spatial-model-editor

Liam Keegan

May 15, 2024

GETTING STARTED

CHAPTER

ONE

GETTING STARTED

1.1 Installation

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

-  linux
-  macOS arm64
-  macOS intel
-  windows

Tip: You may have to give permission before your operating system will run the executable:

- Windows: "More info"->"Run anyway"
- MacOS: right-click open
- Linux: `chmod +x spatial-model-editor`

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

- 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 SBML or COPASI model, go to *File->Open* or press `Ctrl+O`.

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

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

CHAPTER

THREE

IMPORTING GEOMETRY

After importing a non-spatial model, the next step is to import an image of the compartments in the model to define the geometry.

To do this, go to *Import->Geometry from image* or press `Ctrl+I`. Alternatively, go to *Import->Example geometry image* to use one of the built-in example images.

Then you can specify the dimensions of the geometry image, and optionally reduce the resolution of the image or the number of colours to be used. The geometry image should be segmented such that each compartment has a unique colour.

Fig. 1: An example of importing a geometry image

CHAPTER

FOUR

ASSIGN COMPARTMENTS

After importing a geometry image, each compartment can be assigned a region in the image:

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

The compartment geometry will then be made up from all of the pixels in the image of the chosen colour.

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

CHAPTER

FIVE

RESCALE REACTIONS

A non-spatial ODE model contains reaction rates that specify a rate of change of species **total amount** in a given compartment.

In a PDE model, we have

- compartment reactions: rate of change of species **concentration**
- membrane reactions: rate of species **amount crossing unit area** of the membrane

The last step in importing a non-spatial model is to rescale the reactions appropriately, by clicking on "rescale reactions" in the bottom right of the window.

This opens a guided workflow that automatically rescales the image depth and the reaction rate expressions, with the goal of constructing a valid PDE model that initially reproduces the behaviour of the imported ODE model, in the following limit of the spatial model:

- no initial spatial dependence of species concentrations
- infinitely fast diffusion
- consistent compartment volumes between ODE model and PDE model geometry

Fig. 1: An example of the automatic rescaling of image depth and reaction rates when importing a non-spatial model

CHAPTER

SIX

MESH GENERATION

Once the geometry has been created from an image and all compartments have been assigned a colour, a meshed approximation to the geometry is automatically constructed.

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

The *Mesh* tab shows the generated triangular mesh with the currently selected compartment highlighted. The maximum triangle area for each compartment 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

SEVEN

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

EIGHT

REACTIONS

A non-spatial ODE model contains reaction rates that specify a rate of change of species **total amount** in a given compartment.

In a PDE model, we have two kinds of reactions

- **Compartment reactions**
  - take place everywhere within a compartment
  - units: rate of change of species **concentration**
- **Membrane reactions**
  - take place on the membrane between two compartments
  - units: rate of species **amount crossing unit area** of the membrane

When editing a reaction, an image of the location is displayed on the left, and the units are displayed below the reaction rate in the bottom right.

Fig. 1: Editing the reactions in a spatial model.

CHAPTER

NINE

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*, a high-quality FEM solver, which solves the PDE on the triangular meshed approximation of the geometry.

The other alternative is *Pixel*, a simple FTCS solver, which solves the PDE on the grid formed by the pixels of the geometry image.

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

TEN

PYTHON INTERFACE

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

```
pip install sme
```

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.

10.1 sme-contrib

There is also a separate Python module sme-contrib, which contains additional functionality such as plotting and parameter fitting, and can be installed with

```
pip install sme-contrib
```

CHAPTER

ELEVEN

EXAMPLE NOTEBOOKS

Interactive online version

11.1 Getting started

11.1.1 Install and import `sme`

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

print("sme version:", sme.__version__)
```

```
Matplotlib is building the font cache; this may take a moment.
```

```
sme version: 1.6.0+8039f4d
```

11.1.2 Importing a model

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

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

11.1.3 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(builtins.object)
 |  the spatial model
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      __init__(self, filename: str) -> None
 |
 |  __repr__(...)
 |      __repr__(self) -> str
 |
 |  __str__(...)
 |      __str__(self) -> str
 |
 |  export_sbml_file(...)
 |      export_sbml_file(self, 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, 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, filename: str) -> None
 |
 |      sets the geometry of each compartment to the corresponding pixels in the␣
```

(continues on next page)

(continued from previous page)

```
↪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 volume 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(self, simulation_time: float, image_interval: float, timeout_seconds:␣
↪int = 86400, throw_on_timeout: bool = True, simulator_type: sme.SimulatorType =␣
↪SimulatorType.Pixel, continue_existing_simulation: bool = False, return_results: bool␣
↪= True, n_threads: int = 1) -> sme.SimulationResultList
 |      simulate(self, simulation_times: str, image_intervals: str, timeout_seconds: int␣
↪= 86400, throw_on_timeout: bool = True, simulator_type: sme.SimulatorType =␣
↪SimulatorType.Pixel, continue_existing_simulation: bool = False, return_results: bool␣
↪= True, n_threads: int = 1) -> sme.SimulationResultList
 |
 |      Overloaded function.
 |
 |      1. ``simulate(self, simulation_time: float, image_interval: float, timeout_
↪seconds: int = 86400, throw_on_timeout: bool = True, simulator_type: sme.SimulatorType␣
↪= SimulatorType.Pixel, continue_existing_simulation: bool = False, return_results:␣
↪bool = True, n_threads: int = 1) -> sme.SimulationResultList``
 |
 |                  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.
 |                      return_results (bool): Whether to return the simulation results.␣
↪Default value: `True`. If `False`, an empty SimulationResultList is returned.
 |                      n_threads(int): Number of cpu threads to use (for Pixel␣
↪simulations). Default value is 1, 0 means use all available threads.
 |
 |                  Returns:
```

(continues on next page)

(continued from previous page)

```
 |                     SimulationResultList: the results of the simulation
 |
 |                 Raises:
 |                     RuntimeError: if the simulation times out or fails
 |
 |
 |      2. ``simulate(self, simulation_times: str, image_intervals: str, timeout_seconds:
↪ int = 86400, throw_on_timeout: bool = True, simulator_type: sme.SimulatorType =␣
↪SimulatorType.Pixel, continue_existing_simulation: bool = False, return_results: bool␣
↪= True, n_threads: int = 1) -> sme.SimulationResultList``
 |
 |                 returns the results of the simulation.
 |
 |                 Args:
 |                     simulation_times (str): The length(s) of the simulation in model␣
↪units of time as a semicolon-delimited list, e.g. `"5"`, or `"10;100;20"`
 |                     image_intervals (str): The interval(s) between images in model␣
↪units of time as a semicolon-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.
 |                     return_results (bool): Whether to return the simulation results.␣
↪Default value: `True`. If `False`, an empty SimulationResultList is returned.
 |                     n_threads(int): Number of cpu threads to use (for Pixel␣
↪simulations). Default value is 1, 0 means use all available threads.
 |
 |                 Returns:
 |                     SimulationResultList: the results of the simulation
 |
 |                 Raises:
 |                     RuntimeError: if the simulation times out or fails
 |
 |  simulation_results(...)
 |      simulation_results(self) -> sme.SimulationResultList
 |
 |      returns the simulation results.
 |
 |      Returns:
 |          SimulationResultList: the simulation results
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from nanobind.nb_type_0
 |      Create and return a new object.  See help(type) for accurate signature.
 |
```

(continues on next page)

(continued from previous page)

```
|  ----------------------------------------------------------------------
|  Readonly properties defined here:
|
|  compartment_image
|      numpy.ndarray: an image of the compartments in this model
|
|      An array of RGB integer values for each voxel in the image of
|      the compartments in this model,
|      which can be displayed using e.g. ``matplotlib.pyplot.imshow``
|
|      Examples:
|          the image is a 4d (depth x height x width x 3) array of integers:
|
|          >>> import sme
|          >>> model = sme.open_example_model()
|          >>> type(model.compartment_image)
|          <class 'numpy.ndarray'>
|          >>> model.compartment_image.dtype
|          dtype('uint8')
|          >>> model.compartment_image.shape
|          (1, 100, 100, 3)
|
|          each voxel in the image has a triplet of RGB integer values
|          in the range 0-255:
|
|          >>> model.compartment_image[0, 34, 36]
|          array([144,  97, 193], dtype=uint8)
|
|          the first z-slice of the image can be displayed using matplotlib:
|
|          >>> import matplotlib.pyplot as plt
|          >>> imgplot = plt.imshow(model.compartment_image[0])
|
|  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:
|
```

(continues on next page)

(continued from previous page)

```
|          >>> 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,
|      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
|
|  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
```

(continues on next page)

(continued from previous page)

```
|
|           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
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  name
|      str: the name of this model
```

11.1.4 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'>
```

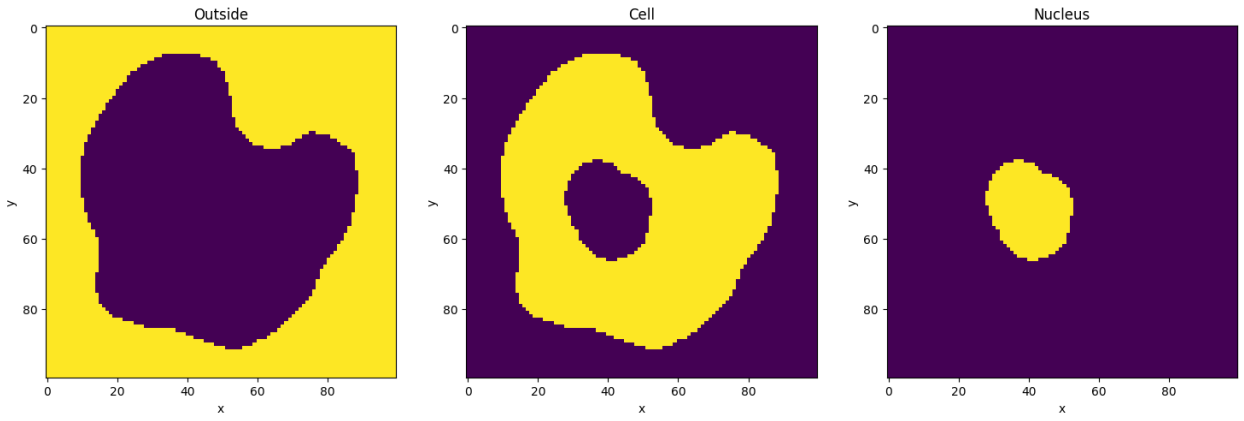
Get compartment by list index

[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[0], 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'
```

11.1.5 Editing model contents

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

Names

[12]:
```
print(f"Model name: {my_model.name}")
```

```
Model name: Very Simple Model
```

[13]:
```
my_model.name = "New model name!"
print(f"Model name: {my_model.name}")
```

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

Model Parameters

[14]:
```
param = my_model.parameters[0]
print(f"{param.name} = {param.value}")
```

```
param = 1
```

[15]:
```
param.value = "2.5"
print(f"{param.name} = {param.value}")
```

```
param = 2.5
```

Reaction Parameters

[16]:
```
k1 = my_model.compartments["Nucleus"].reactions["A to B conversion"].parameters["k1"]
print(f"{k1.name} = {k1.value}")
```

```
k1 = 0.3
```

[17]:
```
k1.value = 0.72
print(f"{k1.name} = {k1.value}")
```

```
k1 = 0.72
```

Species Initial Concentrations

- can be Uniform (`float`), Analytic (`str`) or Image (`np.ndarray`)

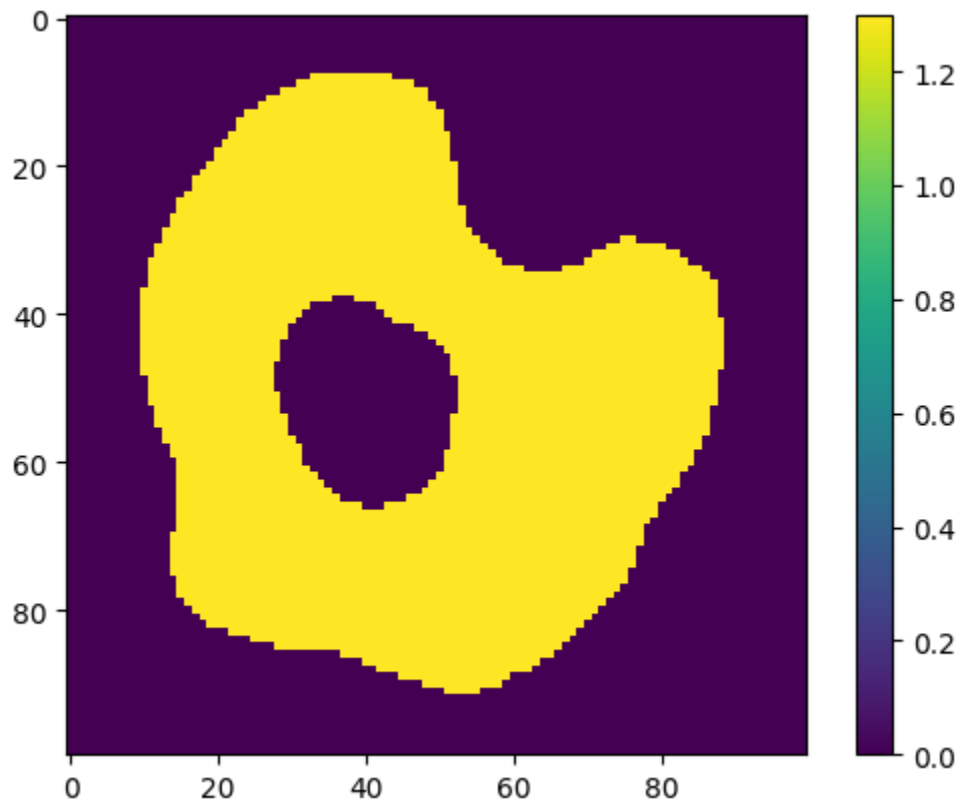
[18]:
```
species = my_model.compartments["Cell"].species[0]
print(
    f"Species '{species.name}' has initial concentration of type '{species.concentration_
↪type}', value '{species.uniform_concentration}'"
)
species.uniform_concentration = 1.3
print(
```

(continues on next page)

(continued from previous page)

