in the previous cases (around 23 Hz). The power spectrum is damped.

C : g = 4.5, $\nu_{extern}/\nu_{threshold} = 0.95$

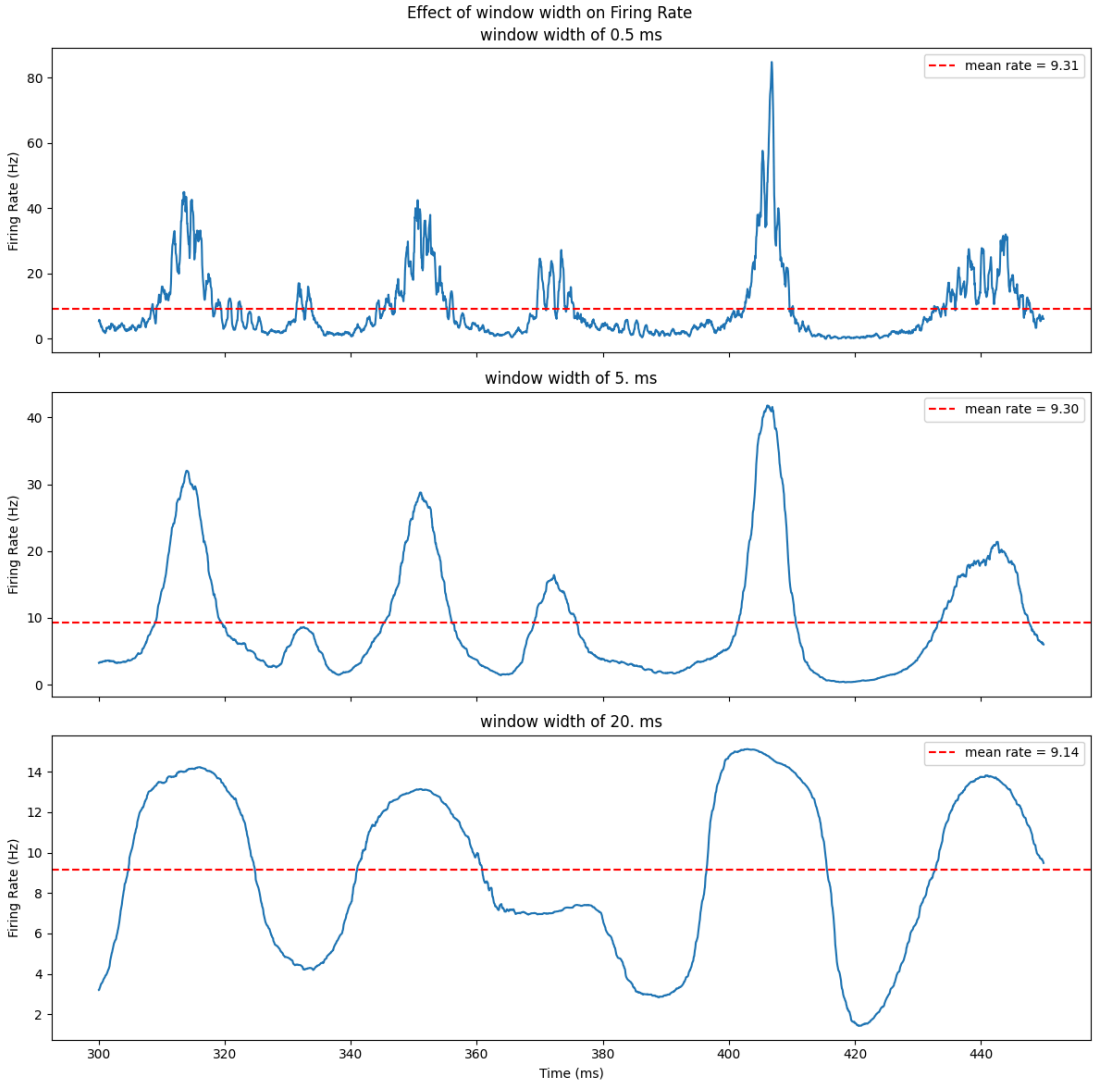
In [47]:
```
g = 4.5
nu_threshold = LIF_spiking_network.FIRING_THRESHOLD / (N_extern * w0 * LIF_
```

```
nu_extern = 0.95*nu_threshold

DEC = DynamicsExplorer(g, nu_extern)
DEC.simulate()
```

