spatial-model-editor

Liam Keegan

May 05, 2021

GETTING STARTED

CHAPTER

ONE

GETTING STARTED

1.1 Installation

No installation required, just download and run the executable for your operating system:

-  linux
-  macOS
-  windows

Tip: You may have to give permission before your operating system will run the executable: `chmod +x spatial-model-editor` on linux, right-click open on macOS, "More info"->"Run anyway" on windows.

Note: on linux some additional system libraries are required that may not be installed by default. To install them:

- Ubuntu/Debian: `sudo apt-get install libxcb-xinerama0`
- Fedora/RHEL/CentOS: `sudo yum install xcb-util-image xcb-util-keysyms xcb-util-renderutil xcb-util-wm`

CHAPTER

TWO

IMPORTING A MODEL

To import an existing model from a SBML file, go to *File->Open SBML file* or type *Ctrl+O*.

There are also some built-in example models, to open one of these go to *File->Open example SBML file*

Fig. 1: One of the built-in example models: *very-simple-model.xml*

CHAPTER

THREE

IMPORTING GEOMETRY

After importing a model, the next step is to import an image of the compartments in the model. To do this, go to *Import->Geometry from image*, or choose one of the built-in example images from *Import->Example geometry image*.

The image should be segmented such that each compartment has a unique colour.

For each compartment in the model:

- click on the compartment name
- click on "Select compartment geometry…"
- then click on the desired part of the image.

Fig. 1: An example of importing a geometry image, and then assigning each compartment to a region in the image.

CHAPTER

FOUR

MESH GENERATION

Once the geometry has been created from an image, a mesh approximation to the geometry can be constructed for the solver.

The *Boundaries* tab shows the boundaries between compartments that have been automatically identified by the editor. The number of points used for each boundary can be altered here to refine or coarsen the boundary approximation.

Once the boundaries are chosen, the *Mesh* tab shows the generated triangular mesh for the currently selected compartment. The maximum triangle area can be altered here to refine or coarsen the mesh.

Fig. 1: An example of changing the points used in the compartment boundaries, and changing the coarseness of the mesh.

CHAPTER

FIVE

SPECIES PROPERTIES

In the Species tab, the species in each compartment are listed. Clicking on a species from this list displays the species concentration and other settings.

The initial concentration of a species can be a spatially uniform concentration, an analytic mathematical expression that may depend on the x and y location, or an image.

Fig. 1: An example of different ways to specify the initial spatial distribution of a species concentration.

CHAPTER

SIX

RUNNING A SIMULATION

To simulate the spatial model, click on the "Simulate" tab, specify the simulation time and desired interval between images, then click "Simulate".

The default simulation type is *DUNE*, which is a high-quality FEM solver. An alternative type is *Pixel*, which provides fast (but less reliable) results. The simulation type can be chosen by clicking on *Tools->Set simulation type*.

The resulting average species concentrations are plotted as a function of time. A snapshot of the spatial distribution is shown on the left, and the slider below the plot can be used to change the time that is being displayed. A 1d slice of the image as a function of time can also be generated.

Fig. 1: An example of a simple spatial simulation.

CHAPTER

SEVEN

PYTHON INTERFACE

A Python interface `sme` is available, which can be installed from PyPI with

```
pip install sme
```

Note: If your python version or platform is not supported, please submit a feature request

To get started, take a look at the *Example Notebooks*, or the *API Reference*.

Tip: There is an online colab notebook where you can try it out in your browser without installing anything on your computer.

CHAPTER

EIGHT

EXAMPLE NOTEBOOKS

Interactive online version

8.1 Getting started

8.1.1 Install and import `sme`

[1]:
```
!pip install -q sme
import sme
from matplotlib import pyplot as plt
print("sme version:", sme.__version__)
```

```
sme version: 1.1.0
```

8.1.2 Importing a model

- to load an existing model: `sme.open_sbml_file('model_filename.xml')`
- to load a built-in example model: `sme.open_example_model()`

[2]:
```
my_model = sme.open_example_model()
```

8.1.3 Getting help

- to see the type of an object: `type(object)`
- to print a one line description of an object: `repr(object)`
- to print a multi-line description of an object: `print(object)`
- to get help on an object, its methods and properties: `help(object)`

[3]:
```
type(my_model)
```

[3]:
```
sme.Model
```

[4]:
```
repr(my_model)
```

[4]:
```
"<sme.Model named 'Very Simple Model'>"
```

[5]:
```
print(my_model)
```

```
<sme.Model>
  - name: 'Very Simple Model'
  - compartments:
     - Outside
     - Cell
     - Nucleus
  - membranes:
     - Outside <-> Cell
     - Cell <-> Nucleus
```

[6]:
```
help(my_model)
```

```
Help on Model in module sme object:

class Model(pybind11_builtins.pybind11_object)
 |  the spatial model
 |
 |  Method resolution order:
 |      Model
 |      pybind11_builtins.pybind11_object
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      __init__(self: sme.Model, filename: str) -> None
 |
 |  __repr__(...)
 |      __repr__(self: sme.Model) -> str
 |
 |  __str__(...)
 |      __str__(self: sme.Model) -> str
 |
 |  export_sbml_file(...)
 |      export_sbml_file(self: sme.Model, filename: str) -> None
 |
 |
 |      exports the model as a spatial SBML file
 |
 |      Args:
 |          filename (str): the name of the file to create
 |
 |  export_sme_file(...)
 |      export_sme_file(self: sme.Model, filename: str) -> None
 |
 |
 |      exports the model as a sme file
 |
 |      Args:
 |          filename (str): the name of the file to create
 |
 |  import_geometry_from_image(...)
 |      import_geometry_from_image(self: sme.Model, filename: str) -> None
 |
 |
 |      sets the geometry of each compartment to the corresponding pixels in the␣
↪supplied geometry image
```

(continues on next page)

(continued from previous page)

```
 |
 |      Note:
 |          Currently this function assumes that the compartments maintain the same␣
↪colour
 |          as they had with the previous geometry image. If the new image does not␣
↪contain
 |          pixels of each of these colours, the new model geometry will not be valid.
 |          The size of a pixel (in physical units) is also unchanged by this␣
↪function.
 |
 |      Args:
 |          filename (str): the name of the geometry image to import
 |
 |  simulate(...)
 |      simulate(*args, **kwargs)
 |      Overloaded function.
 |
 |      1. simulate(self: sme.Model, simulation_time: float, image_interval: float,␣
↪timeout_seconds: int = 86400, throw_on_timeout: bool = True, simulator_type: sme.
↪SimulatorType = <SimulatorType.Pixel: 1>, continue_existing_simulation: bool =␣
↪False) -> std::vector<sme::SimulationResult, std::allocator<sme::SimulationResult> >
 |
 |
 |                  returns the results of the simulation.
 |
 |                  Args:
 |                      simulation_time (float): The length of the simulation in model␣
↪units of time, e.g. `5.5`
 |                      image_interval (float): The interval between images in model␣
↪units of time, e.g. `1.1`
 |                      timeout_seconds (int): The maximum time in seconds that the␣
↪simulation can run for. Default value: 86400 = 1 day.
 |                      throw_on_timeout (bool): Whether to throw an exception on␣
↪simulation timeout. Default value: `true`.
 |                      simulator_type (sme.SimulatorType): The simulator to use: `sme.
↪SimulatorType.DUNE` or `sme.SimulatorType.Pixel`. Default value: Pixel.
 |                      continue_existing_simulation (bool): Whether to continue the␣
↪existing simulation, or start a new simulation. Default value: `false`, i.e. any␣
↪existing simulation results are discarded before doing the simulation.
 |
 |                  Returns:
 |                      SimulationResultList: the results of the simulation
 |
 |                  Raises:
 |                      RuntimeError: if the simulation times out or fails
 |
 |
 |      2. simulate(self: sme.Model, simulation_times: str, image_intervals: str,␣
↪timeout_seconds: int = 86400, throw_on_timeout: bool = True, simulator_type: sme.
↪SimulatorType = <SimulatorType.Pixel: 1>, continue_existing_simulation: bool =␣
↪False) -> std::vector<sme::SimulationResult, std::allocator<sme::SimulationResult> >
 |
 |
 |                  returns the results of the simulation.
 |
 |                  Args:
 |                      simulation_times (str): The length(s) of the simulation in␣
↪model units of time as a comma-delimited list, e.g. `"5"`, or `"10;100;20"`
```

(continues on next page)

(continued from previous page)

```
 |                      image_intervals (str): The interval(s) between images in model␣
↪units of time as a comma-delimited list, e.g. `"1"`, or `"2;10;0.5"`
 |                      timeout_seconds (int): The maximum time in seconds that the␣
↪simulation can run for. Default value: 86400 = 1 day.
 |                      throw_on_timeout (bool): Whether to throw an exception on␣
↪simulation timeout. Default value: `true`.
 |                      simulator_type (sme.SimulatorType): The simulator to use: `sme.
↪SimulatorType.DUNE` or `sme.SimulatorType.Pixel`. Default value: Pixel.
 |                      continue_existing_simulation (bool): Whether to continue the␣
↪existing simulation, or start a new simulation. Default value: `false`, i.e. any␣
↪existing simulation results are discarded before doing the simulation.
 |
 |                  Returns:
 |                      SimulationResultList: the results of the simulation
 |
 |                  Raises:
 |                      RuntimeError: if the simulation times out or fails
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  compartment_image
 |      list of list of list of int: an image of the compartments in this model
 |
 |  compartments
 |      CompartmentList: the compartments in this model
 |
 |      a list of :class:`Compartment` that can be iterated over,
 |      or indexed into by name or position in the list.
 |
 |      Examples:
 |          the list of compartments can be iterated over:
 |
 |          >>> import sme
 |          >>> model = sme.open_example_model()
 |          >>> for compartment in model.compartments:
 |          ...     print(compartment.name)
 |          Outside
 |          Cell
 |          Nucleus
 |
 |          or a compartment can be found using its name:
 |
 |          >>> cell = model.compartments["Cell"]
 |          >>> print(cell.name)
 |          Cell
 |
 |          or indexed by its position in the list:
 |
 |          >>> last_compartment = model.compartments[-1]
 |          >>> print(last_compartment.name)
 |          Nucleus
 |
 |  membranes
 |      MembraneList: the membranes in this model
 |
 |      a list of :class:`Membrane` that can be iterated over,
```

(continues on next page)

(continued from previous page)

```
|      or indexed into by name or position in the list.
|
|      Examples:
|          the list of membranes can be iterated over:
|
|          >>> import sme
|          >>> model = sme.open_example_model()
|          >>> for membrane in model.membranes:
|          ...     print(membrane.name)
|          Outside <-> Cell
|          Cell <-> Nucleus
|
|          or a membrane can be found using its name:
|
|          >>> outer = model.membranes["Outside <-> Cell"]
|          >>> print(outer.name)
|          Outside <-> Cell
|
|          or indexed by its position in the list:
|
|          >>> last_membrane = model.membranes[-1]
|          >>> print(last_membrane.name)
|          Cell <-> Nucleus
|
|  name
|      str: the name of this model
|
|  parameters
|      ParameterList: the parameters in this model
|
|      a list of :class:`Parameter` that can be iterated over,
|      or indexed into by name or position in the list.
|
|      Examples:
|          the list of parameters can be iterated over:
|
|          >>> import sme
|          >>> model = sme.open_example_model()
|          >>> for parameter in model.parameters:
|          ...     print(parameter.name)
|          param
|
|          or a parameter can be found using its name:
|
|          >>> p = model.parameters["param"]
|          >>> print(p.name)
|          param
|
|          or indexed by its position in the list:
|
|          >>> last_param = model.parameters[-1]
|          >>> print(last_param.name)
|          param
|
|  ----------------------------------------------------------------------
|  Static methods inherited from pybind11_builtins.pybind11_object:
|
```

(continues on next page)

(continued from previous page)

```
|  __new__(*args, **kwargs) from pybind11_builtins.pybind11_type
|      Create and return a new object.  See help(type) for accurate signature.
```

8.1.4 Viewing model contents

- the compartments in a model can be accessed as a list: `model.compartments`
- the list can be iterated over, or an item looked up by index or name
- other lists of objects, such as species in a compartment, or parameters in a reaction, behave in the same way

Iterating over compartments

[7]:
```
for compartment in my_model.compartments:
    print(repr(compartment))
```

```
<sme.Compartment named 'Outside'>
<sme.Compartment named 'Cell'>
<sme.Compartment named 'Nucleus'>
```

Get compartment by name

[8]:
```
cell_compartment = my_model.compartments["Cell"]
print(repr(cell_compartment))
```

```
<sme.Compartment named 'Cell'>
```