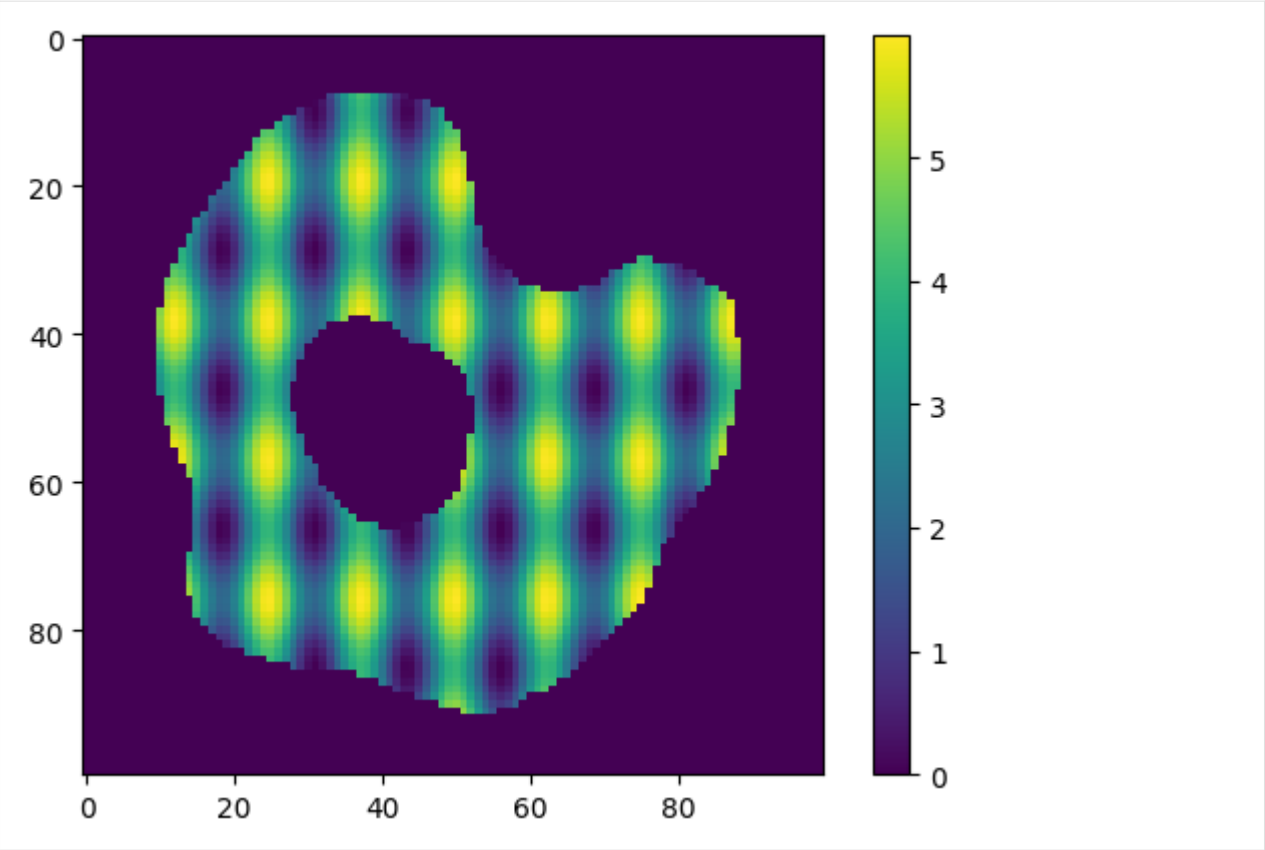
```
    f"Species '{species.name}' has initial concentration of type '{species.concentration_
↪type}', value '{species.uniform_concentration}'"
)
plt.imshow(species.concentration_image[0])
plt.colorbar()
plt.show()
```

```
Species 'A_cell' has initial concentration of type 'ConcentrationType.Uniform', value '0.
↪0'
Species 'A_cell' has initial concentration of type 'ConcentrationType.Uniform', value '1.
↪3'
```



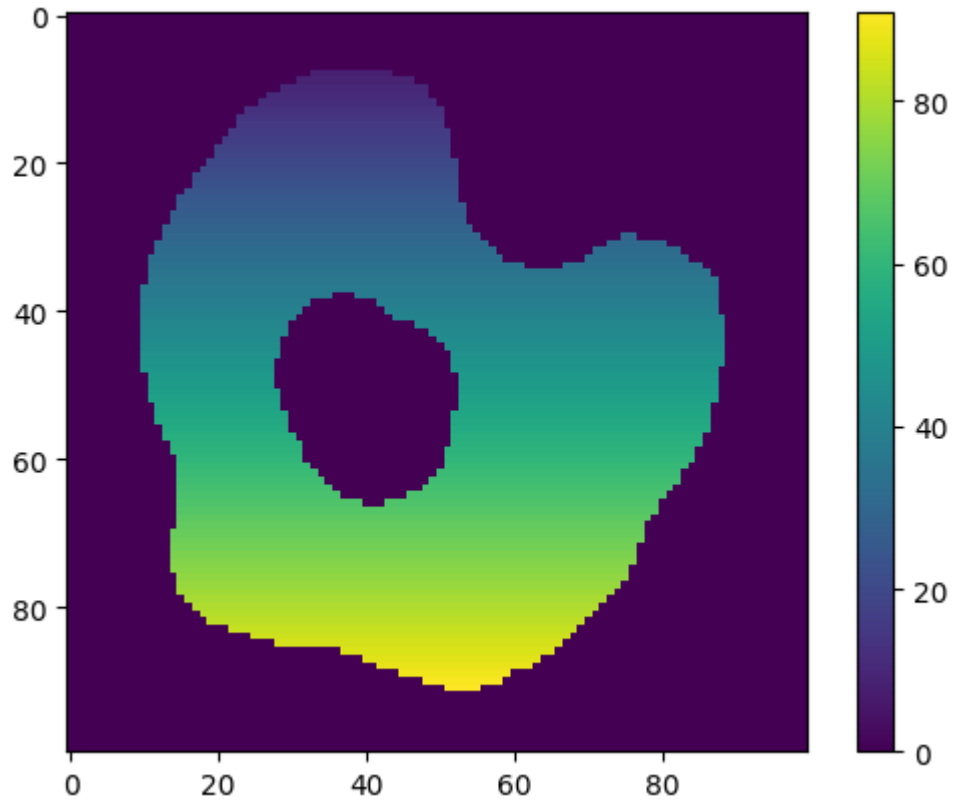
[19]:
```
species.analytic_concentration = "3 + 2*cos(x/2)+sin(y/3)"
print(
    f"Species '{species.name}' has initial concentration of type '{species.concentration_
↪type}', expression '{species.analytic_concentration}'"
)
plt.imshow(species.concentration_image[0])
plt.colorbar()
plt.show()
```

```
Species 'A_cell' has initial concentration of type 'ConcentrationType.Analytic',␣
↪expression '3 + 2 * cos(x / 2) + sin(y / 3)'
```



[20]:
```
# generate concentration image with concentration = x + y
new_concentration_image = np.zeros(species.concentration_image.shape)
for index, _ in np.ndenumerate(new_concentration_image):
    new_concentration_image[index] = index[0] + index[1]

species.concentration_image = new_concentration_image
print(
    f"Species '{species.name}' has initial concentration of type '{species.concentration_
↪type}'"
)
plt.imshow(species.concentration_image[0])
plt.colorbar()
plt.show()
```

```
Species 'A_cell' has initial concentration of type 'ConcentrationType.Image'
```



11.1.6 Exporting a model

- to save the model, including any simulation results: `model.export_sme_file('model_filename.sme')`
- to export the model as an SBML file (no simulation results): `model.export_sbml_file('model_filename.xml')`

[21]:
```
my_model.export_sme_file("model.sme")
```

[ ]:
```

```

Interactive online version

11.2 Simulating

11.2.1 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()
```

11.2.2 Running a simulation

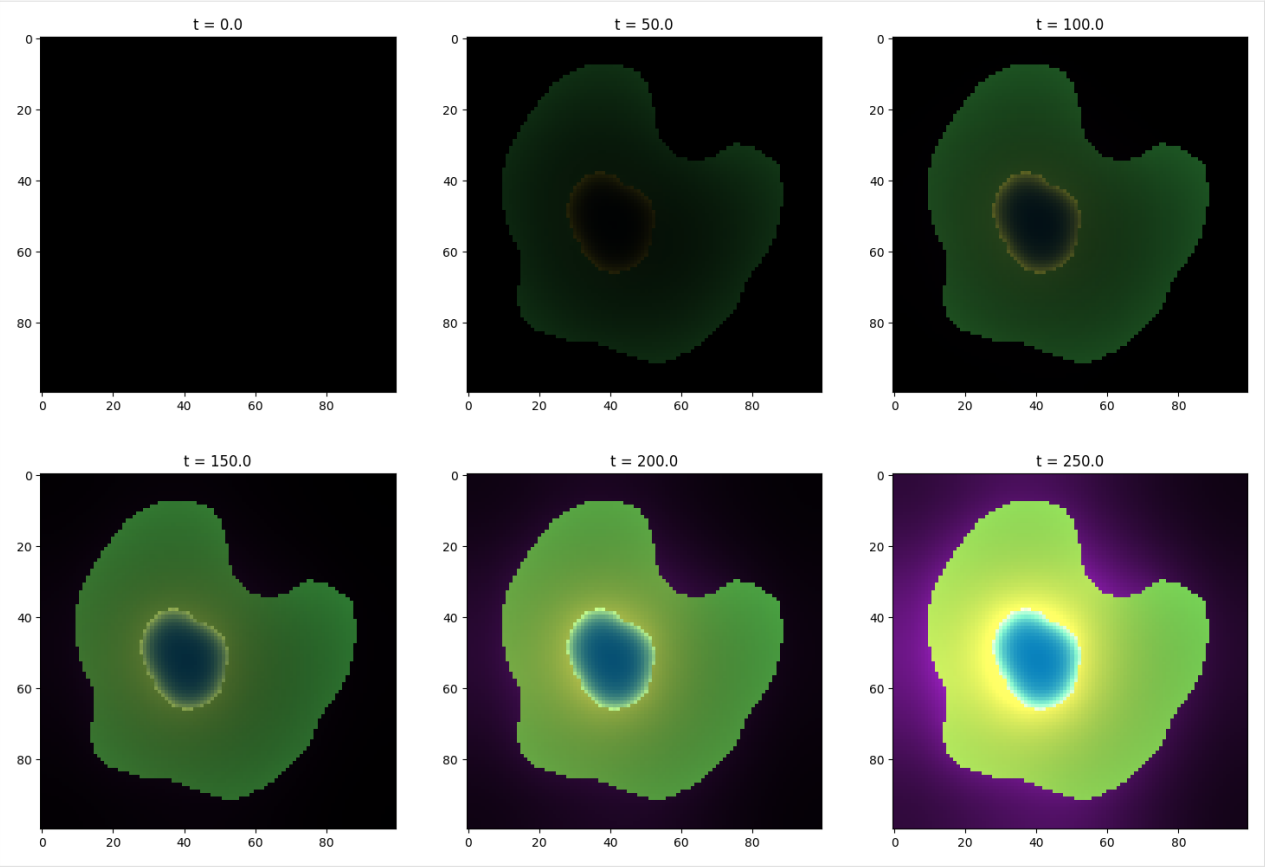
- models can be simulated by specifying the total simulation time, and the interval between images
- the simulation returns a list of `SimulationResult` objects, each of which contains
  - `time_point`: the time point
  - `concentration_image`: an image of the species concentrations at this time point
  - `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)
print(sim_results[0])
print(sim_results[0].species_concentration.keys())
```

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

dict_keys(['B_out', 'A_cell', 'B_cell', 'A_nucl', 'B_nucl'])
```

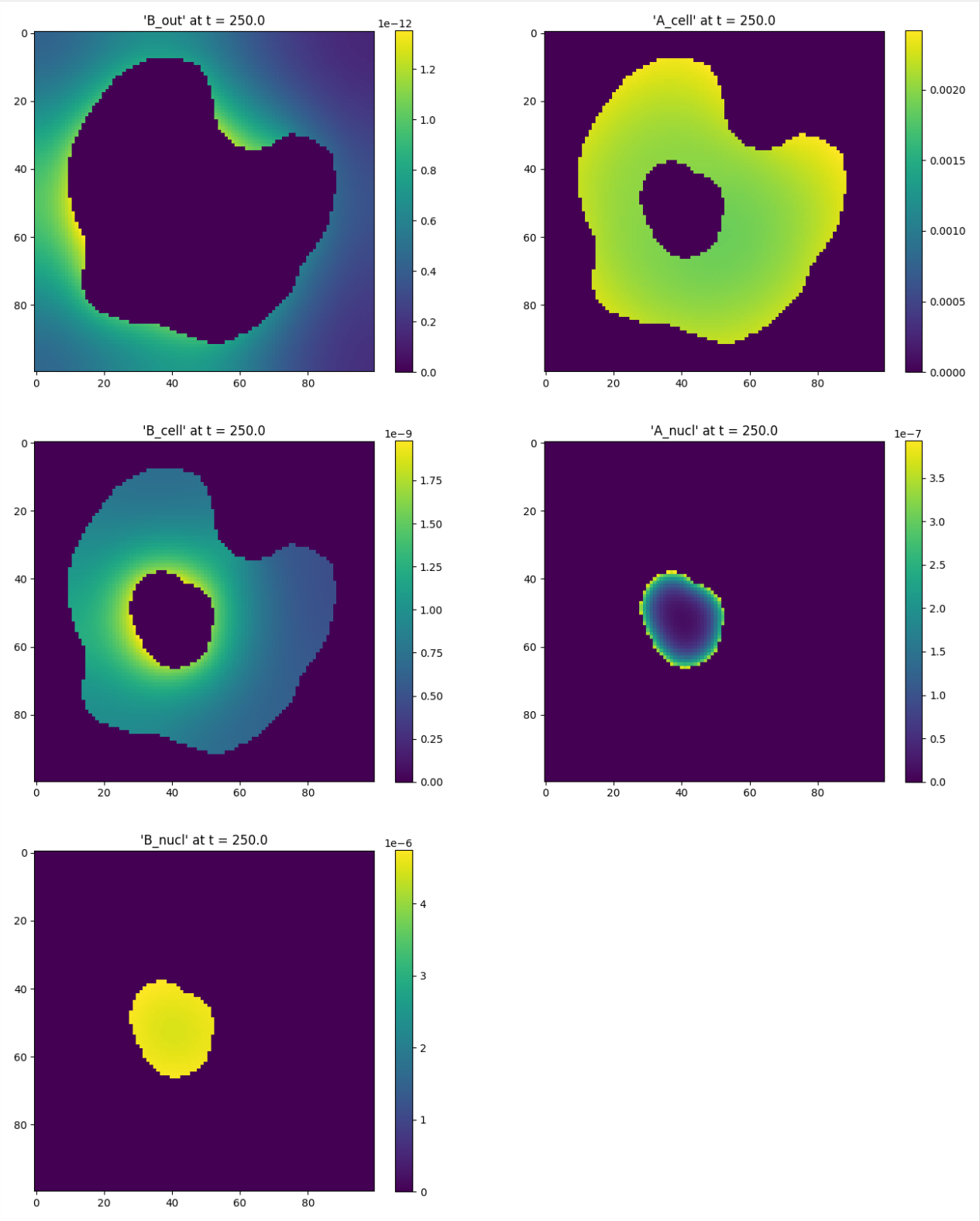
Display images from simulation results

[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[0])
    ax.set_title(f"t = {res.time_point}")
plt.show()
```