In [48]:
```
DEC.show_simulation()
```



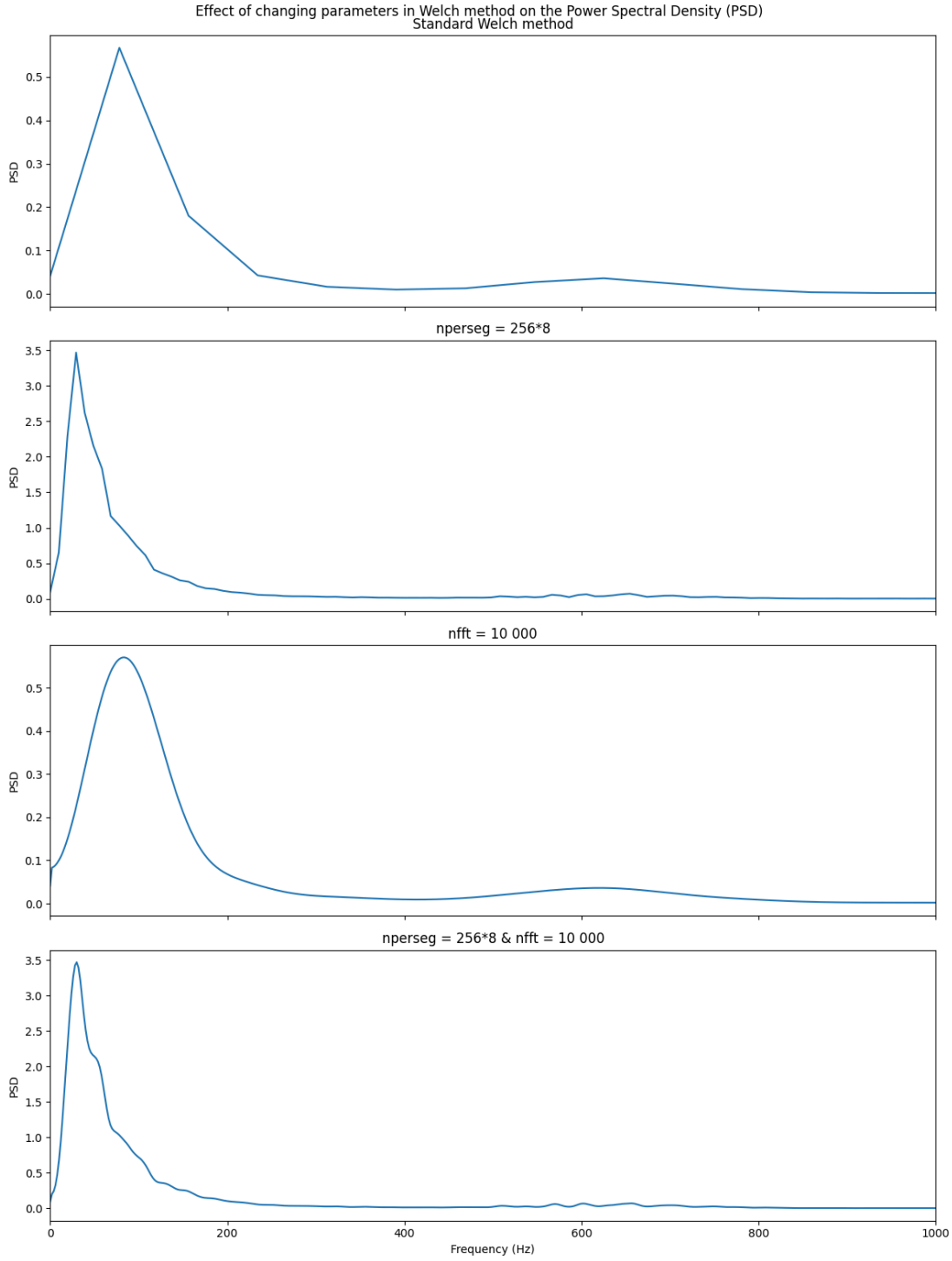
In [49]:
```
DEC.show_smoothed_rate()
```