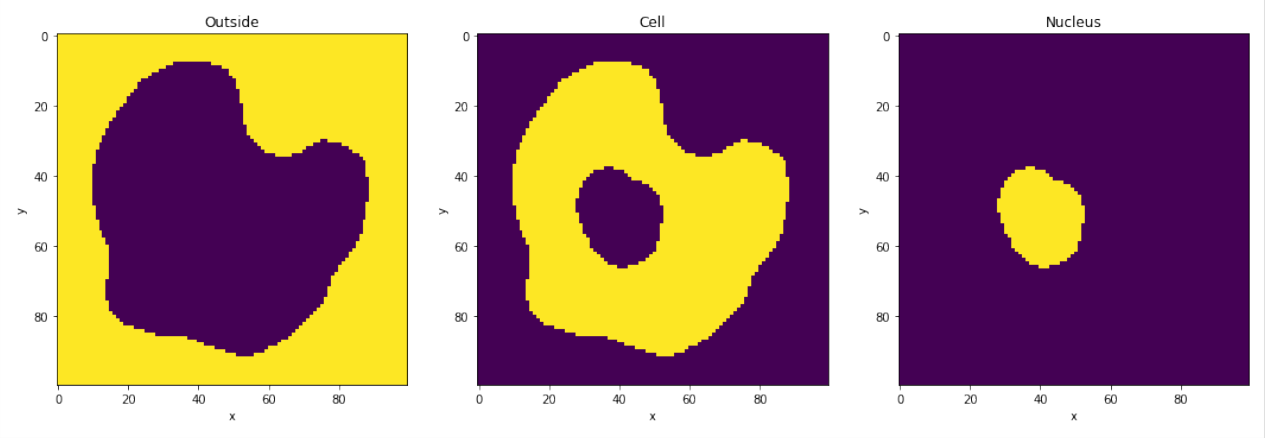
Get compartment by list index

[9]:
```
last_compartment = my_model.compartments[-1]
print(repr(last_compartment))
```

```
<sme.Compartment named 'Nucleus'>
```

Display geometry of compartments

[10]:
```
fig, axs = plt.subplots(nrows=1, ncols=len(my_model.compartments), figsize=(18, 12))
for (ax, compartment) in zip(axs, my_model.compartments):
    ax.imshow(compartment.geometry_mask, interpolation='none')
    ax.set_title(f"{compartment.name}")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
plt.show()
```



Display parameter names and values

[11]:
```
my_reac = my_model.compartments['Nucleus'].reactions['A to B conversion']
print(my_reac)
for param in my_reac.parameters:
    print(param)
```

```
<sme.Reaction>
  - name: 'A to B conversion'

<sme.ReactionParameter>
  - name: 'k1'
  - value: '0.3'
```

8.1.5 Editing model contents

- model parameters and object names can be changed by assigning new values to them

[12]:
```
print("name:", my_model.name)
```

```
name: Very Simple Model
```

[13]:
```
my_model.name = 'New model name!'
```

[14]:
```
print("name:", my_model.name)
```

```
name: New model name!
```

[15]:
```
print("k1 value:", my_model.compartments['Nucleus'].reactions['A to B conversion'].
↪parameters['k1'].value)
```

```
k1 value: 0.3
```

[16]:
```
my_model.compartments['Nucleus'].reactions['A to B conversion'].parameters['k1'].
↪value = 0.72
```

[17]:
```
print("k1 value:", my_model.compartments['Nucleus'].reactions['A to B conversion'].
↪parameters['k1'].value)
```

```
k1 value: 0.72
```

8.1.6 Exporting a model

- to export the model to a SBML file: `model.export_sbml_file('model_filename.xml')`

[18]:
```
my_model.export_sbml_file('model.xml')
```

Interactive online version

8.2 Simulating

8.2.1 Open an example model

[1]:
```
!pip install -q sme
import sme
from matplotlib import pyplot as plt
import numpy as np
```

[2]:
```
my_model = sme.open_example_model()
```

8.2.2 Running a simulation

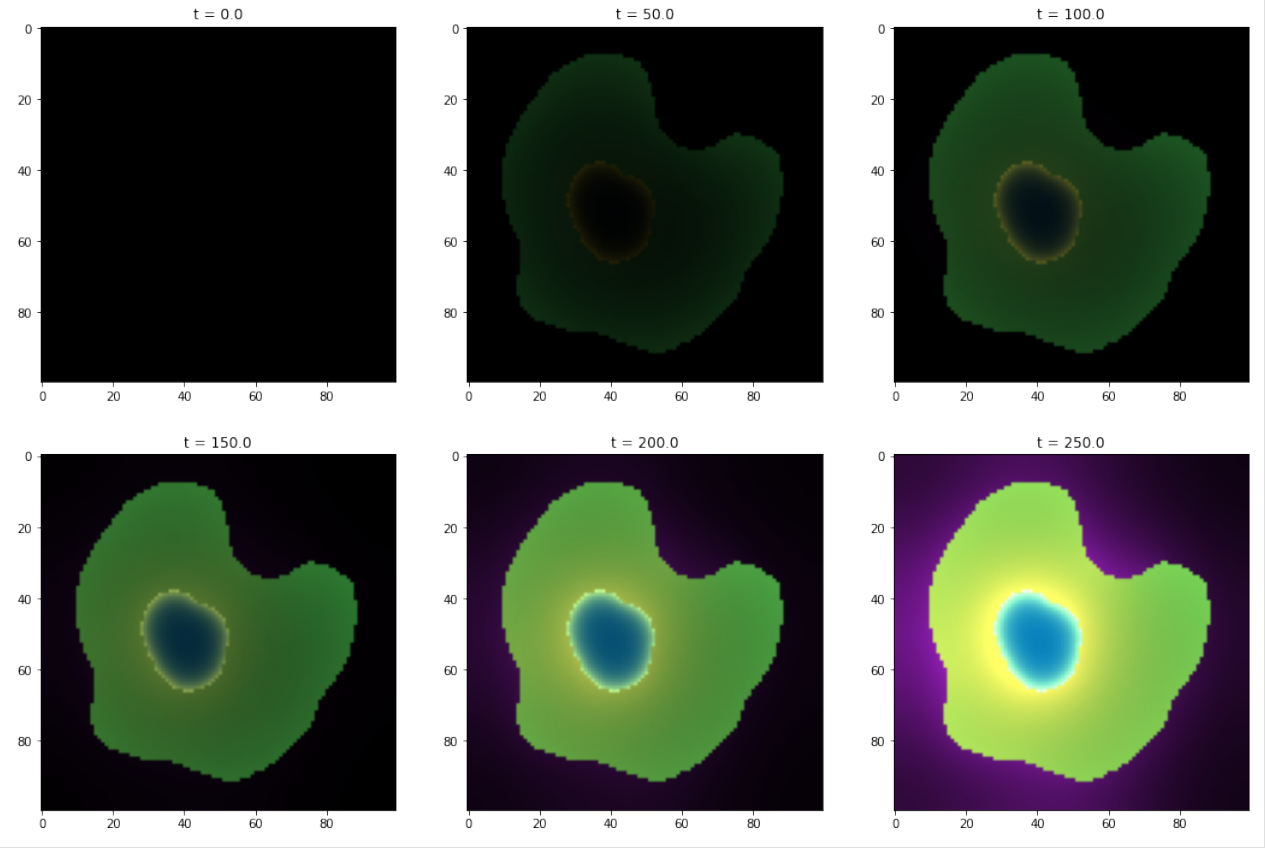
- models can be simulated by specifying the total simulation time, and the interval between images
- the simulation returns a list of `SimulationResult` objects, each of which contains
- `time_point`: the time point
- `concentration_image`: an image of the species concentrations at this time point
- `species_concentration`: a dict of the concentrations for each species at this time point

[3]:
```
sim_results = my_model.simulate(simulation_time=250.0, image_interval=50.0)
type(sim_results[0])
print(sim_results[0])
```

```
<sme.SimulationResult>
  - timepoint: 0
  - number of species: 5
```

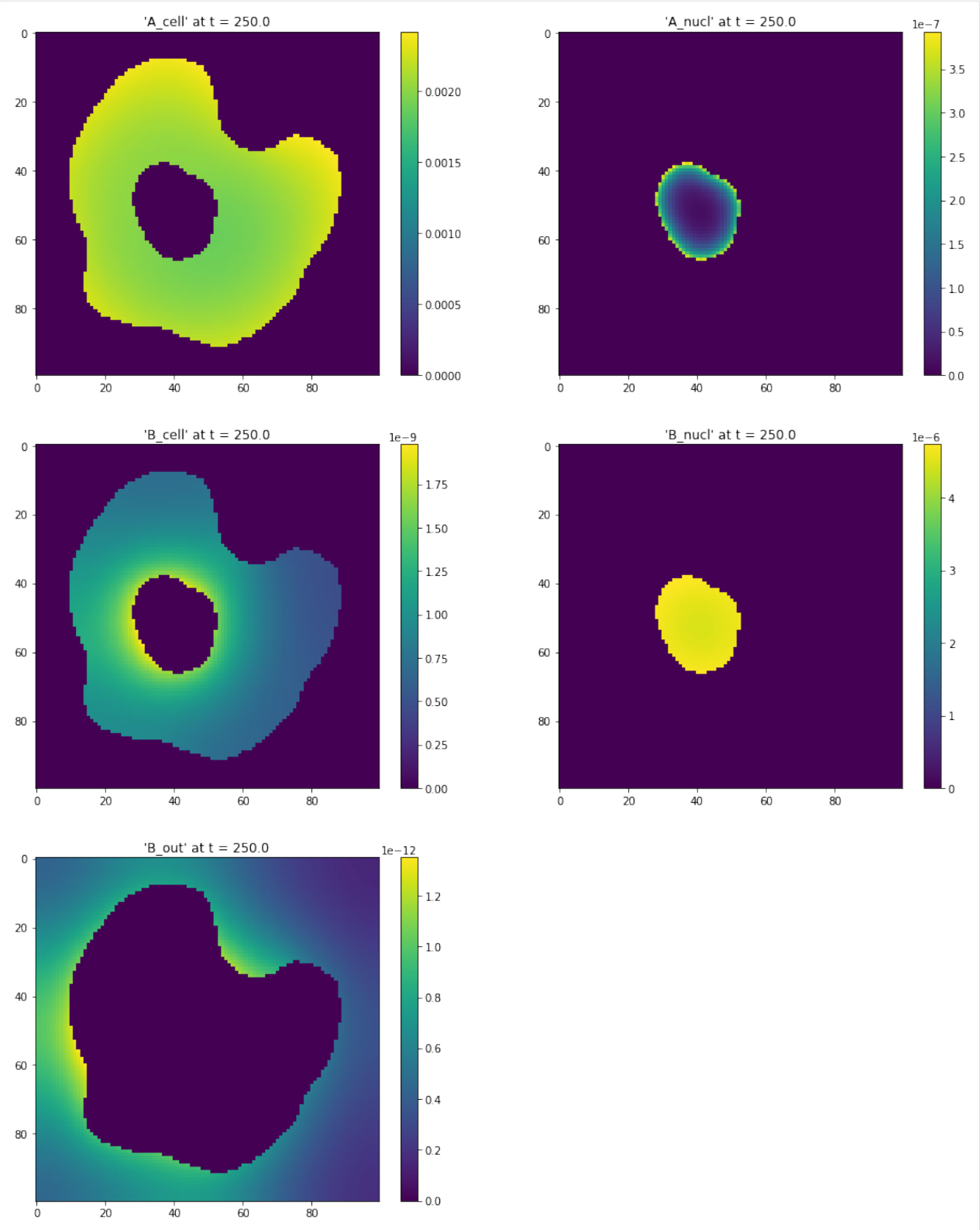
Display images from simulation results

[4]:
```
fig, axs = plt.subplots(nrows=2, ncols=len(sim_results)//2, figsize=(18, 12))
for (ax, res) in zip(fig.axes, sim_results):
    ax.imshow(res.concentration_image)
    ax.set_title(f"t = {res.time_point}")
plt.show()
```



Plot concentrations from simulation results

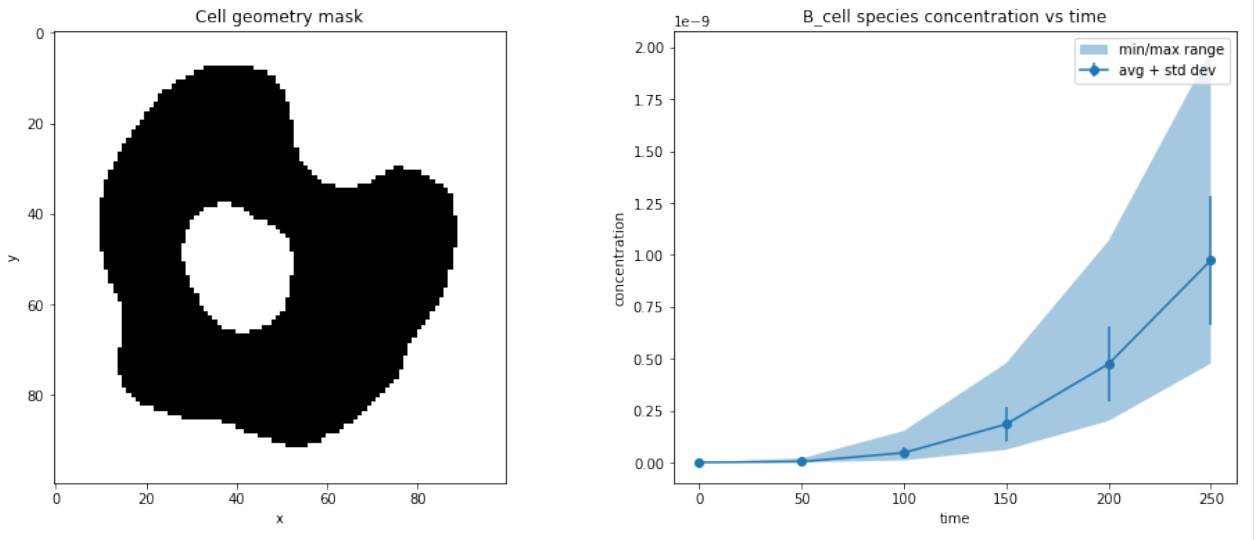
[5]:
```
result = sim_results[5]
fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(16, 20))
fig.delaxes(axs[2,1])
for (ax, (species, concentration)) in zip(fig.axes, result.species_concentration.
↪items()):
    im = ax.imshow(concentration)
    ax.set_title(f"'{species}' at t = {result.time_point}")
    fig.colorbar(im, ax=ax)
plt.show()
```


'A_cell' at t = 250.0
'A_nucl' at t = 250.0
1e−7
'B_cell' at t = 250.0
1e−9
'B_nucl' at t = 250.0
1e−6
'B_out' at t = 250.0
1e−12

Plot average/min/max species concentrations

To get the average (or minimum/maxiumum/etc) concentration of a species in a compartment, we first use the compartment `geometry_mask` to only include the pixels that lie inside the compartment.