Plot concentrations from simulation results

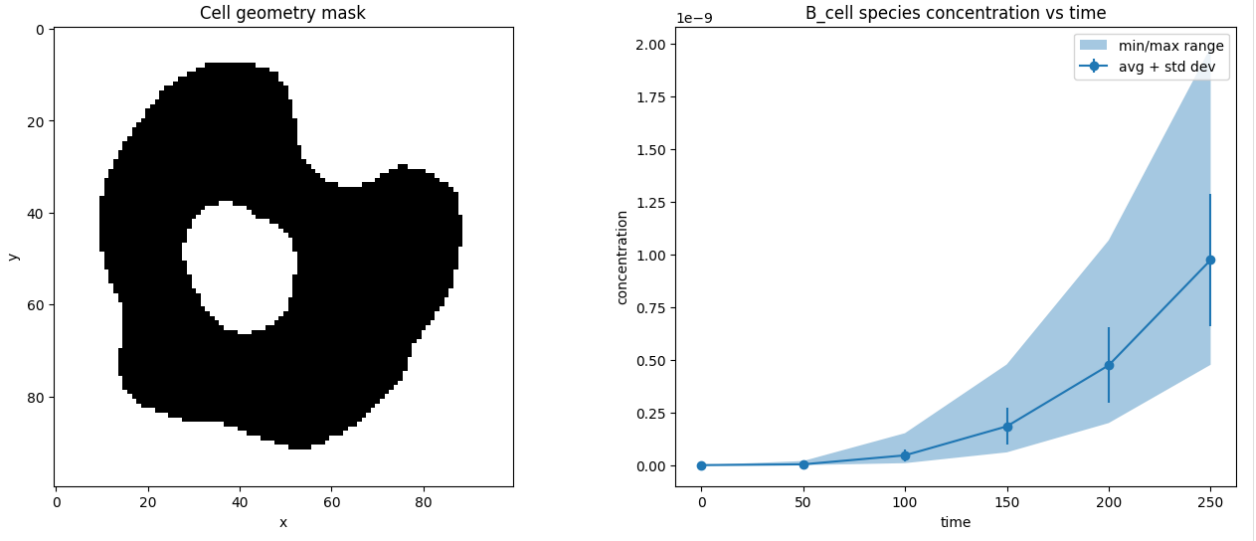
[5]:
```
result = sim_results[5]
fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(16, 20))
fig.delaxes(axs[2, 1])
for ax, (species, concentration) in zip(fig.axes, result.species_concentration.items()):
    im = ax.imshow(concentration[0])
    ax.set_title(f"'{species}' at t = {result.time_point}")
    fig.colorbar(im, ax=ax)
plt.show()
```


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

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.

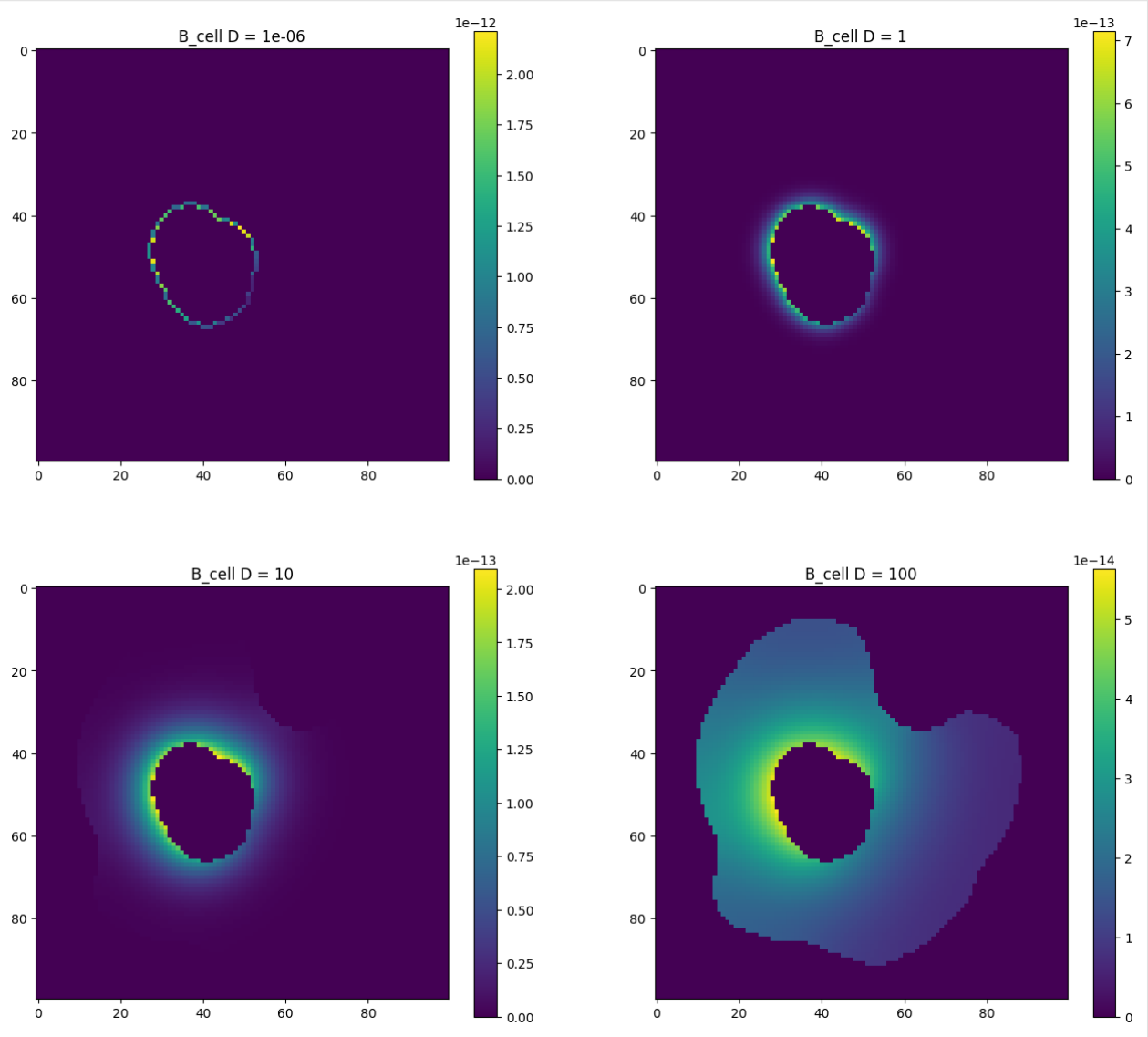
[6]:
```
fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, figsize=(16, 6))

# get mask of compartment pixels
mask = my_model.compartments["Cell"].geometry_mask
ax0.imshow(mask[0], 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 = [r.time_point for r in sim_results]
concs = [r.species_concentration["B_cell"][mask].flatten() for r in sim_results]

# 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()
```



11.2.3 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"][0])
    ax.set_title(f"B_cell D = {diffconst}")
    fig.colorbar(im, ax=ax)
plt.show()
```



[ ]:

Interactive online version

11.3 Visualization

11.3.1 Open an example model & simulate

[1]:
```
!pip install -q sme
import sme
from matplotlib import pyplot as plt
from matplotlib import animation
import numpy as np
from IPython.display import HTML
```

[2]:
```
my_model = sme.open_example_model()
sim_results = my_model.simulate(simulation_time=250.0, image_interval=5.0)
```

11.3.2 Animation of species concentrations

[3]:
```
species = ["B_cell", "B_out"]

fig, axs = plt.subplots(constrained_layout=True, ncols=len(species), figsize=(9, 4))

# set normalization of each plot to maximum concentration of species over entire␣
↪simulation
norms = []
for spec, ax in zip(species, axs):
    c_max = np.max([np.max(r.species_concentration[spec]) for r in sim_results])
    norms.append(plt.Normalize(vmin=0, vmax=c_max))

# make a plot with correctly normalized colorbar for each species
for ax, norm in zip(axs, norms):
    im = ax.imshow(np.zeros((1, 1)), norm=norm)
    fig.colorbar(im, ax=ax)

# create a list of plot artists for each timepoint
artists = []
for sim_result in sim_results:
    artist = []
    for spec, ax, norm in zip(species, axs, norms):
        artist.append(
            ax.imshow(
                sim_result.species_concentration[spec][0],
                animated=True,
                norm=norm,
                interpolation=None,
            )
        )
        artist.append(
            ax.text(
```

(continues on next page)

(continued from previous page)

```
                0.5,
                1.01,
                f"{spec}: t = {sim_result.time_point}",
                horizontalalignment="center",
                verticalalignment="bottom",
                transform=ax.transAxes,
            )
        )
    artists.append(artist)

# make an animation from the list of artists
anim = animation.ArtistAnimation(fig, artists, interval=200, blit=True, repeat=False)
plt.close()
HTML(anim.to_html5_video())
```

[3]:
```
<IPython.core.display.HTML object>
```

11.3.3 Gray-Scott Model

[4]:
```
# simulate for a few different reaction parameter values & store concentrations of␣
↪species V
gray_scott = sme.open_example_model("gray-scott")
fs = ["0.03", "0.04", "0.05"]
vs = []
for f in fs:
    gray_scott.parameters["f"].value = f
    sim_results = gray_scott.simulate(
        simulation_time=10000.0,
        image_interval=100,
        simulator_type=sme.SimulatorType.Pixel,
    )
    times = [r.time_point for r in sim_results]
    vs.append([r.species_concentration["V"][0] for r in sim_results])
```

[5]:
```
fig, axs = plt.subplots(constrained_layout=True, nrows=len(fs), figsize=(9, 16))
for ax, f in zip(axs, fs):
    ax.set_title(f"f = {f}")

artists = []
for i in range(len(vs[0])):
    artist = []
    for ax, v in zip(axs, vs):
        artist.append(ax.imshow(v[i], animated=True, interpolation=None))
    artists.append(artist)

anim = animation.ArtistAnimation(fig, artists, interval=200, blit=True, repeat=False)
plt.close()
HTML(anim.to_html5_video())
```

[5]:
```
<IPython.core.display.HTML object>
```

[ ]:

CHAPTER

TWELVE

API REFERENCE

sme	Spatial Model Editor Python interface

12.1 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/

Classes

Compartment	a compartment where species live
CompartmentList	a list of *Compartment* objects
ConcentrationType(value[, names, module, ...])	
Membrane	a membrane where two compartments meet
MembraneList	a list of *Membrane* objects
Model	the spatial model
Parameter	a parameter of the model
ParameterList	a list of *Parameter* objects
Reaction	a reaction between species
ReactionList	a list of *Reaction* objects
ReactionParameter	a parameter of a reaction
ReactionParameterList	a list of *ReactionParameter* objects
SimulationResult	results at a single timepoint of a simulation
SimulationResultList	a list of *SimulationResult* objects
SimulatorType(value[, names, module, ...])	
Species	a species that lives in a compartment
SpeciesList	a list of *Species* objects

12.1.1 sme.Compartment

class sme.**Compartment**

a compartment where species live

Methods

__init__(*args, **kwargs)	

sme.Compartment.__init__

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

Attributes

geometry_mask	a voxel 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**

a voxel mask of the compartment geometry

An array of boolean values, where `geometry_mask[z][y][x] = True` if the voxel at point (x,y,z) is part of this compartment

Examples

the mask is a 3d (depth x height x width) array of bool:

```
>>> import sme
>>> model = sme.open_example_model()
>>> mask = model.compartments['Cell'].geometry_mask
>>> type(mask)
<class 'numpy.ndarray'>
>>> mask.dtype
dtype('bool')
>>> mask.shape
(1, 100, 100)
```

the first z-slice of the mask can be displayed using matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> imgplot = plt.imshow(mask[0], interpolation='none')
```

Type
numpy.ndarray

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

12.1.2 sme.CompartmentList

class sme.**CompartmentList**

a list of *Compartment* objects

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

12.1.3 sme.ConcentrationType

class sme.**ConcentrationType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Attributes

Uniform
Analytic
Image

sme.ConcentrationType.Uniform

ConcentrationType.**Uniform** = **0**

sme.ConcentrationType.Analytic

ConcentrationType.**Analytic** = **1**

sme.ConcentrationType.Image

ConcentrationType.**Image** = **2**

12.1.4 sme.Membrane

class sme.**Membrane**

a membrane where two compartments meet

Methods

__init__(*args, **kwargs)	

sme.Membrane.__init__

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

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

12.1.5 sme.MembraneList

class sme.**MembraneList**

a list of *Membrane* objects

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

12.1.6 sme.Model

class sme.**Model**

the spatial model

Attributes

compartment_image	an image of the compartments in this model
compartments	the compartments in this model
export_sbml_file	exports the model as a spatial SBML file
export_sme_file	exports the model as a sme file
import_geometry_from_image	sets the geometry of each compartment to the corresponding pixels in the supplied geometry image
membranes	the membranes in this model
name	the name of this model
parameters	the parameters in this model
simulate	Overloaded function.
simulation_results	returns the simulation results.

sme.Model.compartment_image

property Model.**compartment_image**

an image of the compartments in this model

An array of RGB integer values for each voxel in the image of the compartments in this model, which can be displayed using e.g. `matplotlib.pyplot.imshow`