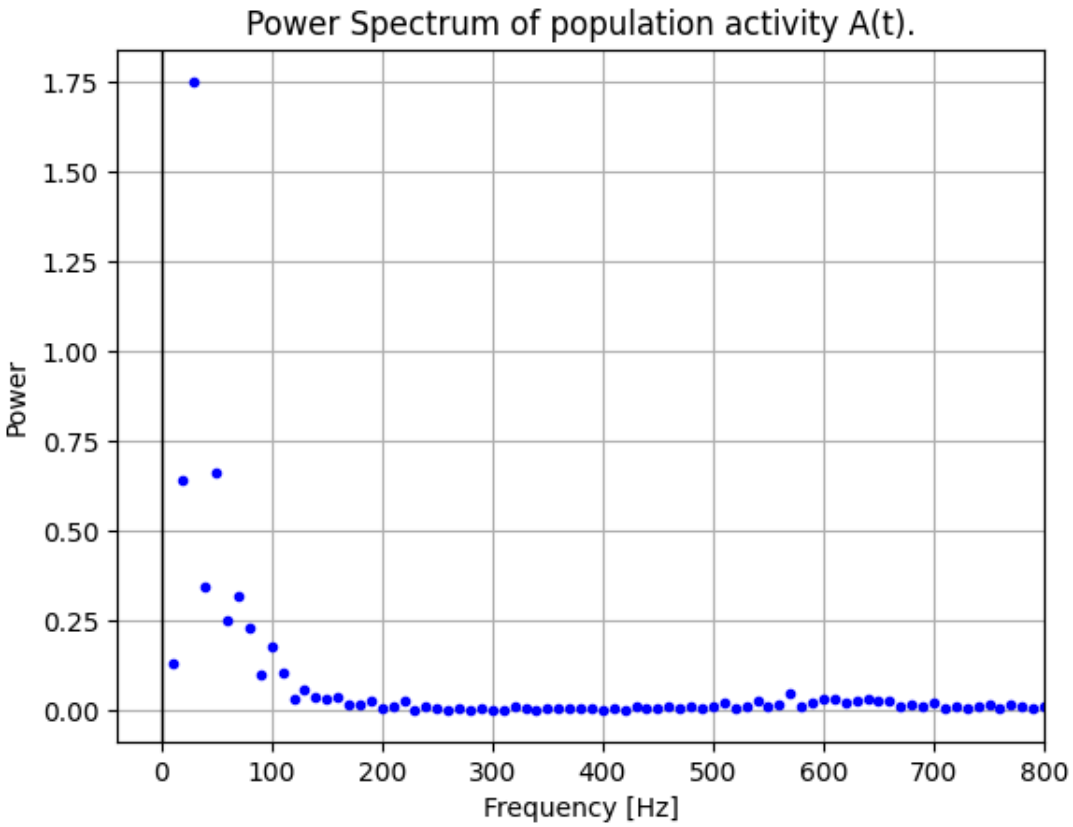
In [50]:
```
DEC.get_spectrum()
DEC.get_freqres()
```

```
Sampling frequency fs           = 20000.0 Hz
Integration time constant       = 50. us
------------------------------------------------------------
Welch frequency resolution                      = 78.125 Hz
Welch frequency resolution (nperseg=256*8)      = 9.765625 Hz
Welch frequency 'resolution' (nfft=10000)       = 2.0 Hz
Welch frequency 'resolution' (nperseg=256*8 & nfft=10000)       = 2.0 Hz
```

In [51]:
```
DEC.show_PSD()
```



In [52]:
```
DEC.get_population_activity(f_lim = 800)
```



3C Answer

The combination of a moderate g with a very low ν_{extern} results in a synchronous irregular state with slow oscillations. There are very low individual neuron firing rates when inhibition dominates and the ν_{extern} is smaller than but close to $\nu_{\text{threshold}}$. The mean spiking rate is even lower, around 7.5 Hz.

After spiking, the inhibitory neurons dominate again and all spiking stops. After some time, the influence of the external neurons will build up again.

In conclusion, there is synchronicity because there is spiking at the same time, but there are large intervals between the spiking events.