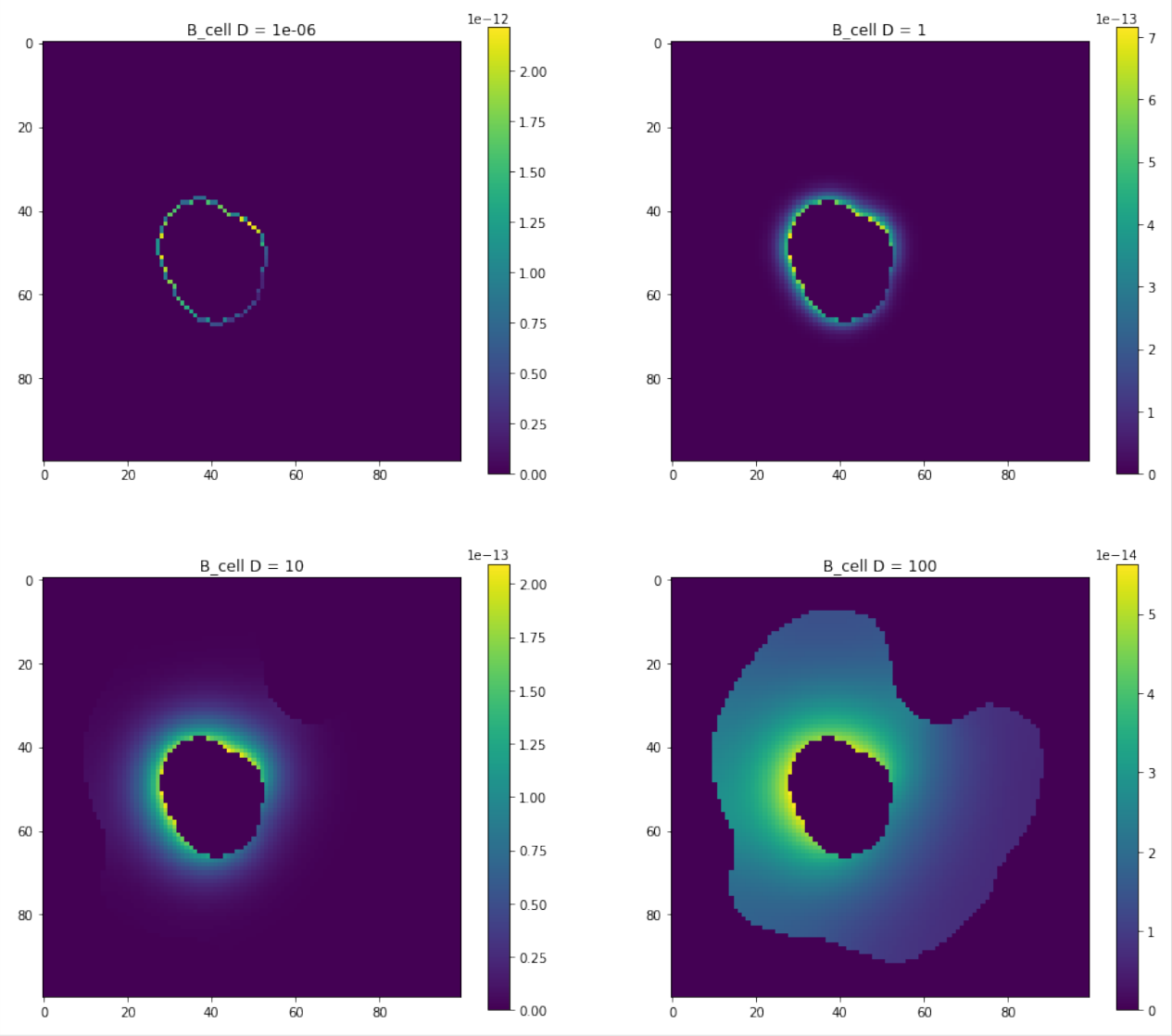
[6]:
```
fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, figsize=(16, 6))

# get mask of compartment pixels
mask = np.asarray(my_model.compartments['Cell'].geometry_mask)
ax0.imshow(mask, interpolation='none', cmap='Greys')
ax0.set_title("Cell geometry mask")
ax0.set_xlabel("x")
ax0.set_ylabel("y")

# apply mask to results to get a flat array of all concentrations
# inside the compartment at each time point
times = []
concs = []
for result in sim_results:
    times.append(result.time_point)
    concs.append(np.asarray(result.species_concentration['B_cell'])[mask].flatten())

# calculate avg, min, max and plot
avg_conc = [np.mean(x) for x in concs]
std_conc = [np.std(x) for x in concs]
min_conc = [np.min(x) for x in concs]
max_conc = [np.max(x) for x in concs]
ax1.set_title("B_cell species concentration vs time")
ax1.set_xlabel("time")
ax1.set_ylabel("concentration")
ax1.errorbar(times, avg_conc, std_conc, label="avg + std dev", marker='o')
ax1.fill_between(times, min_conc, max_conc, label="min/max range", alpha=0.4)
ax1.legend()

plt.show()
```



8.2.3 Diffusion constant example

Here we repeat a simulation four times, each time with a different value for the diffusion constant of species `B_cell`, and plot the resulting concentration of this species at `t=15`.

[7]:
```
diffconsts = [1e-6, 1, 10, 100]
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(16, 14))
for ax, diffconst in zip(fig.axes, diffconsts):
    m = sme.open_example_model()
    m.compartments['Cell'].species['B_cell'].diffusion_constant = diffconst
    results = m.simulate(simulation_time=15.0, image_interval=15.0)
    im = ax.imshow(results[1].species_concentration['B_cell'])
    ax.set_title(f"B_cell D = {diffconst}")
    fig.colorbar(im, ax=ax)
plt.show()
```



CHAPTER

NINE

API REFERENCE

sme	Spatial Model Editor Python interface

9.1 sme

Spatial Model Editor Python interface

Python bindings to a subset of the functionality available in the full GUI Spatial Model Editor

https://spatial-model-editor.readthedocs.io/

Functions

open_example_model()	opens a built in example spatial model
open_file(filename)	opens a sme or SBML file containing a spatial model
open_sbml_file(filename)	opens an SBML file containing a spatial model

9.1.1 sme.open_example_model

sme.**open_example_model**() → sme::Model
opens a built in example spatial model

Returns the example spatial model

Return type *Model*

9.1.2 sme.open_file

sme.**open_file**(*filename: str*) → sme::Model
opens a sme or SBML file containing a spatial model

Parameters **filename** (*str*) – the sme or SBML file to open

Returns the spatial model

Return type *Model*

9.1.3 sme.open_sbml_file

sme.**open_sbml_file**(*filename: str*) → sme::Model
opens an SBML file containing a spatial model

Parameters **filename** (*str*) – the SBML file to open

Returns the spatial model

Return type *Model*

Classes

Compartment	a compartment where species live
CompartmentList	CompartmentList: a list of *Compartment*
Membrane	a membrane where two compartments meet
MembraneList	MembraneList: a list of *Membrane*
Model	the spatial model
Parameter	a parameter of the model
ParameterList	ParameterList: a list of *Parameter*
Reaction	a reaction between species
ReactionList	ReactionList: a list of *Reaction*
ReactionParameter	a parameter of a reaction
ReactionParameterList	ReactionParameterList: a list of *ReactionParameter*
SimulationResult	results at a single timepoint of a simulation
SimulationResultList	SimulationResultList: a list of *SimulationResult*
SimulatorType	Members:
Species	a species that lives in a compartment
SpeciesList	SpeciesList: a list of *Species*

9.1.4 sme.Compartment

class sme.**Compartment**
a compartment where species live

Methods

__init__(*args, **kwargs)	Initialize self.

sme.Compartment.__init__

Compartment.**__init__**(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

geometry_mask	2d pixel mask of the compartment geometry
name	the name of this compartment
reactions	the reactions in this compartment
species	the species in this compartment

sme.Compartment.geometry_mask

property Compartment.**geometry_mask**
2d pixel mask of the compartment geometry

The mask is a list of list of bool, where `geometry_mask[y][x] = True` if the pixel at point (x,y) is part of this compartment

Type list of list of bool

sme.Compartment.name

property Compartment.**name**
the name of this compartment

Type str

sme.Compartment.reactions

property Compartment.**reactions**
the reactions in this compartment

Type *ReactionList*

sme.Compartment.species

property Compartment.**species**
the species in this compartment

Type *SpeciesList*

9.1.5 sme.CompartmentList

class sme.**CompartmentList**

CompartmentList: a list of *Compartment*

the list can be iterated over, or an element can be looked up by its index or name

Methods

__init__(self)	

sme.CompartmentList.__init__

CompartmentList.**__init__**(*self:* sme.CompartmentList) → None

9.1.6 sme.Membrane

class sme.**Membrane**

a membrane where two compartments meet

Methods

__init__(*args, **kwargs)	Initialize self.

sme.Membrane.__init__

Membrane.**__init__**(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Attributes

name	the name of this membrane
reactions	the reactions in this membrane

sme.Membrane.name

property Membrane.**name**
the name of this membrane

Type str

sme.Membrane.reactions

property Membrane.**reactions**
the reactions in this membrane

Type *ReactionList*

9.1.7 sme.MembraneList

class sme.**MembraneList**
MembraneList: a list of *Membrane*

the list can be iterated over, or an element can be looked up by its index or name

Methods

__init__(self)

sme.MembraneList.__init__

MembraneList.**__init__**(*self:* sme.MembraneList) → None

9.1.8 sme.Model

class sme.**Model**
the spatial model

Methods

__init__(self, filename)

export_sbml_file(self, filename)	exports the model as a spatial SBML file
export_sme_file(self, filename)	exports the model as a sme file
import_geometry_from_image(self, filename)	sets the geometry of each compartment to the corresponding pixels in the supplied geometry image
simulate(*args, **kwargs)	Overloaded function.

sme.Model.__init__

`Model.__init__` (*self:* sme.Model, *filename: str*) → None

sme.Model.export_sbml_file

`Model.export_sbml_file` (*self:* sme.Model, *filename: str*) → None
exports the model as a spatial SBML file

Parameters **filename** (*str*) – the name of the file to create

sme.Model.export_sme_file

`Model.export_sme_file` (*self:* sme.Model, *filename: str*) → None
exports the model as a sme file

Parameters **filename** (*str*) – the name of the file to create

sme.Model.import_geometry_from_image

`Model.import_geometry_from_image` (*self:* sme.Model, *filename: str*) → None
sets the geometry of each compartment to the corresponding pixels in the supplied geometry image

---

Note: Currently this function assumes that the compartments maintain the same colour as they had with the previous geometry image. If the new image does not contain pixels of each of these colours, the new model geometry will not be valid. The size of a pixel (in physical units) is also unchanged by this function.

Parameters **filename** (*str*) – the name of the geometry image to import

sme.Model.simulate

`Model.simulate` (**args*, ***kwargs*)
Overloaded function.

1. simulate(self: sme.Model, simulation_time: float, image_interval: float, timeout_seconds: int = 86400, throw_on_timeout: bool = True, simulator_type: sme.SimulatorType = <SimulatorType.Pixel: 1>, continue_existing_simulation: bool = False) -> std::vector<sme::SimulationResult, std::allocator<sme::SimulationResult> >

   returns the results of the simulation.

   Args: simulation_time (float): The length of the simulation in model units of time, e.g. *5.5* image_interval (float): The interval between images in model units of time, e.g. *1.1* timeout_seconds (int): The maximum time in seconds that the simulation can run for. Default value: 86400 = 1 day. throw_on_timeout (bool): Whether to throw an exception on simulation timeout. Default value: *true*. simulator_type (sme.SimulatorType): The simulator to use: *sme.SimulatorType.DUNE* or *sme.SimulatorType.Pixel*. Default value: Pixel. continue_existing_simulation (bool): Whether to continue the existing simulation, or start a new simulation. Default value: *false*, i.e. any existing simulation results are discarded before doing the simulation.