Examples

the image is a 4d (depth x height x width x 3) array of integers:

```
>>> import sme
>>> model = sme.open_example_model()
>>> type(model.compartment_image)
<class 'numpy.ndarray'>
>>> model.compartment_image.dtype
dtype('uint8')
>>> model.compartment_image.shape
(1, 100, 100, 3)
```

each voxel in the image has a triplet of RGB integer values in the range 0-255:

```
>>> model.compartment_image[0, 34, 36]
array([144,  97, 193], dtype=uint8)
```

the first z-slice of the image can be displayed using matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> imgplot = plt.imshow(model.compartment_image[0])
```

Type
numpy.ndarray

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.export_sbml_file

`Model.`**`export_sbml_file`**

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`**

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`**

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 volume 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.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

sme.Model.simulate

`Model.`**`simulate`**

Overloaded function.

1. `simulate(self, simulation_time: float, image_interval: float, timeout_seconds: int = 86400, throw_on_timeout: bool = True, simulator_type: sme.SimulatorType = SimulatorType.Pixel, continue_existing_simulation: bool = False, return_results: bool = True, n_threads: int = 1) -> sme.SimulationResultList`

   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. return_results (bool): Whether to return the simulation results. Default value: *True*. If *False*, an empty SimulationResultList is returned. n_threads(int): Number of cpu threads to use (for Pixel simulations). Default value is 1, 0 means use all available threads.

   Returns:
   SimulationResultList: the results of the simulation

   Raises:
   RuntimeError: if the simulation times out or fails

2. `simulate(self, simulation_times: str, image_intervals: str, timeout_seconds: int = 86400, throw_on_timeout: bool = True, simulator_type: sme.SimulatorType = SimulatorType.Pixel, continue_existing_simulation: bool = False, return_results: bool = True, n_threads: int = 1) -> sme.SimulationResultList`

   returns the results of the simulation.

   Args:
   simulation_times (str): The length(s) of the simulation in model units of time as a

semicolon-delimited list, e.g. *"5"*, or *"10;100;20"* image_intervals (str): The interval(s) between images in model units of time as a semicolon-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. return_results (bool): Whether to return the simulation results. Default value: *True*. If *False*, an empty SimulationResultList is returned. n_threads(int): Number of cpu threads to use (for Pixel simulations). Default value is 1, 0 means use all available threads.

Returns:
SimulationResultList: the results of the simulation

Raises:
RuntimeError: if the simulation times out or fails

sme.Model.simulation_results

`Model.`**`simulation_results`**

returns the simulation results.

Returns
the simulation results

Return type
SimulationResultList

12.1.7 sme.Parameter

`class` `sme.`**`Parameter`**

a parameter of the model

Methods

__init__(*args, **kwargs)

sme.Parameter.__init__

`Parameter.`**`__init__`**(**args*, ***kwargs*)

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

12.1.8 sme.ParameterList

class sme.**ParameterList**

a list of *Parameter* objects

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

12.1.9 sme.Reaction

class sme.**Reaction**

a reaction between species

Methods

__init__(*args, **kwargs)

sme.Reaction.__init__

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

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

12.1.10 sme.ReactionList

class sme.**ReactionList**

a list of *Reaction* objects

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

12.1.11 sme.ReactionParameter

class sme.**ReactionParameter**

a parameter of a reaction

Methods

__init__(*args, **kwargs)	

sme.ReactionParameter.__init__

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

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

12.1.12 sme.ReactionParameterList

class sme.**ReactionParameterList**

a list of *ReactionParameter* objects

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

12.1.13 sme.SimulationResult

class sme.**SimulationResult**

results at a single timepoint of a simulation

Methods

__init__(*args, **kwargs)

sme.SimulationResult.__init__

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

Attributes

concentration_image	an image of the species concentrations at this timepoint
species_concentration	the species concentrations at this timepoint
species_dcdt	the species concentration rate of change 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

An array of RGB integer values for each voxel in the image of the compartments in this model, which can be displayed using e.g. `matplotlib.pyplot.imshow`

Examples

do a short simulation and get the concentration image from the last timepoint:

```
>>> import sme
>>> model = sme.open_example_model()
>>> results = model.simulate(10, 1)
>>> concentration_image = results[-1].concentration_image
```

the image is a 4d (depth x height x width x 3) array of integers:

```
>>> type(concentration_image)
<class 'numpy.ndarray'>
>>> concentration_image.dtype
dtype('uint8')
>>> concentration_image.shape
(1, 100, 100, 3)
```

each voxel in the image has a triplet of RGB integer values in the range 0-255:

```
>>> concentration_image[0, 34, 36]
array([33, 23,  9], dtype=uint8)
```

the image can be displayed using matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> imgplot = plt.imshow(concentration_image[0])
```

Type
numpy.ndarray

sme.SimulationResult.species_concentration

property SimulationResult.**species_concentration**

the species concentrations at this timepoint

for each species, the concentrations are provided as a 3d array, where `species_concentration['A'][z][y][x]` is the concentration of species "A" at the point (x,y,z)

Examples

do a short simulation and get the species concentrations from the last timepoint:

```
>>> import sme
>>> model = sme.open_example_model()
>>> results = model.simulate(10, 1)
>>> species_concentration = results[-1].species_concentration
```

this is a dict with an entry for each species:

```
>>> type(species_concentration)
<class 'dict'>
>>> species_concentration.keys()
dict_keys(['B_out', 'A_cell', 'B_cell', 'A_nucl', 'B_nucl'])
```

the concentrations are a 3d ndarray of doubles, one for each pixel in the geometry image:

```
>>> b_cell = species_concentration['B_cell']
>>> type(b_cell)
<class 'numpy.ndarray'>
>>> b_cell.dtype
dtype('float64')
>>> b_cell.shape
(1, 100, 100)
```

the concentrations can be displayed using matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> imgplot = plt.imshow(b_cell[0])
```

Type
Dict[str, numpy.ndarray]

sme.SimulationResult.species_dcdt

property SimulationResult.**species_dcdt**

the species concentration rate of change at this timepoint

for each species, the rate of change of concentration is provided as a 3d array, where `species_dcdt['A'][z][y][x]` is the rate of change of the concentration of species "A" at the point (x,y,z)

Note: The rate of change of species concentrations is only provided for the last timepoint of a simulation, and only when using the Pixel simulator. Otherwise `species_dcdt` is an empty dict.

Examples

do a short Pixel simulation and get the rate of change of species concentrations from the last timepoint:

```
>>> import sme
>>> model = sme.open_example_model()
>>> results = model.simulate(10, 1)
>>> species_dcdt = results[-1].species_dcdt
```

this is a dict with an entry for each species:

```
>>> type(species_dcdt)
<class 'dict'>
>>> species_dcdt.keys()
dict_keys(['B_out', 'A_cell', 'B_cell', 'A_nucl', 'B_nucl'])
```

the rate of change of concentration is a 2d ndarray of doubles, one for each pixel in the geometry image:

```
>>> b_cell = species_dcdt['B_cell']
>>> type(b_cell)
<class 'numpy.ndarray'>
>>> b_cell.dtype
dtype('float64')
>>> b_cell.shape
(1, 100, 100)
```

the rate of change of concentration can be displayed using matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> imgplot = plt.imshow(b_cell[0])
```

Type
Dict[str, numpy.ndarray]

sme.SimulationResult.time_point

property SimulationResult.**time_point**

the timepoint these simulation results are from

Type
float

12.1.14 sme.SimulationResultList

class sme.**SimulationResultList**

a list of *SimulationResult* objects

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

12.1.15 sme.SimulatorType

class sme.**SimulatorType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Attributes

DUNE
Pixel

sme.SimulatorType.DUNE

SimulatorType.**DUNE = 0**

sme.SimulatorType.Pixel

SimulatorType.**Pixel = 1**

12.1.16 sme.Species

class sme.**Species**

a species that lives in a compartment

Methods

__init__(*args, **kwargs)

sme.Species.__init__

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

Attributes

analytic_concentration	the initial concentration of this species as an analytic_2d expression
concentration_image	the initial concentration of this species as a 3d array of floats, one for each voxel in the geometry image
concentration_type	the type of initial concentration of this species (Uniform, Analytic or Image)
diffusion_constant	the diffusion constant of this species
name	the name of this species
uniform_concentration	the uniform initial concentration of this species as a float

sme.Species.analytic_concentration

property Species.**analytic_concentration**

the initial concentration of this species as an analytic_2d expression

Type
str

sme.Species.concentration_image

property Species.**concentration_image**

the initial concentration of this species as a 3d array of floats, one for each voxel in the geometry image

Type
np.ndarray(float)

sme.Species.concentration_type

property Species.**concentration_type**

the type of initial concentration of this species (Uniform, Analytic or Image)

Type
Species.ConcentrationType

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

sme.Species.uniform_concentration

property Species.**uniform_concentration**

the uniform initial concentration of this species as a float

Type
float

12.1.17 sme.SpeciesList

class sme.**SpeciesList**

a list of *Species* objects

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

CHAPTER

THIRTEEN

DUNE-COPASI SIMULATOR

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

13.1 Simulation options

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

- **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**

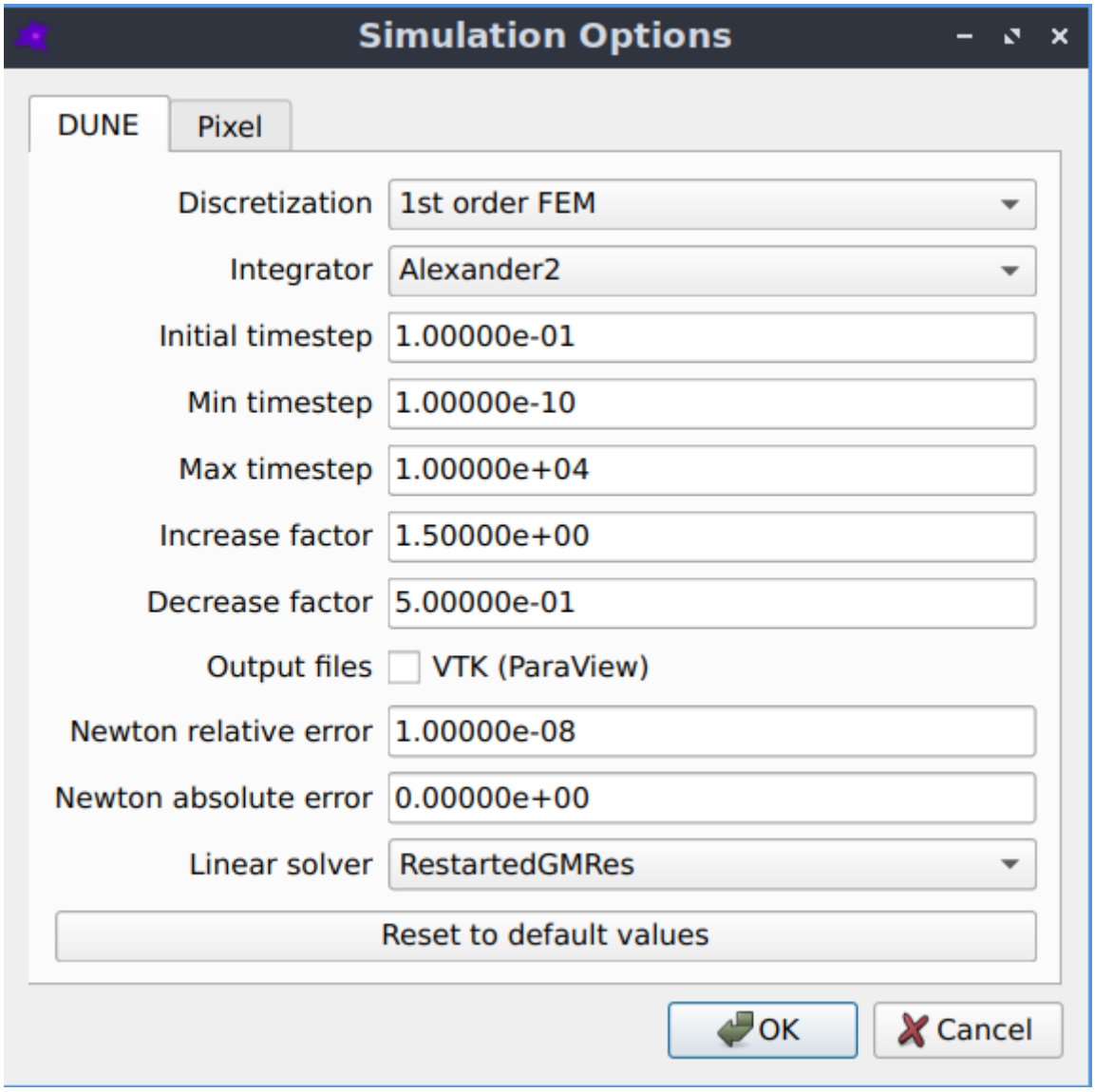


Fig. 1: The simulation options that can be used to fine tune-the dune-copasi solver.

  - 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

- **Newton relative error**
  - the relative error where Newton iteration is considered to have converged
  - currently this may need to be altered depending on the units and geometry size (see #315)

- **Newton absolute error**
  - the absolute error where Newton iteration is considered to have converged
  - currently this may need to be altered depending on the units and geometry size (see #315)

- **Linear solver**
  - a variety of iterative and direct solvers are available
  - the default is RestartedGMRes

For more information see the dune-copasi documentation.

CHAPTER

FOURTEEN

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.

14.1 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**

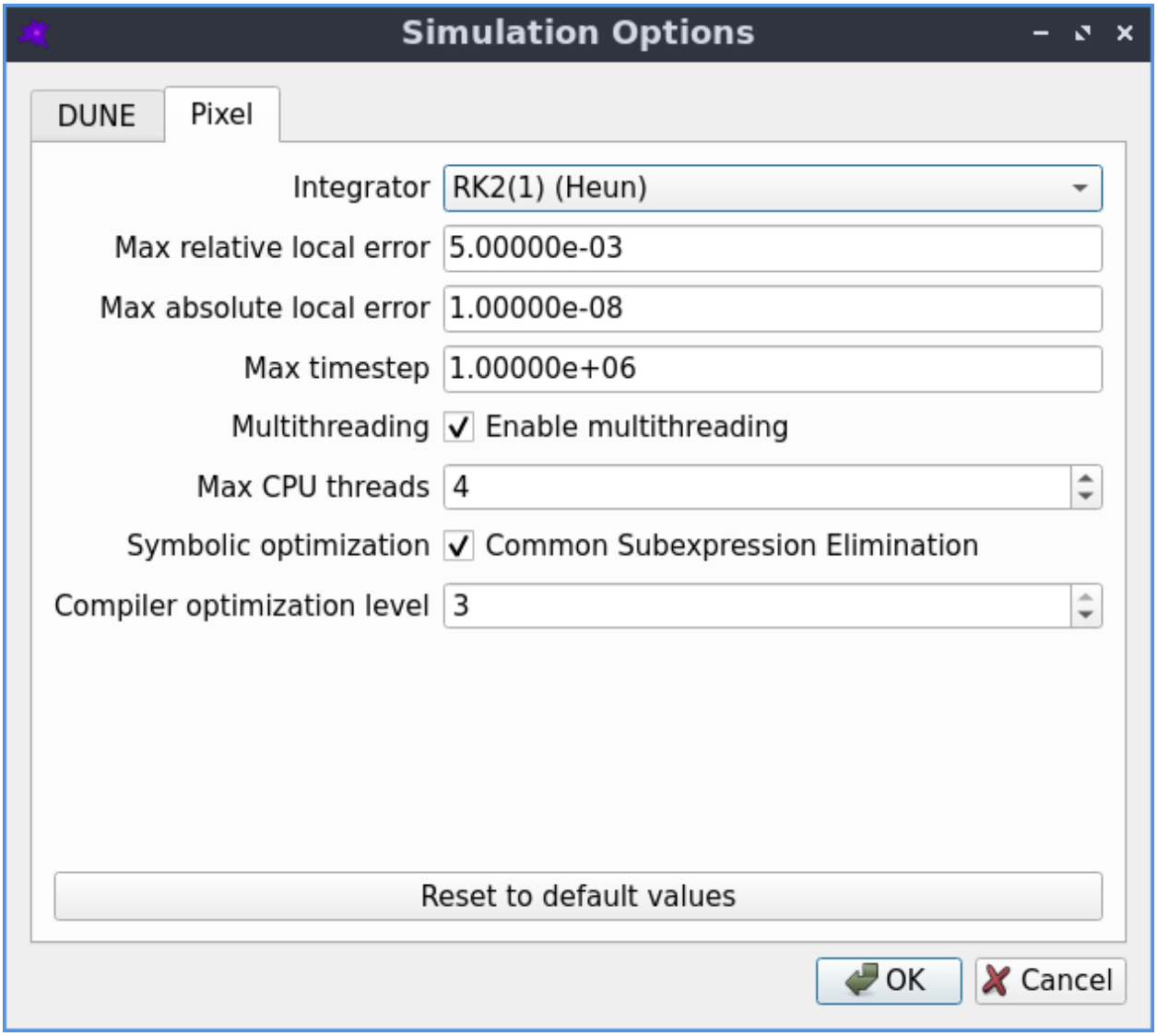


Fig. 1: The simulation options that can be used to fine tune-the Pixel solver.

  - 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

14.2 Spatial discretization

Space is discretized using a linear grid with spacing δx, δy, δz. The concentration is defined as a 3d array of values $c_{i,j,k}$, where the value with index (i, j, k) corresponds to the concentration at the spatial point $(x = i\delta x, y = j\delta y, z = k\delta z)$.

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

$$to$$

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right) c_{i,j,k} =$$
$$+ \left[c_{i+1,j,k} + c_{i-1,j,k}\right] / (\delta x^2)$$
$$+ \left[c_{i,j+1,k} + c_{i,j-1,k}\right] / (\delta y^2)$$
$$+ \left[c_{i,j,k+1} + c_{i,j,k-1}\right] / (\delta z^2)$$
$$- \left[\frac{1}{\delta x^2} + \frac{1}{\delta y^2} + \frac{1}{\delta z^2}\right] 2c_{i,j,k}$$
$$+\mathcal{O}(\delta x^2) + \mathcal{O}(\delta y^2) + \mathcal{O}(\delta z^2) \quad (14.1)$$

$$=$$
$$+ \left[c_{i+1,j,k} + c_{i-1,j,k}\right] / (\delta x^2)$$
$$+ \left[c_{i,j+1,k} + c_{i,j-1,k}\right] / (\delta y^2)$$
$$+ \left[c_{i,j,k+1} + c_{i,j,k-1}\right] / (\delta z^2)$$

$$-\left[\frac{1}{\delta x^2}+\frac{1}{\delta y^2}+\frac{1}{\delta z^2}\right]2c_{i,j,k}$$
$$+\mathcal{O}(\delta x^2)+\mathcal{O}(\delta y^2)+\mathcal{O}(\delta z^2)$$

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

14.3 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

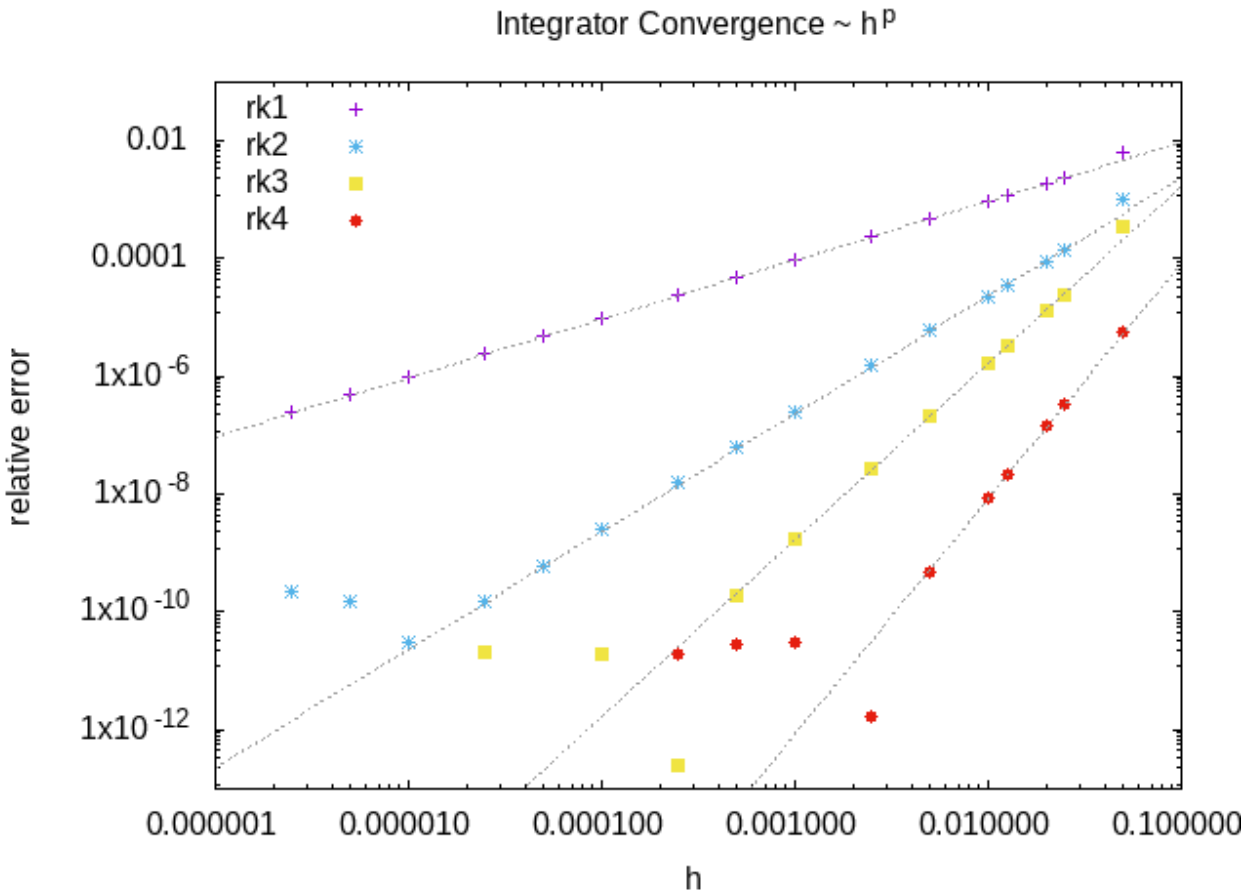


Fig. 2: An example of the convergence of the included RK integrators: relative error of the solution at a particular pixel as a function of the stepsize.

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

14.4 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 divide 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

14.5 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 (see p125 of https://doi.org/10.1017/CBO9780511781438), above which the system becomes unstable:

$$\delta t \leq \frac{1}{2D\left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2} + \frac{1}{\delta z^2}\right)}$$

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.

14.6 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.

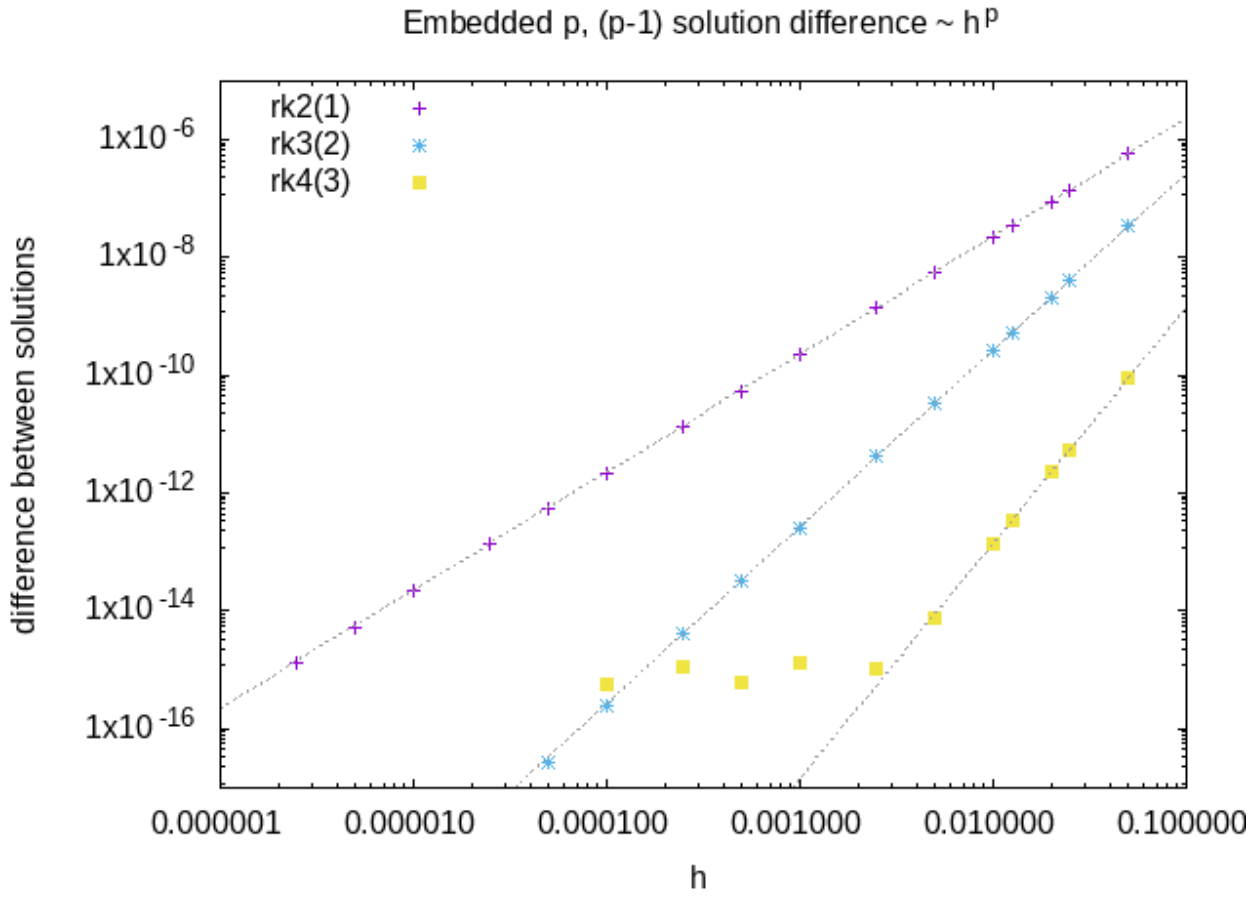


Fig. 3: An example of the difference between order p and order p-1 solutions from embedded schemes as a function of the stepsize. This quantity is a measure of the local integration error, and scales like h^p

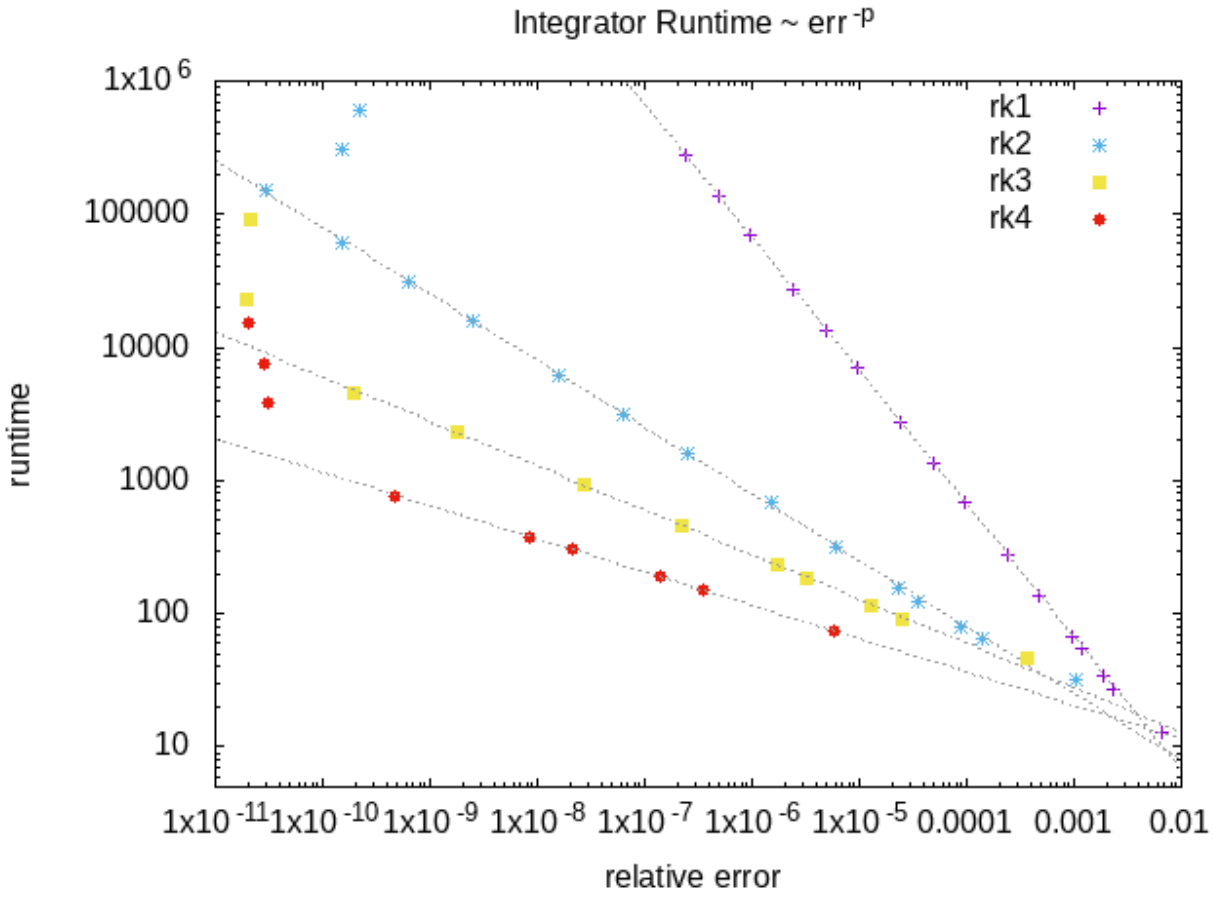


Fig. 4: An example of the runtime of the RK integrators as a function of the relative error on the final solution. The higher order integrators offer better performance if a very accurate solution is required, but at lower accuracy the lower order integrators are much faster.