   Returns: SimulationResultList: the results of the simulation

Raises: RuntimeError: if the simulation times out or fails

2. simulate(self: sme.Model, simulation_times: str, image_intervals: str, timeout_seconds: int = 86400, throw_on_timeout: bool = True, simulator_type: sme.SimulatorType = <SimulatorType.Pixel: 1>, continue_existing_simulation: bool = False) -> std::vector<sme::SimulationResult, std::allocator<sme::SimulationResult> >

   returns the results of the simulation.

   Args: simulation_times (str): The length(s) of the simulation in model units of time as a comma-delimited list, e.g. *"5"*, or *"10;100;20"* image_intervals (str): The interval(s) between images in model units of time as a comma-delimited list, e.g. *"1"*, or *"2;10;0.5"* timeout_seconds (int): The maximum time in seconds that the simulation can run for. Default value: 86400 = 1 day. throw_on_timeout (bool): Whether to throw an exception on simulation timeout. Default value: *true*. simulator_type (sme.SimulatorType): The simulator to use: *sme.SimulatorType.DUNE* or *sme.SimulatorType.Pixel*. Default value: Pixel. continue_existing_simulation (bool): Whether to continue the existing simulation, or start a new simulation. Default value: *false*, i.e. any existing simulation results are discarded before doing the simulation.

   Returns: SimulationResultList: the results of the simulation

   Raises: RuntimeError: if the simulation times out or fails

Attributes

compartment_image	an image of the compartments in this model
compartments	the compartments in this model
membranes	the membranes in this model
name	the name of this model
parameters	the parameters in this model

sme.Model.compartment_image

property Model.**compartment_image**
an image of the compartments in this model

Type list of list of list of int

sme.Model.compartments

property Model.**compartments**
the compartments in this model

a list of *Compartment* that can be iterated over, or indexed into by name or position in the list.

Examples

the list of compartments can be iterated over:

```
>>> import sme
>>> model = sme.open_example_model()
>>> for compartment in model.compartments:
...     print(compartment.name)
Outside
Cell
Nucleus
```

or a compartment can be found using its name:

```
>>> cell = model.compartments["Cell"]
>>> print(cell.name)
Cell
```

or indexed by its position in the list:

```
>>> last_compartment = model.compartments[-1]
>>> print(last_compartment.name)
Nucleus
```

Type *CompartmentList*

sme.Model.membranes

property Model.**membranes**
the membranes in this model

a list of *Membrane* that can be iterated over, or indexed into by name or position in the list.

Examples

the list of membranes can be iterated over:

```
>>> import sme
>>> model = sme.open_example_model()
>>> for membrane in model.membranes:
...     print(membrane.name)
Outside <-> Cell
Cell <-> Nucleus
```

or a membrane can be found using its name:

```
>>> outer = model.membranes["Outside <-> Cell"]
>>> print(outer.name)
Outside <-> Cell
```

or indexed by its position in the list:

```
>>> last_membrane = model.membranes[-1]
>>> print(last_membrane.name)
Cell <-> Nucleus
```

Type *MembraneList*

sme.Model.name

property Model.**name**
the name of this model

Type str

sme.Model.parameters

property Model.**parameters**
the parameters in this model

a list of *Parameter* that can be iterated over, or indexed into by name or position in the list.

Examples

the list of parameters can be iterated over:

```
>>> import sme
>>> model = sme.open_example_model()
>>> for parameter in model.parameters:
...     print(parameter.name)
param
```

or a parameter can be found using its name:

```
>>> p = model.parameters["param"]
>>> print(p.name)
param
```

or indexed by its position in the list:

```
>>> last_param = model.parameters[-1]
>>> print(last_param.name)
param
```

Type *ParameterList*

9.1.9 sme.Parameter

class sme.**Parameter**
a parameter of the model

Methods

__init__(*args, **kwargs)	Initialize self.

sme.Parameter.__init__

Parameter.**__init__**(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

name	the name of this parameter
value	the mathematical expression for this reaction parameter

sme.Parameter.name

property Parameter.**name**
the name of this parameter

Type str

sme.Parameter.value

property Parameter.**value**
the mathematical expression for this reaction parameter

Type str

9.1.10 sme.ParameterList

class sme.**ParameterList**
ParameterList: a list of *Parameter*

the list can be iterated over, or an element can be looked up by its index or name

Methods

__init__(self)	

sme.ParameterList.__init__

ParameterList.**__init__**(*self:* sme.ParameterList) → None

9.1.11 sme.Reaction

class sme.**Reaction**

a reaction between species

Methods

__init__(*args, **kwargs)	Initialize self.

sme.Reaction.__init__

Reaction.**__init__**(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Attributes

name	the name of this reaction
parameters	the parameters of this reaction

sme.Reaction.name

property Reaction.**name**

the name of this reaction

Type str

sme.Reaction.parameters

property Reaction.**parameters**

the parameters of this reaction

Type *ReactionParameterList*

9.1.12 sme.ReactionList

class sme.**ReactionList**

ReactionList: a list of *Reaction*

the list can be iterated over, or an element can be looked up by its index or name

Methods

__init__(self)	

sme.ReactionList.__init__

ReactionList.**__init__**(*self:* sme.ReactionList) → None

9.1.13 sme.ReactionParameter

class sme.**ReactionParameter**

a parameter of a reaction

Methods

__init__(*args, **kwargs)	Initialize self.

sme.ReactionParameter.__init__

ReactionParameter.**__init__**(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Attributes

name	the name of this reaction parameter
value	the value of this reaction parameter

sme.ReactionParameter.name

property ReactionParameter.**name**
the name of this reaction parameter

Type str

sme.ReactionParameter.value

property ReactionParameter.**value**
the value of this reaction parameter

Type float

9.1.14 sme.ReactionParameterList

class sme.**ReactionParameterList**
ReactionParameterList: a list of *ReactionParameter*

the list can be iterated over, or an element can be looked up by its index or name

Methods

__init__(self)	

sme.ReactionParameterList.__init__

ReactionParameterList.**__init__**(*self:* sme.ReactionParameterList) → None

9.1.15 sme.SimulationResult

class sme.**SimulationResult**
results at a single timepoint of a simulation

Methods

__init__(*args, **kwargs)	Initialize self.

sme.SimulationResult.__init__

SimulationResult.**__init__**(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

concentration_image	an image of the species concentrations at this time-point
species_concentration	the species concentrations at this timepoint
species_dcdt	the rate of change of the species concentrations at this timepoint
time_point	the timepoint these simulation results are from

sme.SimulationResult.concentration_image

property SimulationResult.**concentration_image**
an image of the species concentrations at this timepoint

a triplet of red, green, blue values for each pixel in the image concentration_image[y][x] = [r, g, b]

Type list of list of list of int

sme.SimulationResult.species_concentration

property SimulationResult.**species_concentration**
the species concentrations at this timepoint

for each species, the concentrations are provided as a list of list of float, where species_concentration['A'][y][x] is the concentration of species 'A" at the point (x,y)

Type dict

sme.SimulationResult.species_dcdt

property SimulationResult.**species_dcdt**
the rate of change of the species concentrations at this timepoint

for each species, the rate of change of the concentrations are provided as a list of list of float, where species_dcdt['A'][y][x] is the rate of change of concentration of species 'A' at the point (x,y)

Type dict

sme.SimulationResult.time_point

property SimulationResult.**time_point**
the timepoint these simulation results are from

Type float

9.1.16 sme.SimulationResultList

class sme.**SimulationResultList**
SimulationResultList: a list of *SimulationResult*

the list can be iterated over, or an element can be looked up by its index or name

Methods

__init__(self)

---

sme.SimulationResultList.__init__

SimulationResultList.**__init__**(*self:* sme.SimulationResultList) → None

9.1.17 sme.SimulatorType

class sme.**SimulatorType**
Members:

DUNE

Pixel

Methods

__init__(self, value)

---

sme.SimulatorType.__init__

SimulatorType.**__init__**(*self:* sme.SimulatorType, *value: int*) → None

Attributes

DUNE
Pixel
name
value

sme.SimulatorType.DUNE

SimulatorType.**DUNE = <SimulatorType.DUNE: 0>**

sme.SimulatorType.Pixel

SimulatorType.**Pixel = <SimulatorType.Pixel: 1>**

sme.SimulatorType.name

property SimulatorType.**name**

sme.SimulatorType.value

property SimulatorType.**value**

9.1.18 sme.Species

class sme.**Species**

a species that lives in a compartment

Methods

__init__(*args, **kwargs)	Initialize self.

sme.Species.__init__

Species.**__init__**(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

diffusion_constant	the diffusion constant of this species
name	the name of this species

sme.Species.diffusion_constant

property Species.**diffusion_constant**
the diffusion constant of this species

Type float

sme.Species.name

property Species.**name**
the name of this species

Type str

9.1.19 sme.SpeciesList

class sme.**SpeciesList**
SpeciesList: a list of *Species*

the list can be iterated over, or an element can be looked up by its index or name

Methods

__init__(self)	

sme.SpeciesList.__init__

SpeciesList.**__init__**(*self:* sme.SpeciesList) → None

Exceptions

InvalidArgument
RuntimeError

9.1.20 sme.InvalidArgument

exception sme.**InvalidArgument**

9.1.21 sme.RuntimeError

exception sme.**RuntimeError**

CHAPTER

TEN

DUNE-COPASI SIMULATOR

dune-copasi is the default PDE solver, which solves the PDE on a triangular mesh using finite element discretization methods. The mesh is automatically constructed from the geometry image, as described in *Mesh generation*.

10.1 Simulation options

The default settings should work well in most cases, but if desired they can be adjusted by going to *Advanced->Simulation options*

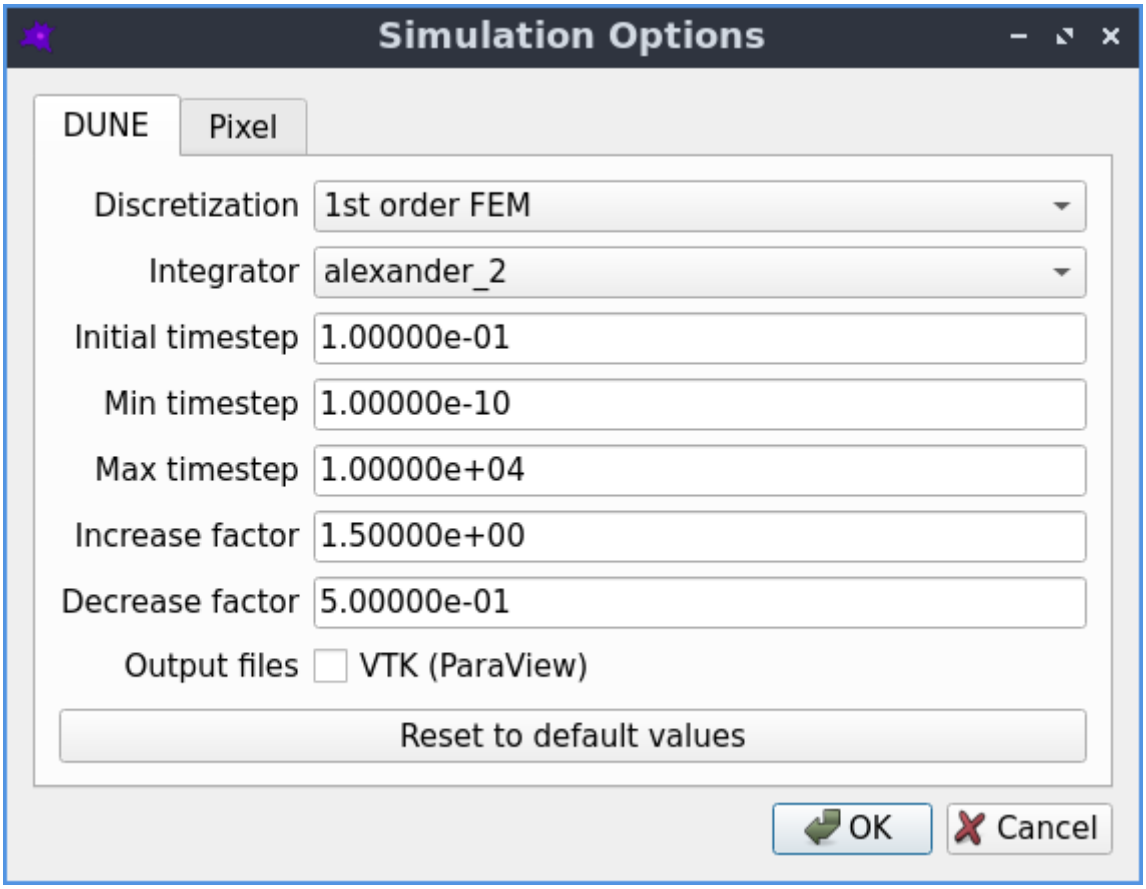


Fig. 1: The simulation options that can be used to fine tune-the dune-copasi solver.