14.6.1 Compartments

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

14.6.2 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.

14.6.3 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

FIFTEEN

PARAMETER OPTIMIZATION

To start parameter optimization, click on *Tools->Optimization*, or use the keyboard shortcut *Ctrl+P*.

The parameter optimization interface displays a history of the best set of parameters and their fitness, along with images of the optimization targets (species concentrations or concentration rate of changes) compared with the resulting values from the current best set of parameters.

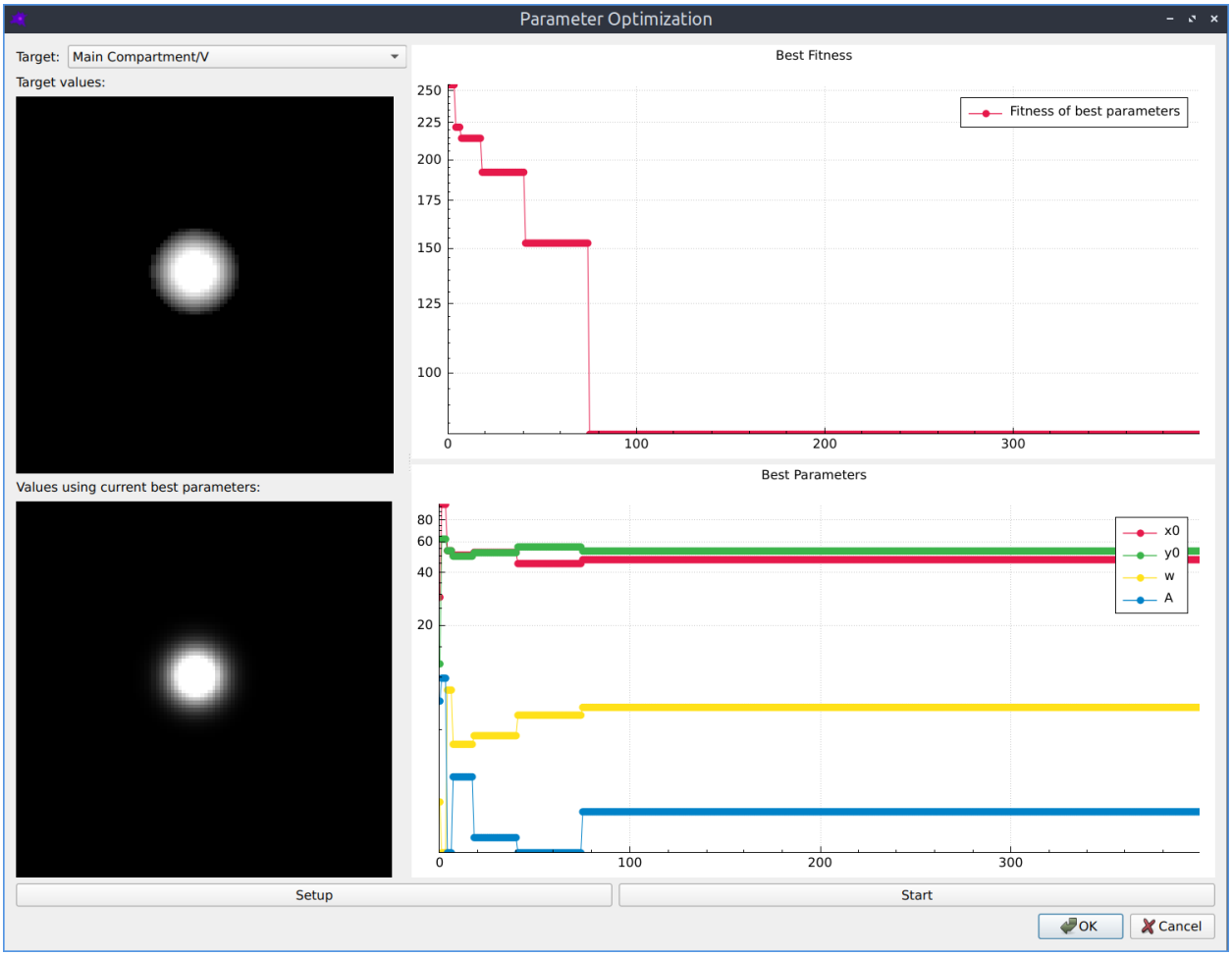


Fig. 1: The parameter optimization interface.

- **Targets and results (left side)**
  - If you have multiple targets, you can choose which one to display from the dropdown list

  - The upper image shows the target concentration (or concentration rate of change) values
  - The lower image shows the actual values from the simulation using the current best set of parameters

- **Fitness and parameters (right side)**
  - The upper plot shows the history of fitness values during the optimization (lower is better)
  - The lower plot shows the history of the best set of parameters for at each optimization iteration

- **Setup**
  - Click the *Setup* button to change the optimization settings
  - See the *Optimization Setup* section for more details

- **Start**
  - Click the *Start* button to start optimizing
  - You can stop and continue optimization

When you are happy with your parameters you can press *Stop* and then *OK*. You will then be asked if you want to apply the best parameters found during optimization to your model. The optimization settings (but not any existing optimization history or results) are also saved in the model.

15.1 Optimization Setup

- **Algorithm**
  - Choose which optimization algorithm to use
  - See the Pagmo algorithms documentation for more details about the available algorithms

- **Threads**
  - The number of populations to evolve in parallel
  - Typically this would be equal to the number of available CPU cores

- **Population**
  - The population size to evolve on each thread
  - Typically the more parameters are being optimized the larger this should be

- **Targets**
  - A list of target concentrations (or rate of change of concentrations) to optimize for
  - See the *Optimization Target* section for more details

- **Parameters**
  - A list of parameters to optimize
  - See the *Optimization Parameter* section for more details

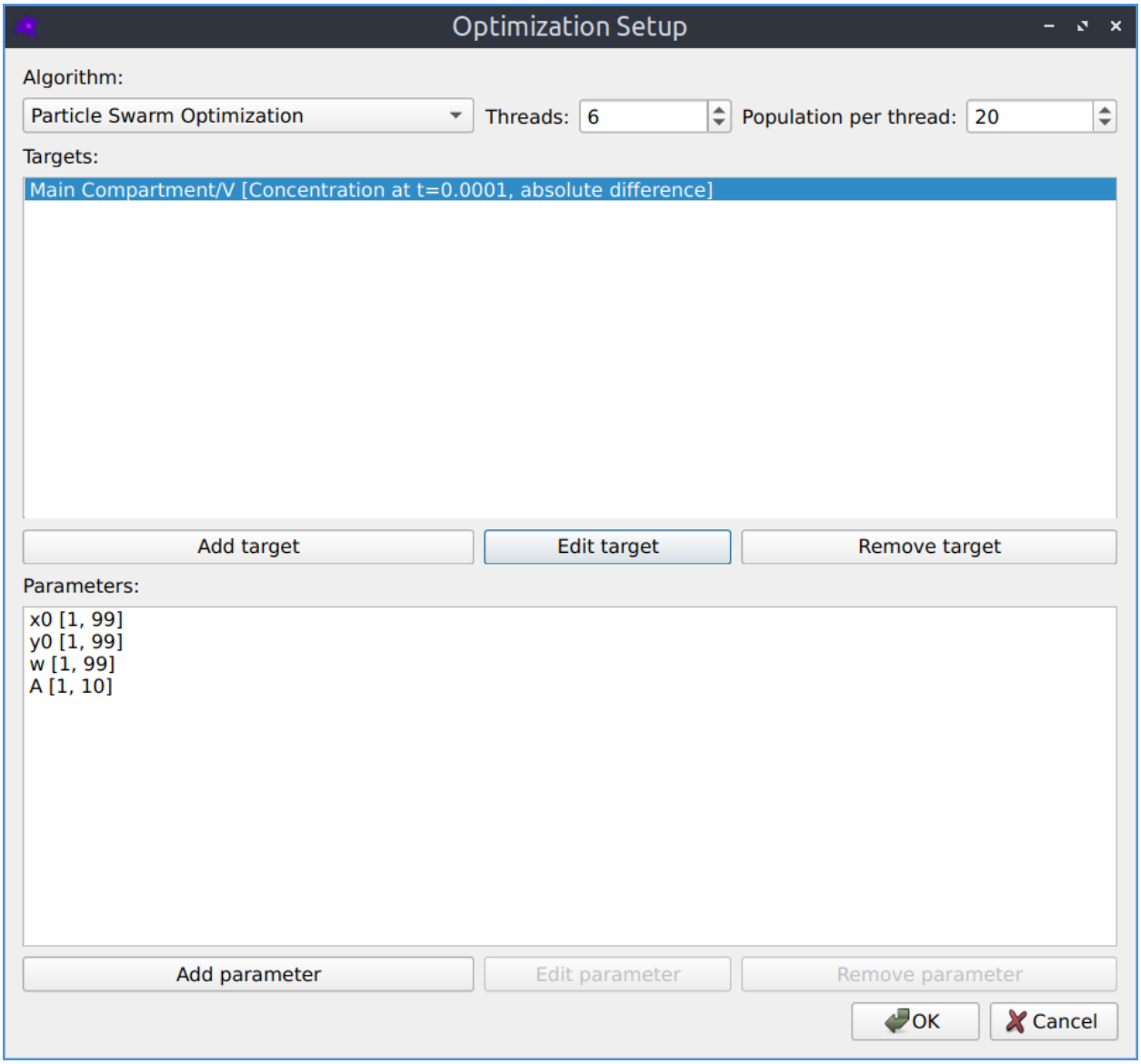


Fig. 2: Parameter optimization setup.

15.2 Optimization Parameter

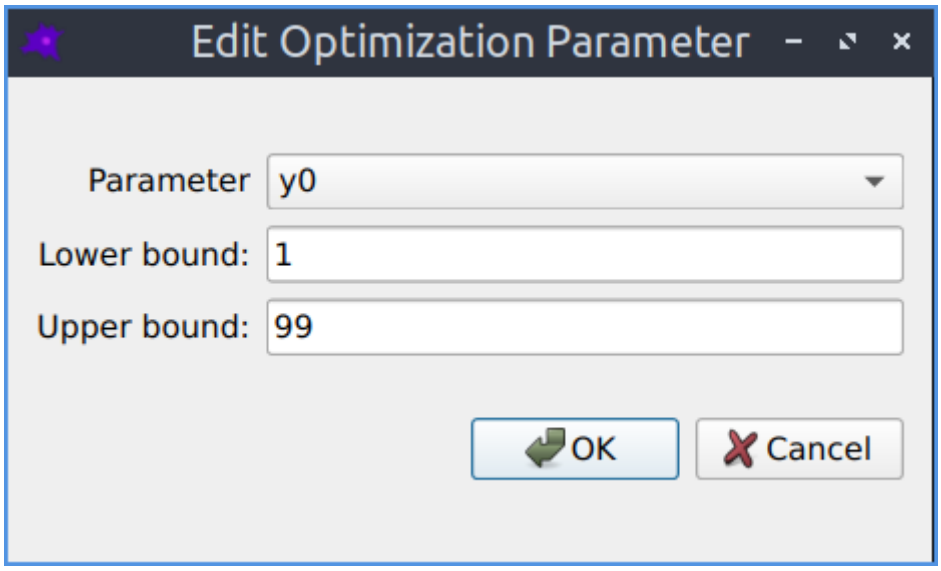


Fig. 3: Adding or modifying a parameter to be optimized.

- **Parameter**
  - This can be a model parameter or a reaction parameter
- **Lower bound**
  - The minimum allowed value this parameter can take
- **Upper bound**
  - The maximum allowed value this parameter can take

15.3 Optimization Target

- **Species**
  - The species to target
- **Target type**
  - Either the concentration, or the concentration rate of change
- **Simulation time**
  - The timepoint in the simulation when this target should apply
- **Target values**
  - The desired spatial distribution of values
  - If not specifying this defaults to zero everywhere in the compartment
  - See the *Optimization Target Image Import* section for more details
- **Difference type**

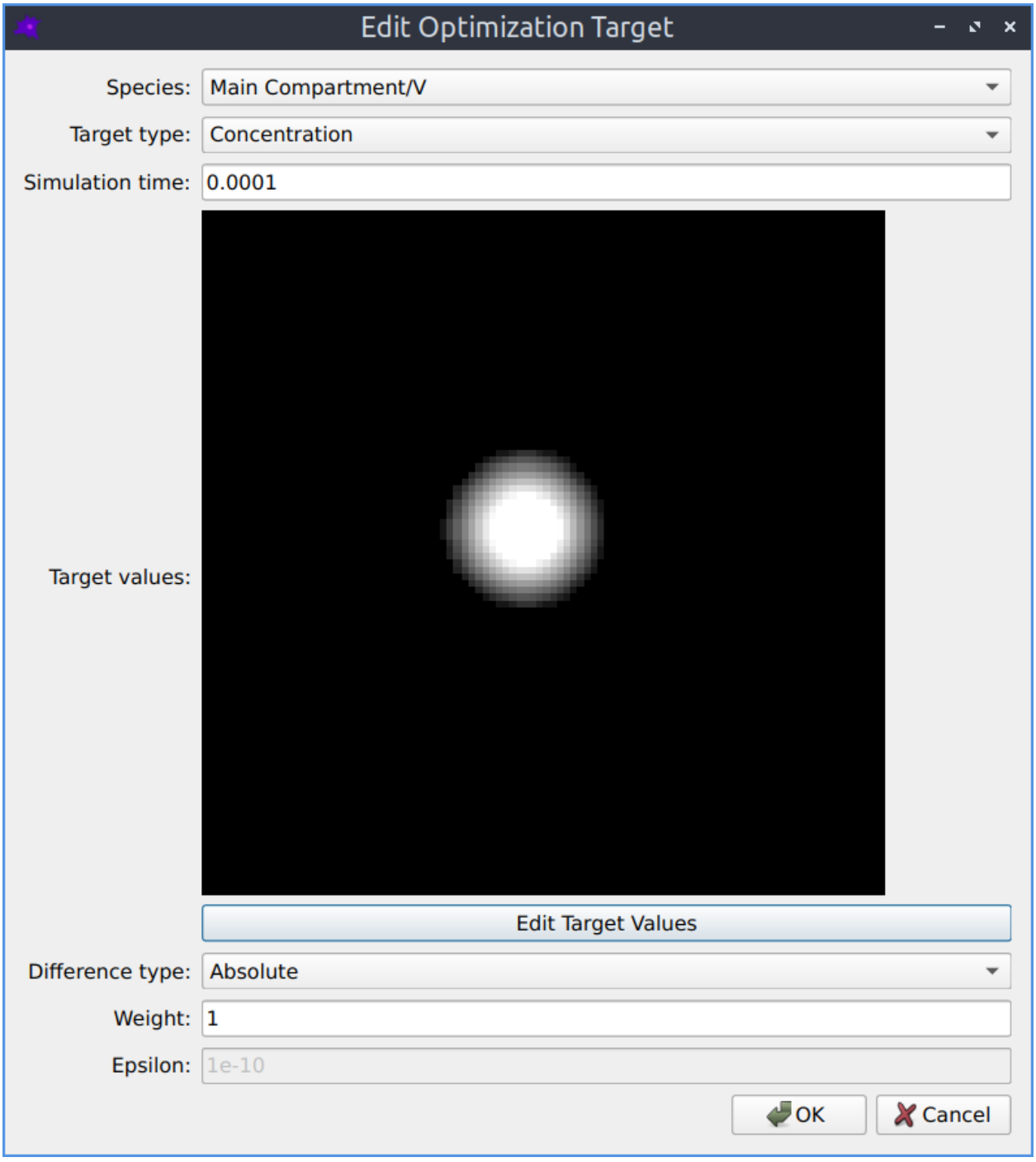


Fig. 4: Adding or modifying a target to optimize for.