- **Discretization**

  - currently only 1st order FEM is supported
  - other discretizations (such as 2nd order FEM) may be added in the future
- **Integrator**
  - the Runge-Kutta integration scheme used for time integration
  - a variety of implicit and explicit schemes of different orders are available
  - the default is the 2nd order Alexander scheme, which is a Diagonally Implicit Runge Kutta method
- **Initial timestep**
  - the timestep used at the start of a simulation
- **Min timestep**
  - the minimum allowed timestep
  - for a very stiff model it may be necessary to reduce this value
- **Max timestep**
  - the maximum allowed timestep
  - reducing this value may increase the accuracy of the solution but simulations will take longer to run
- **Increase factor**
  - after a successful integration step, the timestep is multiplied by this factor
  - this must be greater than or equal to 1
  - if equal to 1, the timestep is never increased between integration steps
  - the larger the value, the more the timestep is increased after successful integration steps
- **Decrease factor**
  - if an integration step is unsuccessful, the timestep is multiplied by this factor and the step is repeated
  - this must be less than 1
  - the smaller the value, the more the timestep is decreased in the case of an unsuccessful integration step
- **Output files**
  - VTK files of the species concentrations throughout the simulation can be generated
  - these files can be viewed using ParaView

For more information on the solver see the dune-copasi documentation.

CHAPTER

ELEVEN

PIXEL SIMULATOR

Pixel is an alternative PDE solver which uses the simple FTCS method to solve the PDE using the pixels of the geometry image as the grid.

11.1 Simulation options

The default settings should work well in most cases, but if desired they can be adjusted by going to *Advanced->Simulation options*

- **Integrator**
  - the explicit Runge-Kutta integration scheme used for time integration
  - default: 2nd order Heun scheme, with embedded 1st order error estimate
  - a higher order scheme may be more efficient if the maximum allowed error is very small
  - see the *Time integration* section for more information on the integrators
- **Max relative local error**
  - the maximum relative error allowed on the concentration of any species at any pixel
  - default: 0.005
  - local means the estimated error for a single timestep, at a single point
  - relative means each error estimate is divided by the species concentration
  - making this number smaller makes the simulation more accurate, but slower
- **Max absolute local error**
  - the maximum error allowed on the concentration of any species at any pixel
  - default: infinite
  - local means the estimated error for a single timestep, at a single point
  - absolute means the error estimate is not normalised by the species concentration
  - making this number smaller makes the simulation more accurate, but slower
- **Max timestep**
  - the maximum allowed timestep
  - default: infinite
- **Multithreading**

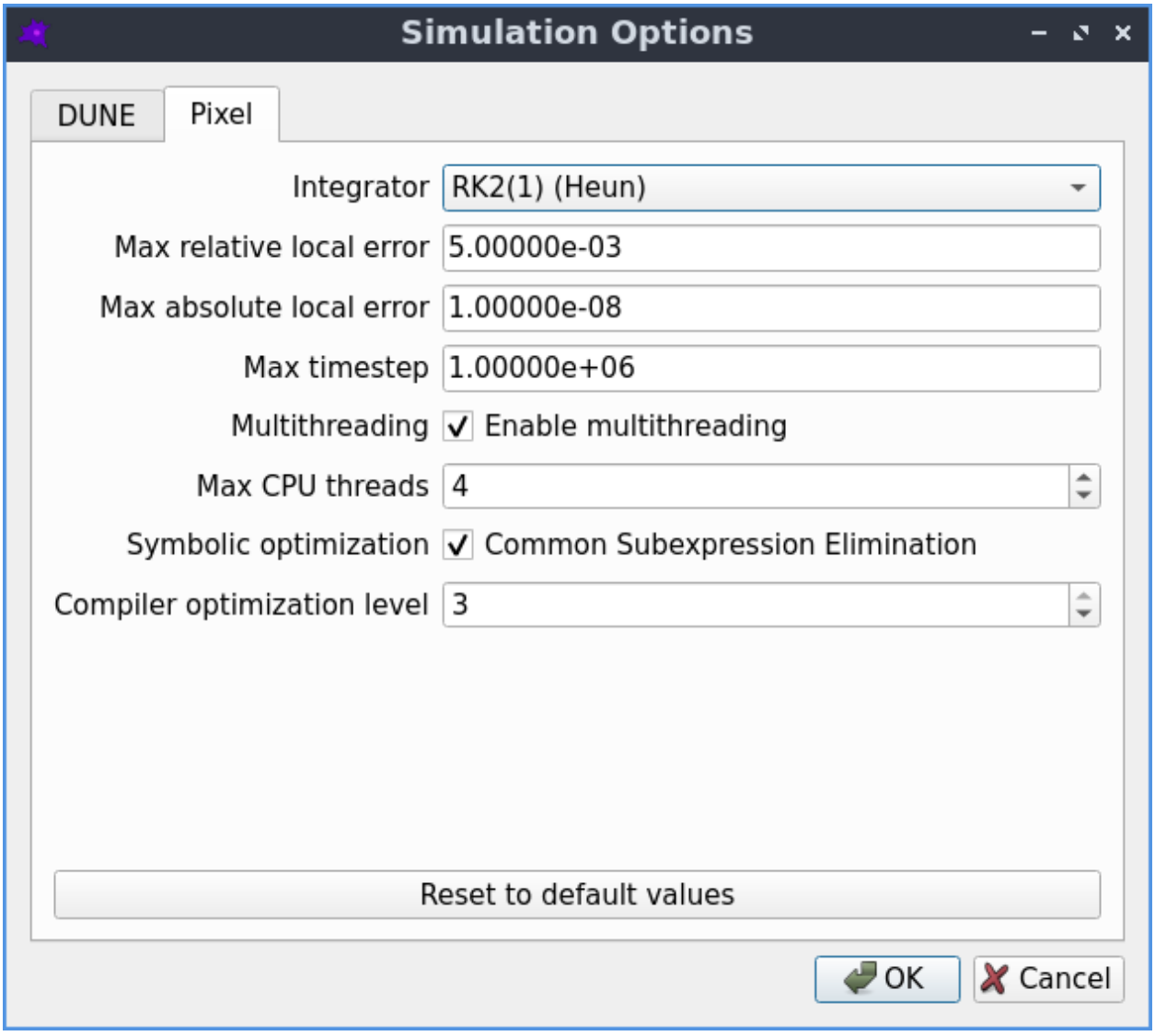


Fig. 1: The simulation options that can be used to fine tune-the Pixel solver.

  - if enabled, multiple CPU threads can be used
  - default: disabled
  - enabling this can make simulations of large models run faster
  - however it can also make small models run slower

- **Max CPU threads**
  - limit the maximum number of CPU threads to be used
  - default: unlimited
- **Symbolic optimization**
  - factor out common subexpressions when constructing the reaction terms
  - default: enabled
- **Compiler optimization level**
  - how much optimization is done when compiling the reaction terms
  - default: 3

11.2 Spatial discretization

Space is discretized using a uniform, linear grid with spacing a. The concentration is defined as a 2d array of values $c_{i,j}$, where the value with index (i, j) corresponds to the concentration at the spatial point $(x = ia, y = ja)$.

The Laplacian is approximated on this grid using a central difference scheme

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) c_{i,j} = \left[c_{i+1,j} + c_{i,j+1} - 4c_{i,j} + c_{i-1,j} + c_{i,j-1}\right]/a^2 + \mathcal{O}(a^2)$$

which has $\mathcal{O}(a^2)$ discretisation errors. Inserting this approximation into the reaction-diffusion equation converts the PDE into a system of coupled ODEs.

11.3 Time integration

Time integration is performed using explicit Runge-Kutta integrators. Compared to implicit integrators, they are easier to implement and offer better performance (for the same timestep). However they become unstable if the timestep h is made too large, so in practice they can end up being slower than implicit methods for stiff problems, where the timestep is forced to be very small to maintain stability.

Integrators differ in their:

- order of truncation error
- order of embedded error estimate (if any)
- number of stages (i.e. cost of a step)
- region of stability (can be increased by adding more stages)
- memory requirements

Implemented integrators:

- **Euler**

  - 1st order solution
  - no error estimate
  - 1 stage
  - see e.g. https://en.wikipedia.org/wiki/Euler_method
- **Embedded Heun / modified Euler**
  - 2nd order solution
  - 1st order error estimate
  - 2 stages
  - see e.g. eq (2.15) of https://doi.org/10.1016/0021-9991(88)90177-5
- **Embedded Shu-Osher**
  - 3rd order solution
  - 2nd order error estimate
  - 3 stages
  - see eq (2.17) of https://doi.org/10.1016/0021-9991(88)90177-5
- **RK4(3)5[3S*]**
  - 4th order solution
  - 3rd order error estimate
  - 5 stages
  - see alg.6 & tab.6 of https://doi.org/10.1016/j.jcp.2009.11.006

These integrators have three sources of error:

- **Round-off error due to finite precision**
  - mostly only relevant for high order solvers: not relevant here
- **Truncation error due to finite order of integration scheme**
  - we are generally forced by the diffusion term to make the timestep small to maintain stability
  - also no benefit from making the time integration errors significantly smaller than the spatial discretisation errors
  - so this is also typically not a concern
- **Numerical instability of integrator**
  - a problem when ODEs become stiff, e.g. high rate of diffusion, stiff reaction terms
  - avoiding these instabilities is our main concern

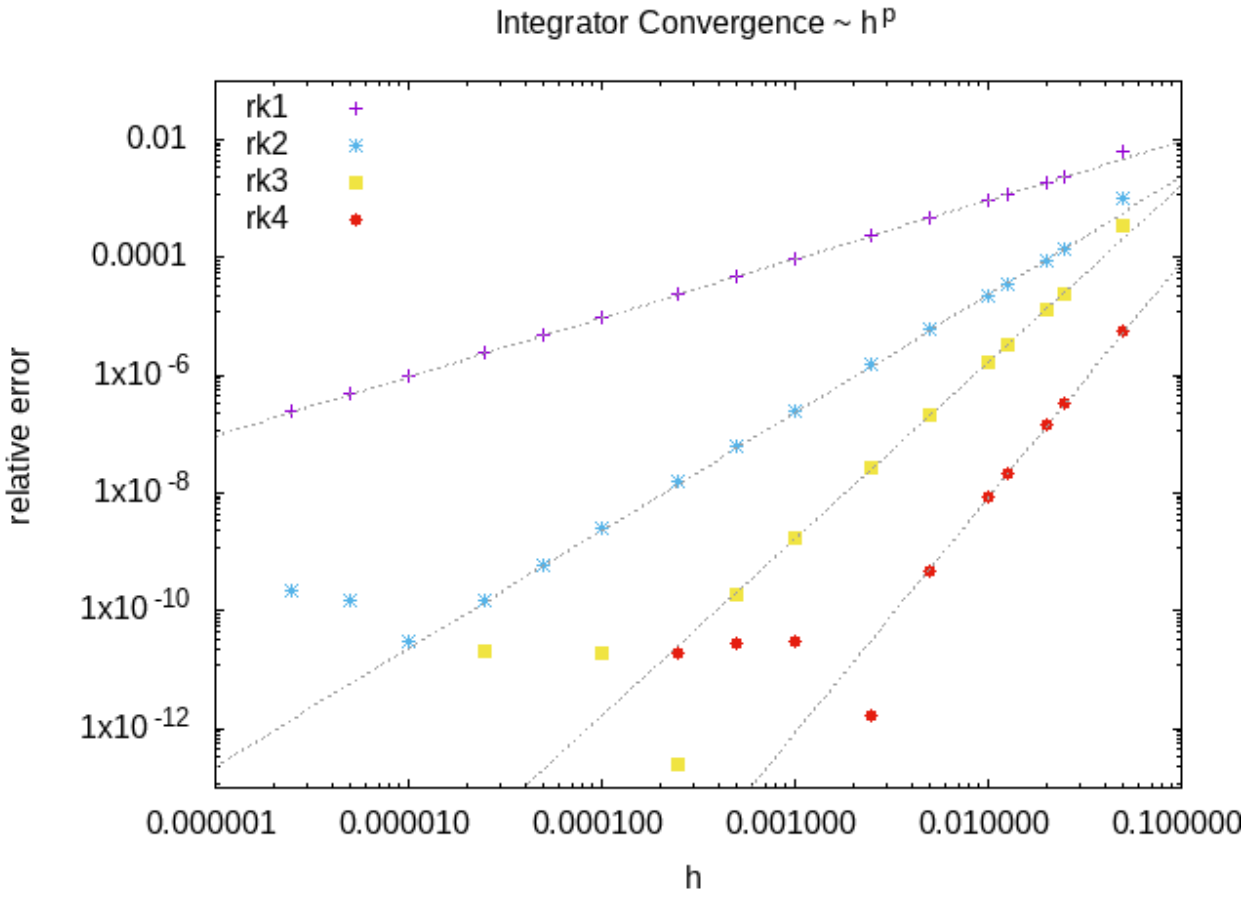


Fig. 2: An example of the convergence of the included RK integrators: relative error of the solution at a particular pixel as a function of the stepsize.