  - How to compare the target t with the results r
  - Absolute: $|r - t|$
  - Relative: $|r - t|/|t + \epsilon|$

- **Weight**
  - The relative importance of this target
  - The cost function for this target is multiplied by this weight
  - Only relevant when there are multiple targets
- **Epsilon**
  - The ϵ parameter in the relative difference measure
  - Avoids infinities caused by dividing by zero

15.4 Optimization Target Image Import

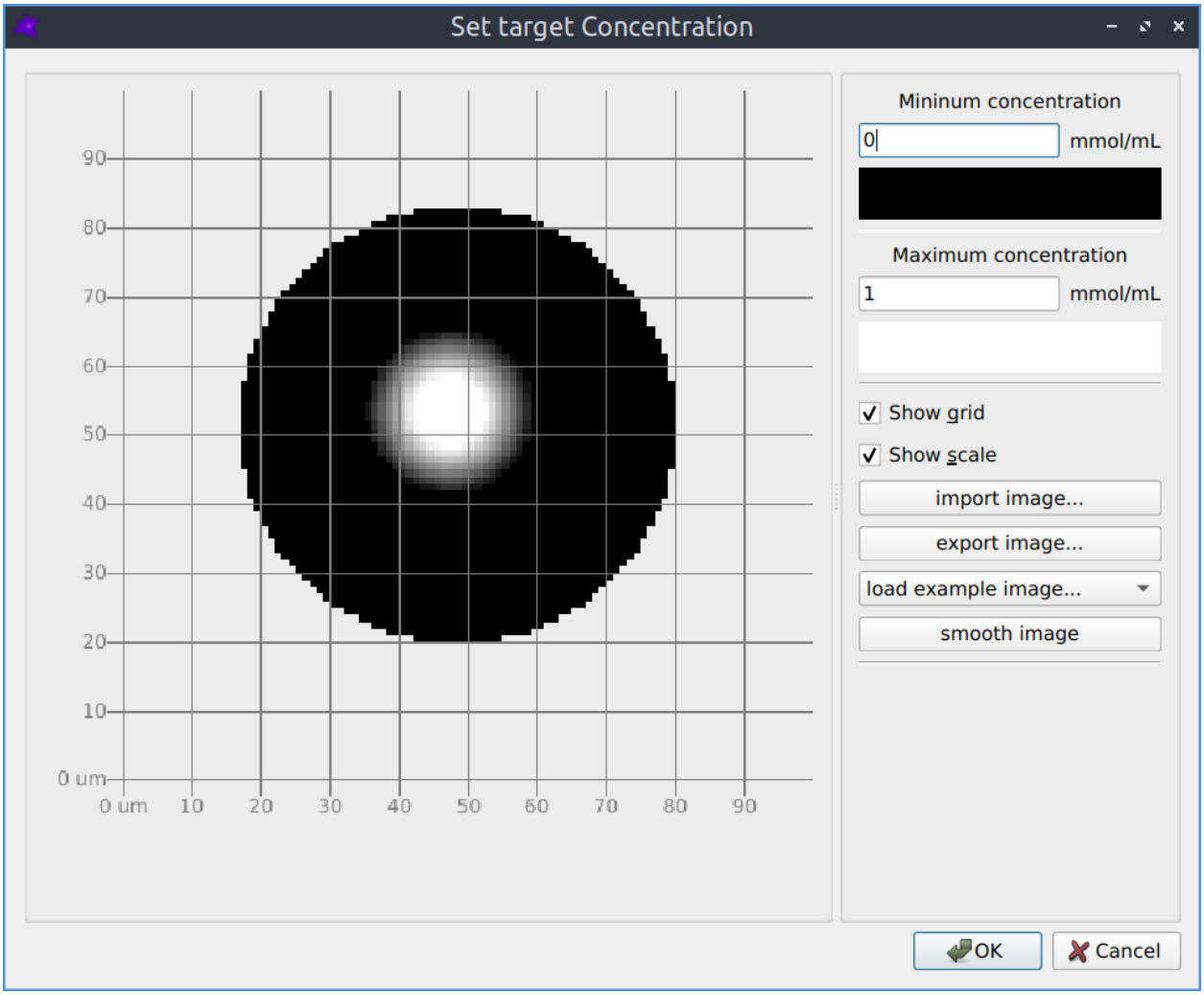


Fig. 5: Importing a target species concentration from an image.

The spatial distribution of the target concentration or rate of change of concentration can be imported from a grayscale image in the same way as initial species concentrations.

- **Minimum concentration**
  - The concentration corresponding to a black pixel in the image
- **Maximum concentration**
  - The concentration corresponding to a white pixel in the image

15.5 More information

- On each thread, a separate optimization is performed
- Within each thread, each item in the population has a value for each parameter being optimized
- It also has a *fitness*, which is the sum of the differences between each target and the corresponding result
- With each optimization iteration, the parameters are evolved to improve (reduce) this *fitness* value
- This means each iteration requires *threads* * *population* simulations of the model
- The algorithms are all *derivative free* optimization methods

CHAPTER

SIXTEEN

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*
- *Mesh refinement*

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

16.1 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.

16.2 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

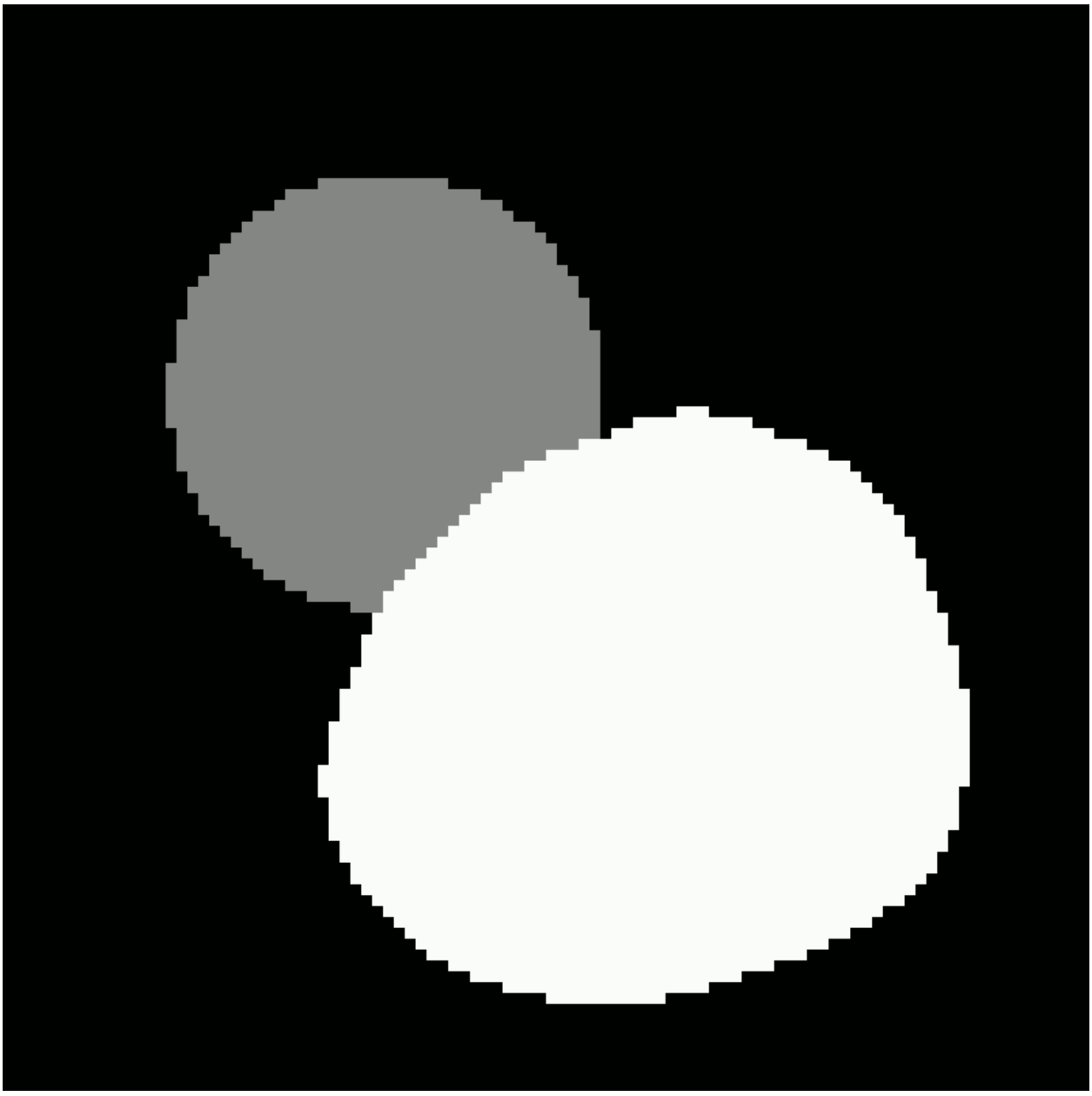


Fig. 1: Initial segmented geometry image.

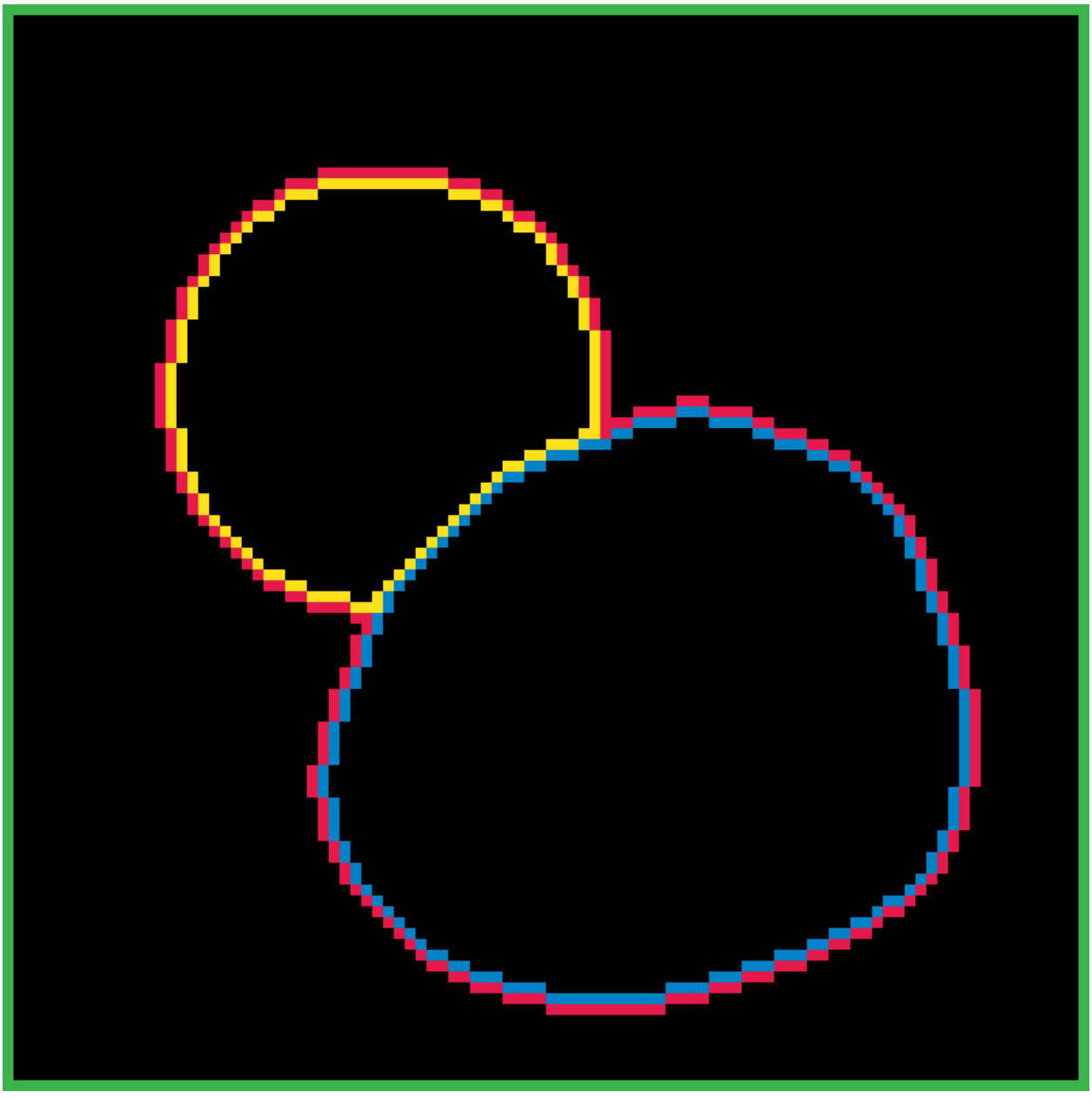


Fig. 2: Pixel Boundary contours.

intersect are identified and used to split the contours into lines which separate pairs of adjacent compartments, and duplicates are removed.