11.4 Adaptive timestep

We use the embedded lower order solution to estimate the error at each timestep, and use this to adapt the stepsize during the integration:

- RK gives us a pair of $u_{n+1}^{(p)} = u_n + \mathcal{O}(h^{p+1})$ solutions
- difference between $p, p-1$ solutions gives local error of order $\mathcal{O}(p)$
- to get the relative error we symbolicDivide this by $c = (|c_{n+1}| + |c_n| + \epsilon)/2$
- we use the average of the old and new concentration, plus a small constant, to avoid dividing by zero
- we do this for all species, compartments and spatial points, and take the maximum value
- if this error is larger than the desired value, the whole step is discarded
- the new timestep is given by $0.98 dt_{old}(err_{desired}/err_{measured})^{1/p}$
- the 0.98 factor is slightly less than 1 to account for the higher order terms that are neglected here
- it is better to have a slightly smaller timestep than to have to repeat the whole step

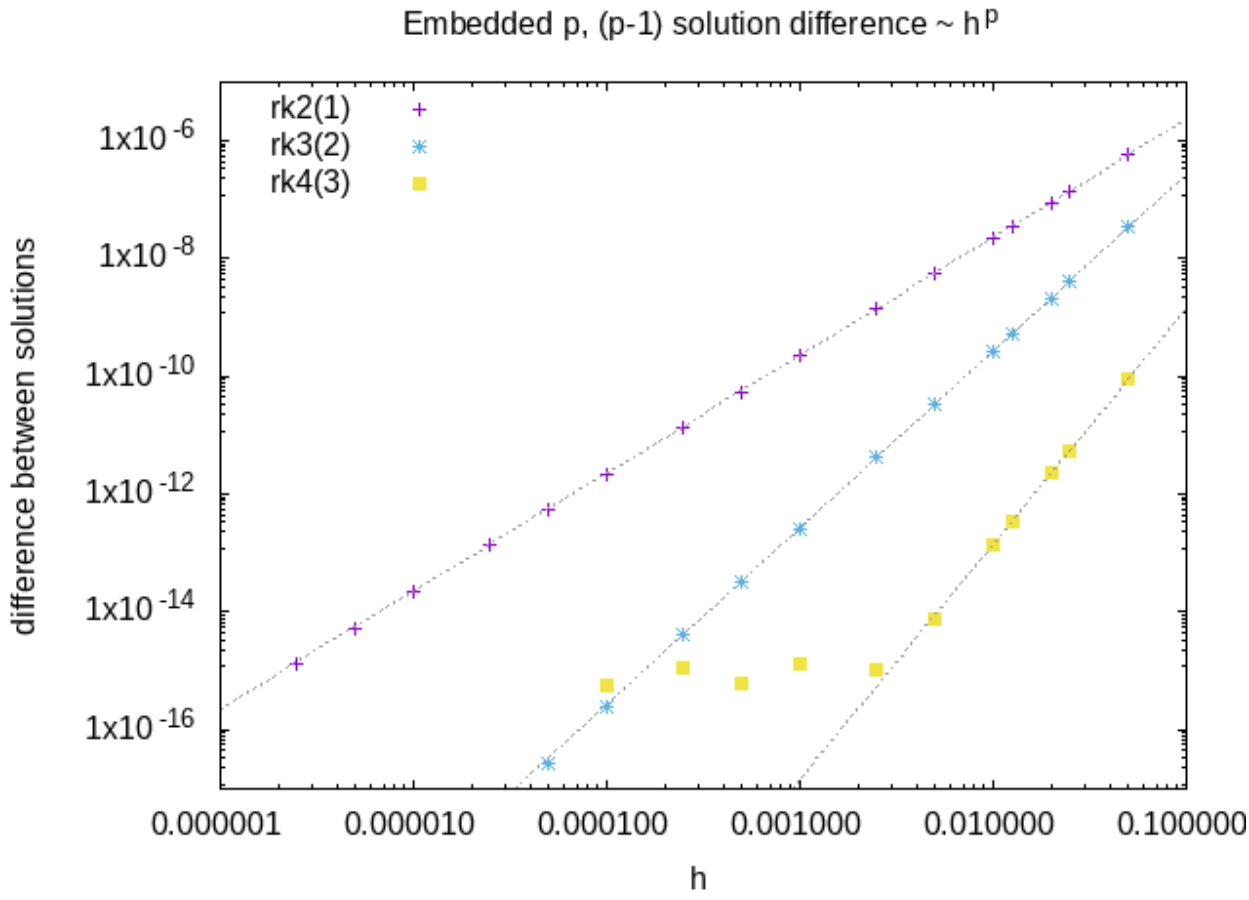


Fig. 3: An example of the difference between order p and order p-1 solutions from embedded schemes as a function of the stepsize. This quantity is a measure of the local integration error, and scales like h^p

11.5 Maximum timestep

For the Euler method, we don't have an embedded lower order solution from which we can estimate of the error, so we can't automatically adjust the stepsize. However, if we ignore the reaction terms, there is an analytic upper bound on the size of timestep that can be used for Euler, above which the system becomes unstable:

$$\delta t \leq \frac{a^2}{4D}$$

So if the user selects a timestep larger than this, the simulator automatically reduces it to the above value to avoid the system becoming unstable. Note that the system can still become unstable if the reaction terms are stiffer than the diffusion terms.

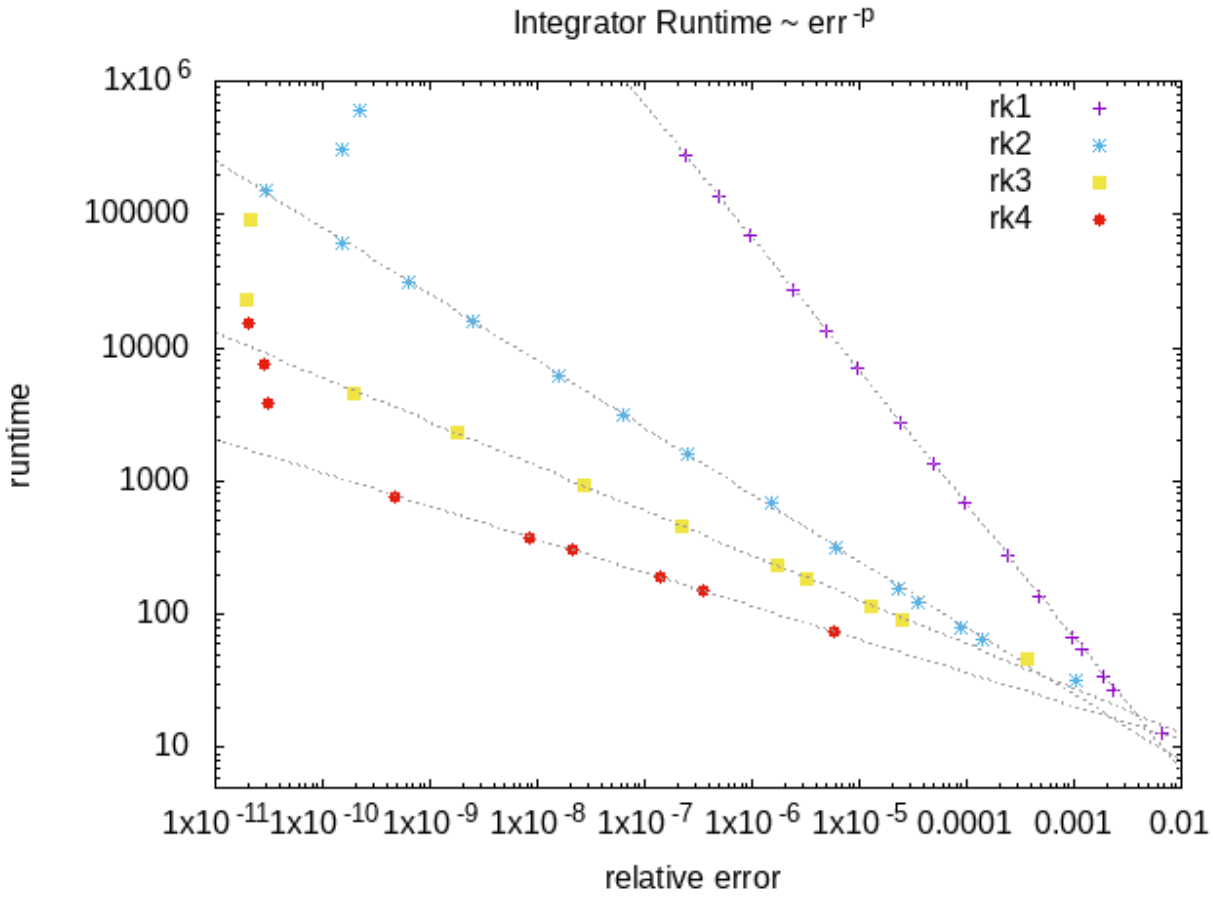


Fig. 4: An example of the runtime of the RK integrators as a function of the relative error on the final solution. The higher order integrators offer better performance if a very accurate solution is required, but at lower accuracy the lower order integrators are much faster.

11.6 Boundary Conditions

The boundary condition for all boundaries is the "zero-flux" Neumann boundary condition. This is implemented in the spatial discretization by setting the concentration in the neighbouring pixel that lies outside the compartment boundary to be equal to the concentration in the boundary pixel value, or equivalently by setting the neighbour of each boundary pixel to itself.

11.6.1 Compartments

Each compartment is discretized, with the above boundary conditions applied for the diffusion term.

11.6.2 Membranes

Reactions that take place between two compartments involve a flux across the membrane separating the two compartments. For each neighbouring pair of pixels from the two compartments, whose common boundary constitutes the membrane, the flux term is converted into a reaction term that creates or destroys the appropriate amount of species concentration in each pixel.

11.6.3 Non-spatial species

A species can be 'non-spatial', which means that at each timestep, its time derivative is calculated as normal at each point in the compartment, but is then spatially averaged over the whole compartment. This can be used to approximate a species with a very high diffusion constant without requiring a correspondingly tiny timestep to maintain the stability of the solver.

CHAPTER

TWELVE

MESH GENERATION

Generating a triangular mesh for the dune-copasi solver from a pixel image of the compartment geometry involves multiple steps:

- *Pixel contours*
- *Pixel-Edge contours*
- *Boundary line simplification*
- *Interior points*
- *Triangulation*

These steps are described in more detail below, starting from this initial segmented image of the model geometry to illustrate each stage:

12.1 Pixel contours

The first step in generating the mesh is to identify the set of contours that make up the boundaries of each compartment, as well as the boundary between the model geometry (i.e. all the compartments) and the outside.

The contour tracing is done using the findContours function from the OpenCV library, which implements the method described in Suzuki et. al.. This method returns an ordered, closed loop of 8-connected pixels for each contour. Each compartment has at least one contour around its outer boundary, and it may also contain inner contours around any holes in the compartment shape. Outer contours trace an outer boundary of a compartment in an anti-clockwise direction, while the inner contours trace an inner boundary of a compartment in a clockwise direction. All the pixels used to construct the contours lie within the compartment.

12.2 Pixel-Edge contours

If our compartments were all independent closed loops, then we could directly use the pixel contours to construct the compartment boundaries, and simplify each contour independently. However, once two compartments touch this is no longer the case, as we have to ensure that the part of each contour that is shared between the two compartments is simplified in the same way, to avoid creating gaps between the compartments.

To deal with this, we construct a 4-connected contour of outer pixel edges from each 8-connected pixel contour. If the image has width W and height H, then the vertices of the edge contours are located on a grid of width $W+1$ and height $H+1$, where the *(0,0)* vertex corresponds to the top-left corner of the *(0,0)* pixel. The advantage of this contour representation is that if the outer part of two pixel contours are adjacent to each other, their outer pixel-edge contours will coincide, which allows us to identify shared sub-contours unambiguously. Vertices where three different contours intersect are identified and used to split the contours into lines which separate pairs of adjacent compartments, and duplicates are removed.

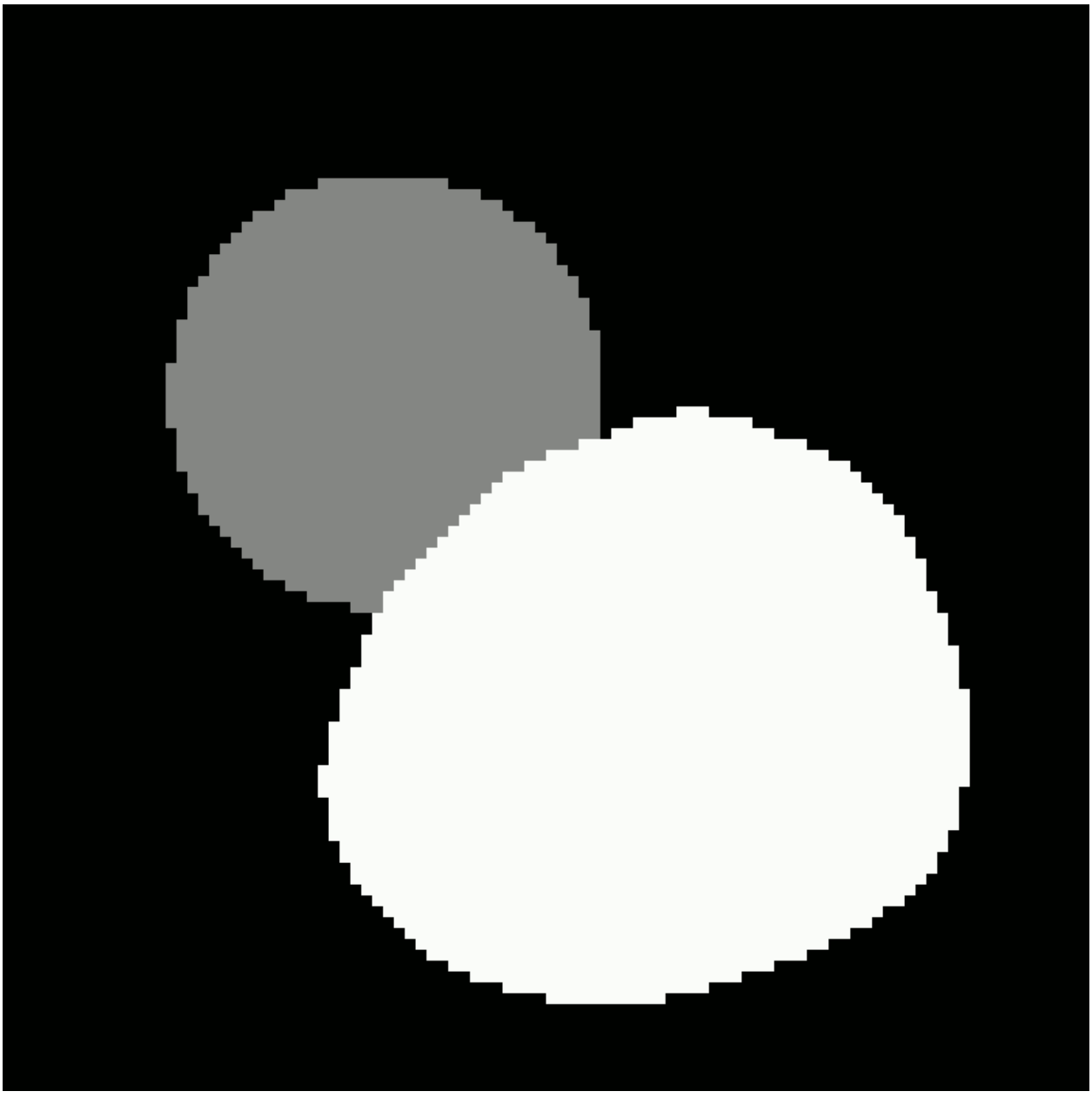


Fig. 1: Initial segmented geometry image.

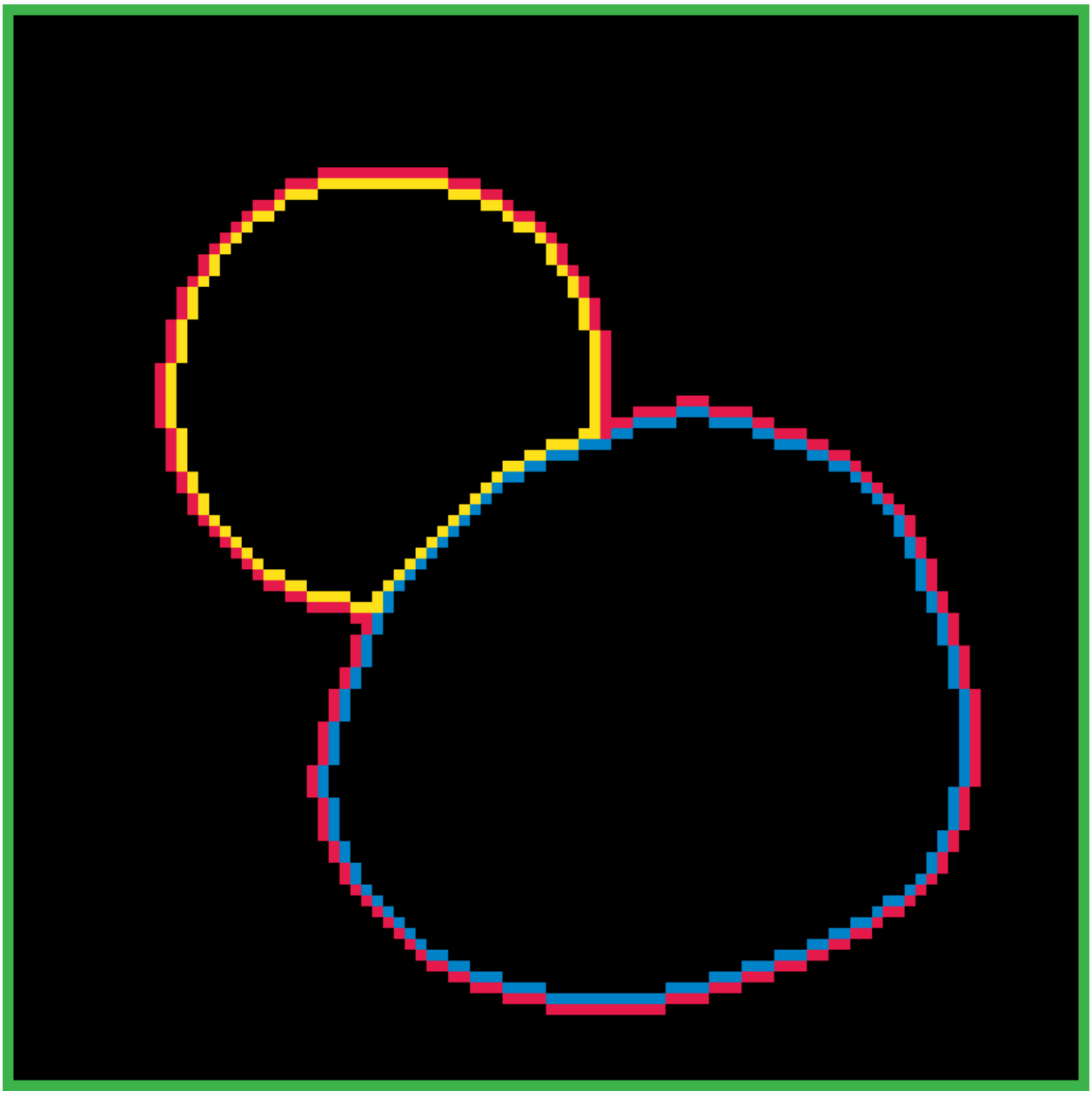


Fig. 2: Pixel Boundary contours.

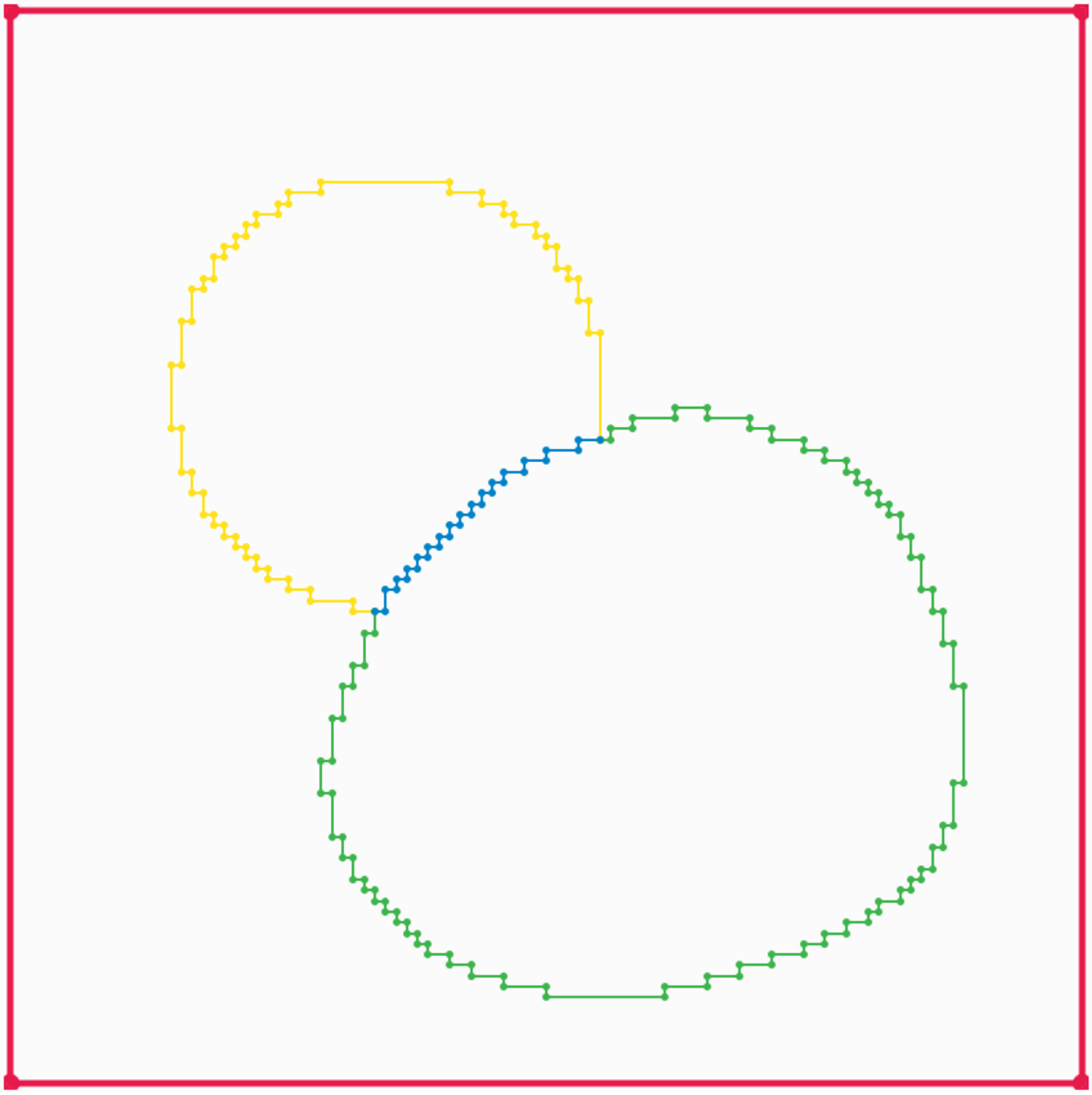


Fig. 3: Split pixel-edge boundary contours.

12.3 Boundary line simplification

Once we have identified all of our boundaries, we want to simplify them by removing points from the boundary. We do this using Visvalingam-Whyatt polyline simplification. The algorithm starts by calculating the area of the triangle formed by each point on the boundary with its two nearest neighbouring points. Then at each step:

- the point with the smallest area is removed
- the two neighbouring points areas are recalculated
- the larger of the previous and the new area is used

This allows us to order the points in the boundary by their order of importance, and then the user can adjust the number of points used for each boundary as desired.

12.4 Interior points

Each connected region of pixels in a compartment needs an interior point to be specified to identify the region during triangulation. An interior point is determined for each connected region by repeatedly eroding a binary image of the region until all pixels are gone, then taking one of the remaining pixels from the previous step.

It is important that the interior point is as far as possible from any of the boundaries of the compartment, to ensure that it remains within these boundaries even when they are simplified, so that the compartment that corresponds to the region is correctly identified during triangulation.

The connected regions are identified using the

The animations below show how the algorithm works, which uses the connectedComponents() and erode() functions from the OpenCV library

Meaning of the colours used in the animations:

- grey area: original connected region pixels
- black area: remaining pixels after previous erosions
- green rectangle: region of interest where the next erosion operation will be applied
- red dot: current interior point

12.5 Triangulation

The set of boundaries can then be triangulated using the Triangle library. This generates a constrained conforming Delaunay triangulation (CCDT) from the boundary lines, by inserting points inside the compartments and triangulating them. If necessary it will also add additional points on the boundary lines (known as Steiner points). The maximum allowed triangle area for each compartment can be specified by the user.

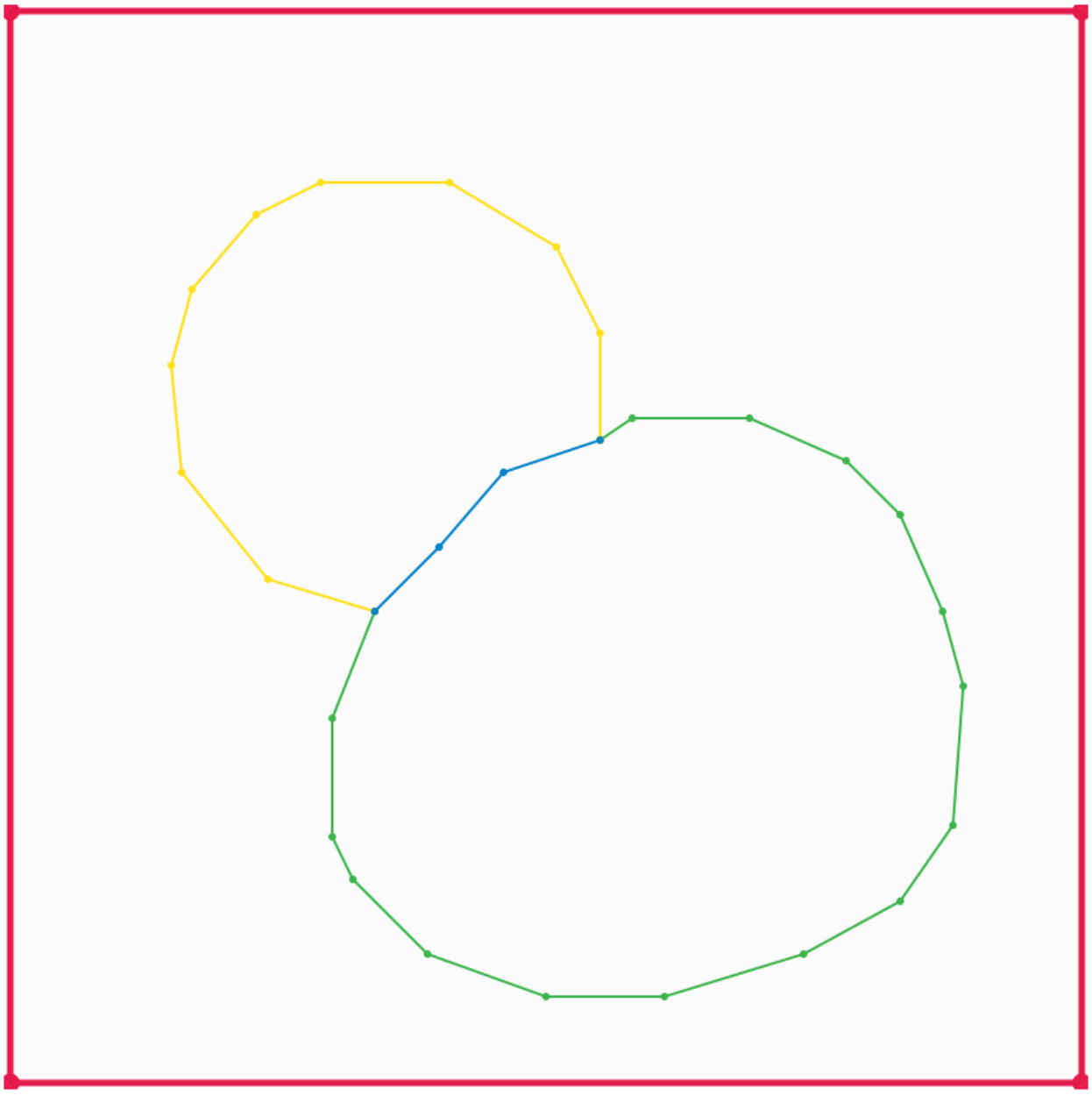


Fig. 4: Simplified boundary lines.

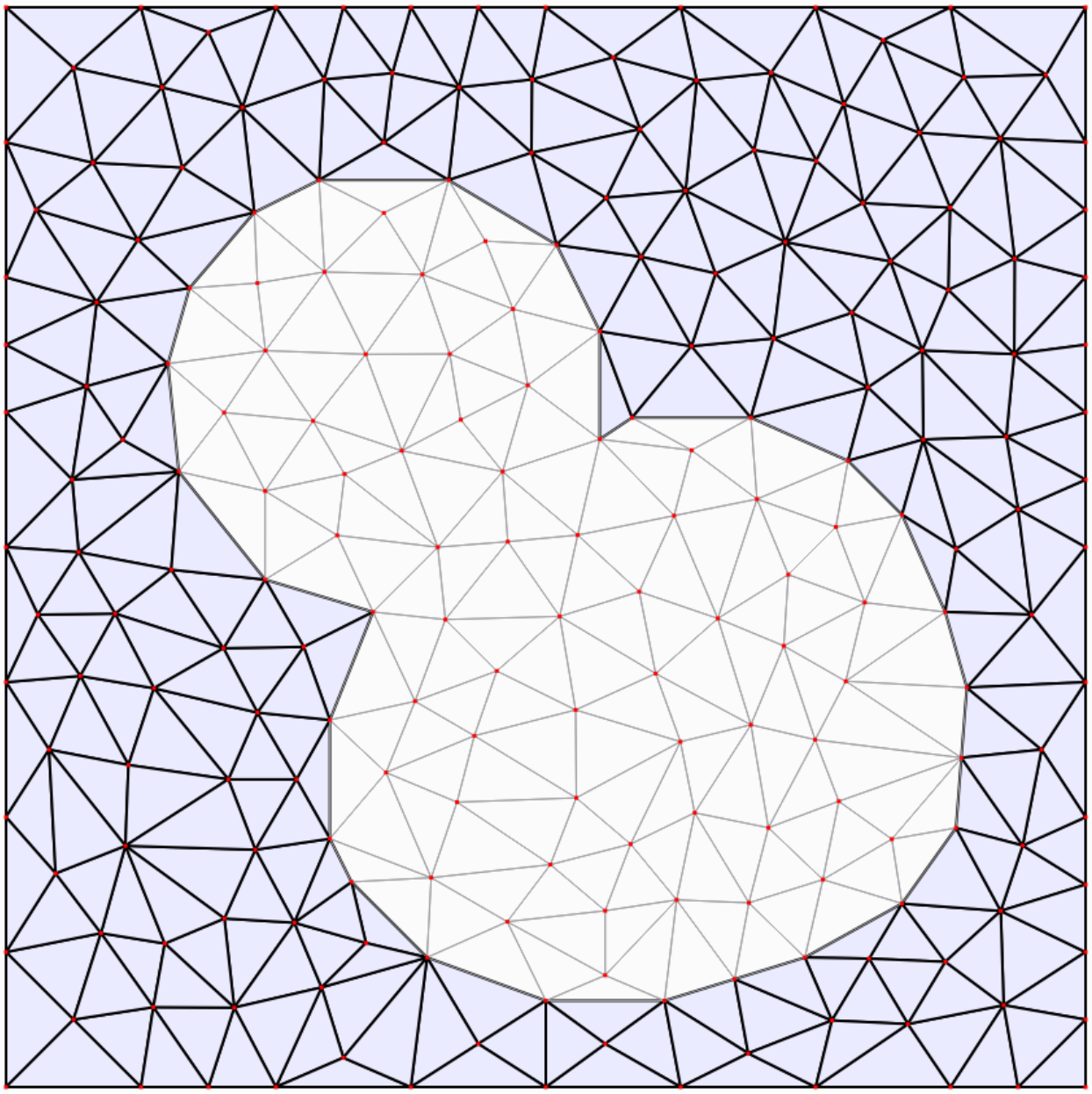


Fig. 5: Generated triangular mesh.

CHAPTER

THIRTEEN

COMMAND LINE INTERFACE

A Command Line Interface (CLI) is also provided for running long simulations:

-  linux
-  macOS
-  windows

13.1 Use

It can be used to simulate a sbml model and save the results. For example, this would simulate the model from the file `filename.xml` for ten units of time, with 1 unit of time between images, and store the results in `results.sme`:

```
./spatial-cli filename.xml 10 1 -o results.sme
```

The file `results.sme` can be opened in the GUI to see the simulation results.

An existing simulation can also be continued, for example this would simulate 5 more steps of length 1 and append the results to the existing simulation results in `results.sme`:

```
./spatial-cli results.sme 5 1
```

If the output file is not specified it defaults to overwriting the input file.

Multiple time intervals can be specified as semicolon delimited lists, the same as in the GUI. For example:

```
./spatial-cli results.sme 5;25;10 1;2.5;0.1
```

13.2 Command line parameters

```
Spatial Model Editor CLI v1.0.9
Usage: ./cli/spatial-cli [OPTIONS] file times image-intervals

Positionals:
  file TEXT:FILE REQUIRED     The spatial SBML model to simulate
  times TEXT REQUIRED         The simulation time(s) (in model units of time)
  image-intervals TEXT REQUIRED
                              The interval(s) between saving images (in model units␣
↪of time)

Options:
  -h,--help                   Print this help message and exit
  -s,--simulator ENUM:value in {dune->0,pixel->1} OR {0,1}=0
                              The simulator to use: dune or pixel
  -o,--output-file TEXT       The output file to write the results to. If not set,␣
↪then the input file is used.
  -n,--nthreads UINT:NONNEGATIVE=0
                              The maximum number of CPU threads to use (0 means␣
↪unlimited)
  -v,--version                Display the version number and exit
  -d,--dump-config            Dump the default config ini file and exit
  -c,--config                 Read an ini file containing simulation options
```

13.3 Using a config file

To create an ini file with the default options

```
./spatial-cli -d > config.ini
```

You can then edit this file as desired, and use it when running a simulation

```
./spatial-cli filename.xml -c config.ini
```

CHAPTER

FOURTEEN

MATHS

14.1 Reaction-Diffusion

The system of PDEs that we simulate in each compartment is the two-dimensional reaction-diffusion equation:

$$\frac{\partial c_s}{\partial t} = D_s \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) c_s + R_s$$

where

- c_s is the concentration of species s at position (x, y) and time t
- D_s is the diffusion constant for species s
- R_s is the reaction term for species s

and we assume that

- the diffusion constant D_s is a scalar that does not vary with position or time
- the reaction term R_s is a function that can depend on the concentrations of other species in the model, but only locally, i.e. the concentrations at the same spatial coordinate.

Note: This is equivalent to simulating a 3-d system with no spatial variation in the z-direction. In our simulations we then assume that we are simulating a 2-d slice of such a 3-d system with unit length in the z-direction (i.e. the system has extent 1 in the length units of our model in the z-direction). This allows the user to use the usual 3-d units for concentration, etc.

14.2 Compartment Reactions

Compartment reaction terms correspond to the R_s term in the reaction-diffusion equation, and describe the rate of change of species concentration with time, and are evaluated at every point inside the compartment

14.3 Membrane reactions

Membrane reactions are reactions that occur on the membrane between two compartments, and describe the species amount that crosses the membrane per unit membrane area per unit time.

14.4 Boundary Conditions

All boundaries have "zero-flux" Neumann boundary conditions, whether they are boundaries between two compartments or boundaries between a compartment and the outside (except for the flux caused by any membrane reactions).

CHAPTER

FIFTEEN

UNITS

15.1 Fundamental Units

To describe a spatial model we need to define the following fundamental units:

- `amount` (e.g. *Mole*)
- `length` (e.g. *metre*)
- `time` (e.g. *second*)

Volume is not a fundamental unit. However, for user convenience we treat volume as a fundamental unit

- `volume` (e.g. *litre*)

This allows the use of units such as *cm* for length and *mMol/mL* for concentration, instead of the equivalent but less common *mMol/cm^3*.

15.2 Derived Units

All quantities in the model have units that can be written as some combination of these fundamental units:

- **species concentrations**
  - have units of `amount` / `volume`
- **reactions *inside* a compartment**
  - describe the rate of change of the concentration of species
  - have units of concentration / time, i.e. `amount` / `volume` / `time`
- **reactions *between* two compartments**
  - describe that rate at which a unit amount of species crosses a unit area of the *membrane*
  - where the membrane is the area where the two compartments touch each other
  - have units of amount / membrane-area / time, i.e. `amount` / `length` / `length` / `time`
- **diffusion constants**
  - have units of *area / time*, i.e. `length` * `length` / `time`

15.3 More information

For more information see `units.pdf`

CHAPTER

SIXTEEN

SOURCE CODE

The source code is available from GitHub, where Bug reports and Feature requests are very welcome.

CHAPTER

SEVENTEEN

DIFFUSION

17.1 Model

The example model single-compartment-diffusion is a single compartment that contains two species: 'fast' and 'slow', each with the same analytic initial distribution

$$c_s(t=0) = e^{-((x-48)^2+(y-48)^2)/36}$$

The two species have different diffusion coefficients: $D = 1cm^2/s$ for species 'slow', and $D = 3cm^2/s$ for species 'fast', and the model contains no reactions.

17.2 Analytic solution

For this system without reactions, we are simulating the two-dimensional diffusion equation,

$$\frac{\partial c_s}{\partial t} = D_s \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) c_s$$

where

- c_s is the concentration of species s at position (x, y) and time t
- D_s is the diffusion constant for species s

For the initial condition $c_s(t=0) = \delta(x)\delta(y)$, the analytic solution at time *t* of this equation is the heat kernel:

$$c_s(t) = \frac{1}{4\pi D_s t} e^{-(x^2+y^2)/(4D_s t)}$$

and a solution for our model can then be found by convolving this expression with our initial condition, to give

$$c_s(t) = \frac{t_0}{t+t_0} e^{-((x-48)^2+(y-48)^2)/(4D_s(t+t_0))}$$

where $t_0 = 9/D_s$. Note that this solution ignores boundary effects, so will not be valid at late times or close to the compartment boundary.

The total amount of species in the compartment is a conserved quantity,

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} c_s(t) dx dy = 36\pi$$

and this is also valid at late times, since our zero flux Neumann boundary conditions also conserve the amount of species in the compartment.

PYTHON MODULE INDEX

S

INDEX

Symbols

C

D

E

G

I

M

N

O

P

R

S

T

V