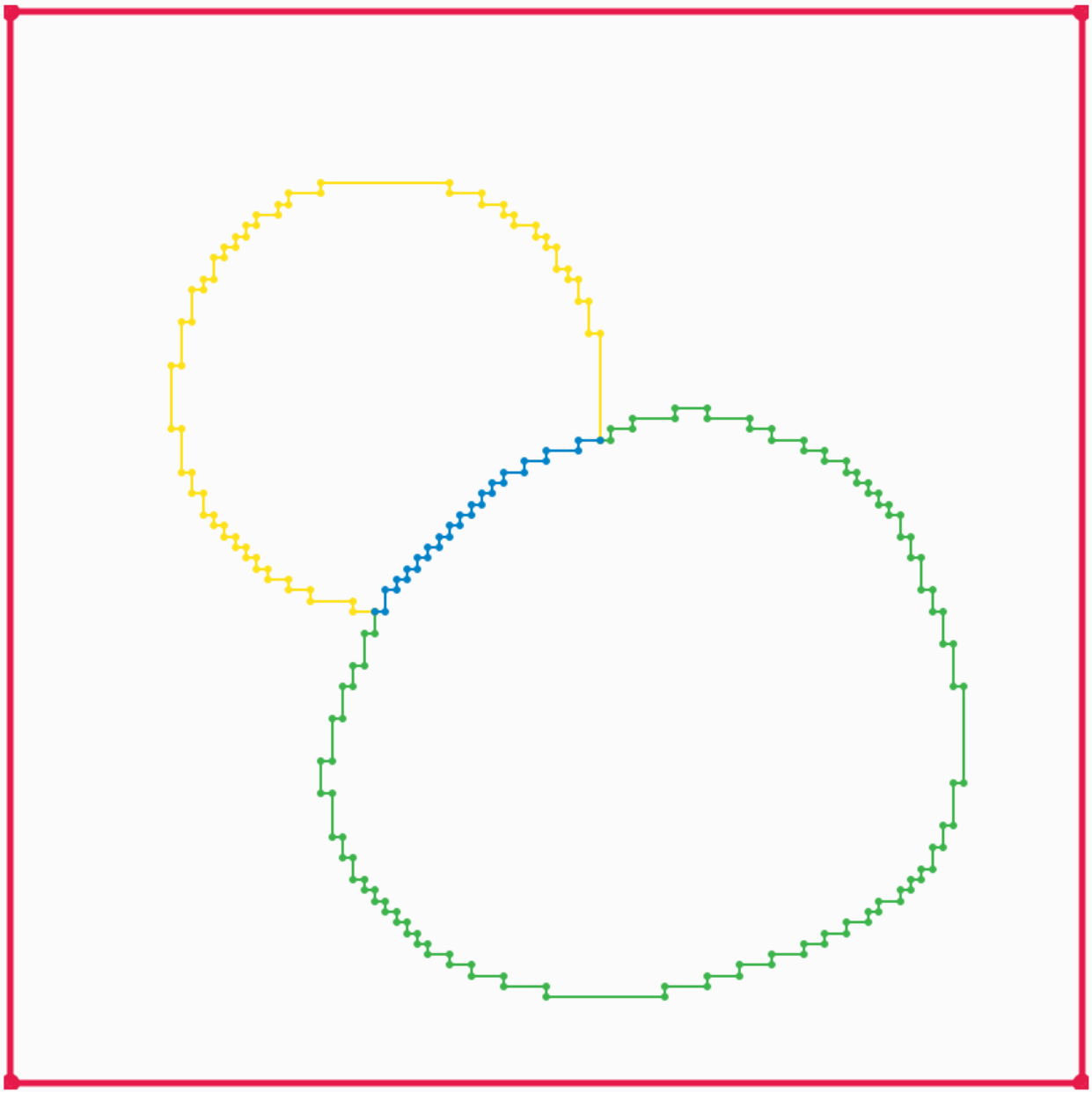


Fig. 3: Split pixel-edge boundary contours.

16.3 Boundary line simplification

Once we have identified all of our boundaries, we want to simplify them by removing points from the boundary. There are two ways of doing this. The default one is Visvalingam-Whyatt polyline simplification, which allows each boundary line to be independently simplified. 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.

An alternative topology-preserving polyline simplification method is also available. This simplifies all the lines simultaneously with the constraint that no new intersections are created, or in other words without any of the simplified boundary lines crossing each other. This method can be useful for very complicated geometry images with many boundary lines, where manually adjusting the number of points for each line would be impractical.

16.4 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 erosion operations
- green rectangle: region of interest where the next erosion operation will be applied
- red dot: current interior point

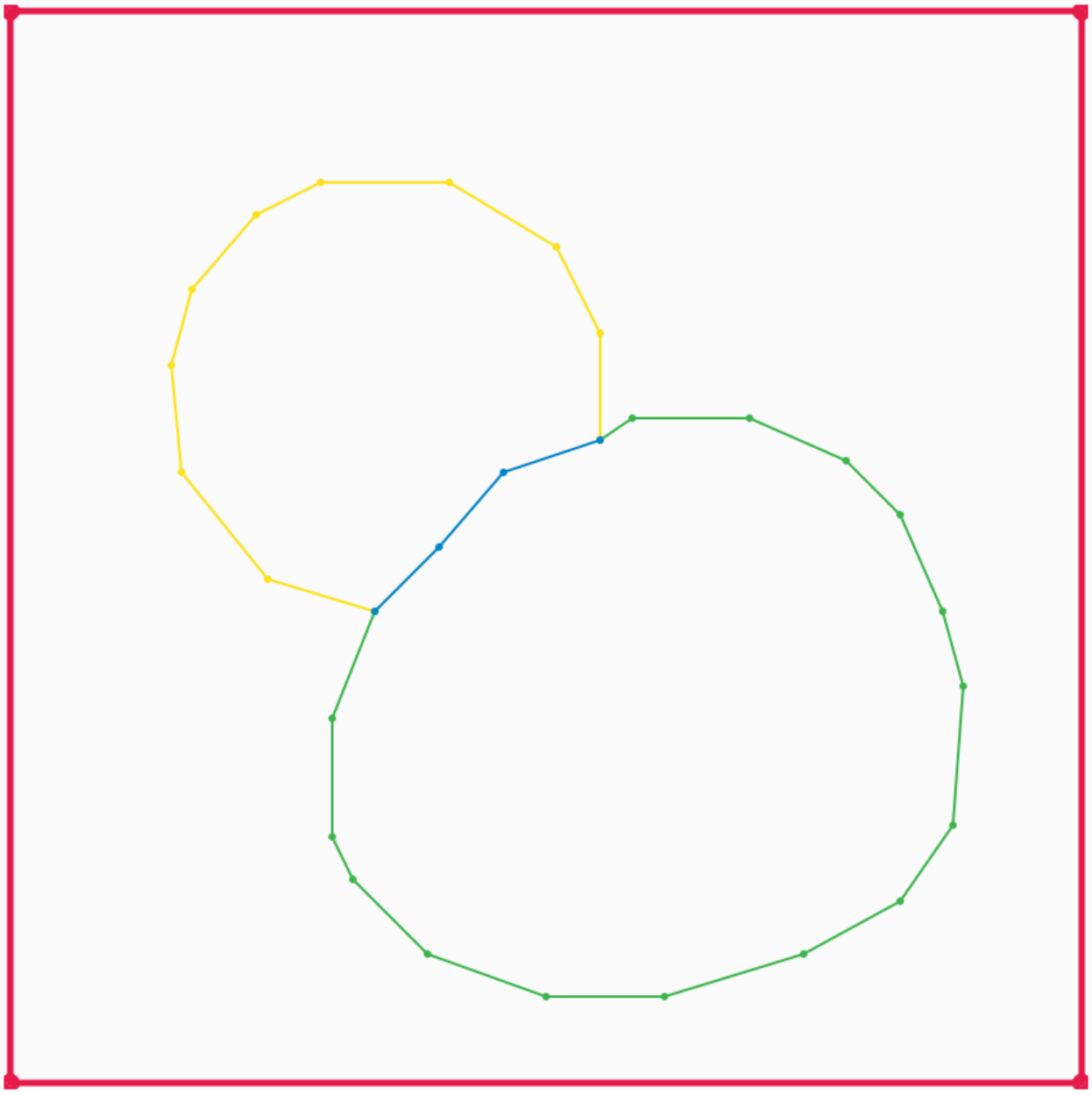


Fig. 4: Simplified boundary lines.

16.5 Triangulation

The set of boundaries is then triangulated using the 2D Conforming Triangulations and Meshes package from the CGAL library. This generates a constrained conforming Delaunay triangulation (CCDT) from the boundary lines and interior points.

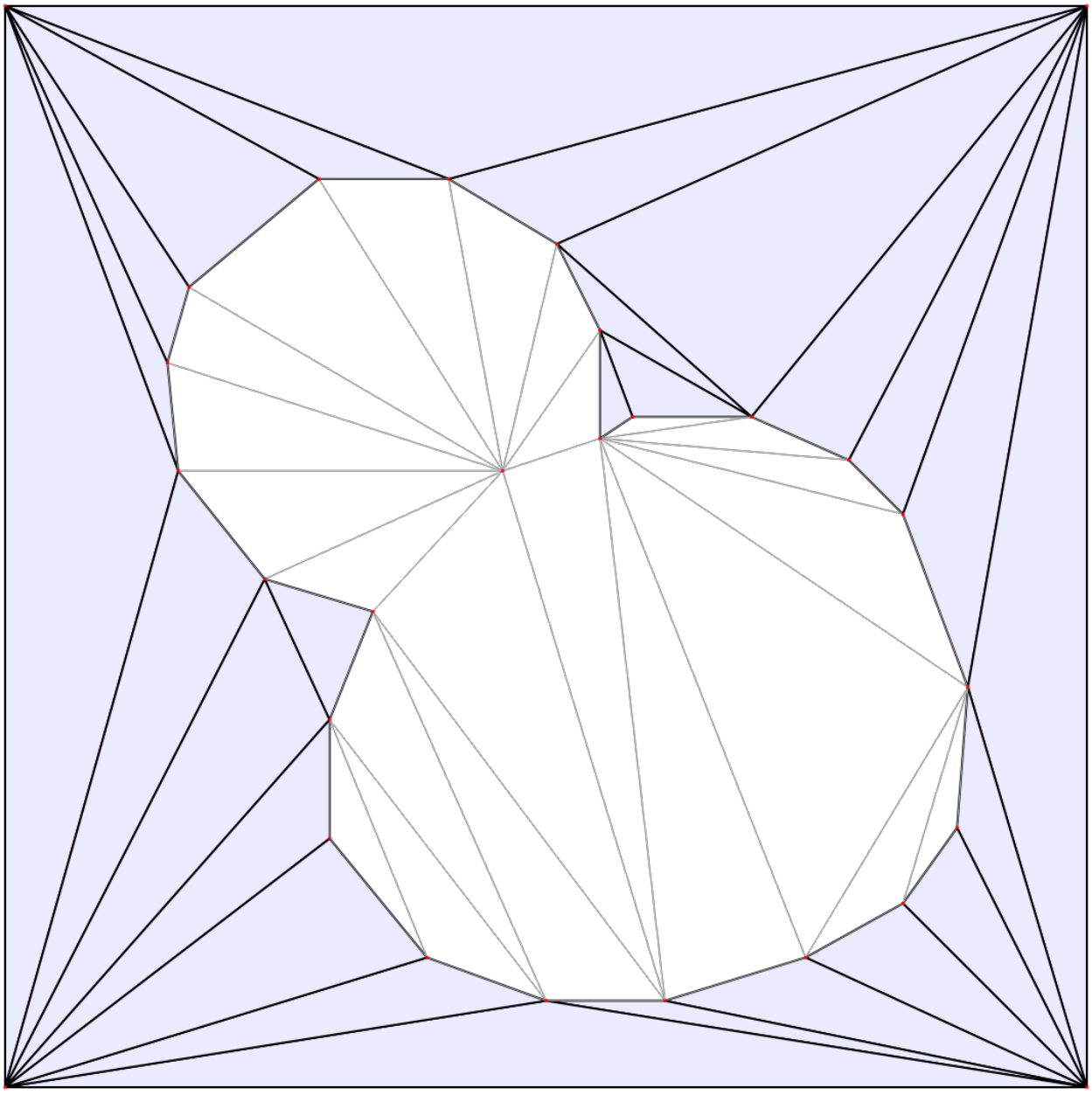
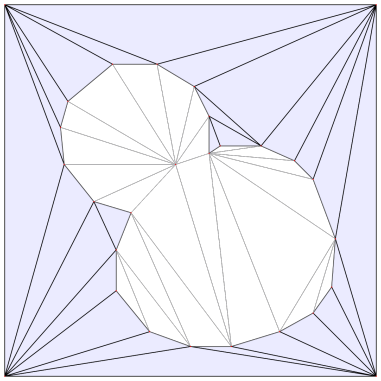
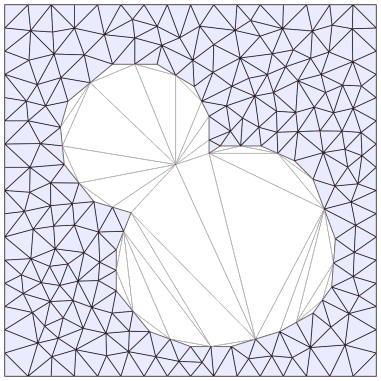


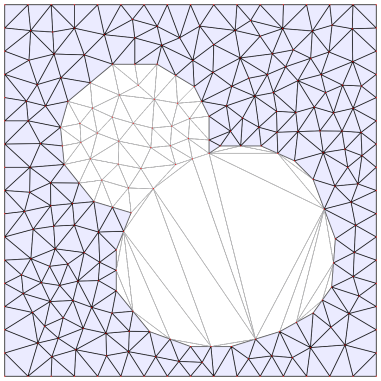
Fig. 5: Initial triangulation.

16.6 Mesh refinement

The mesh is then constructed from the triangulation using Delauney refinement, which inserts points inside the compartment as far away as possible from the existing points, and triangulates them. It continues this until the minimum triangle angle is sufficiently large, and the maximum triangle area is sufficiently small. The minimum required angle is fixed to the largest value for which the algorithm is guaranteed to succeed. The maximum allowed triangle area for each compartment can be specified by the user. If necessary points will also be added to the boundary lines (known as Steiner points).







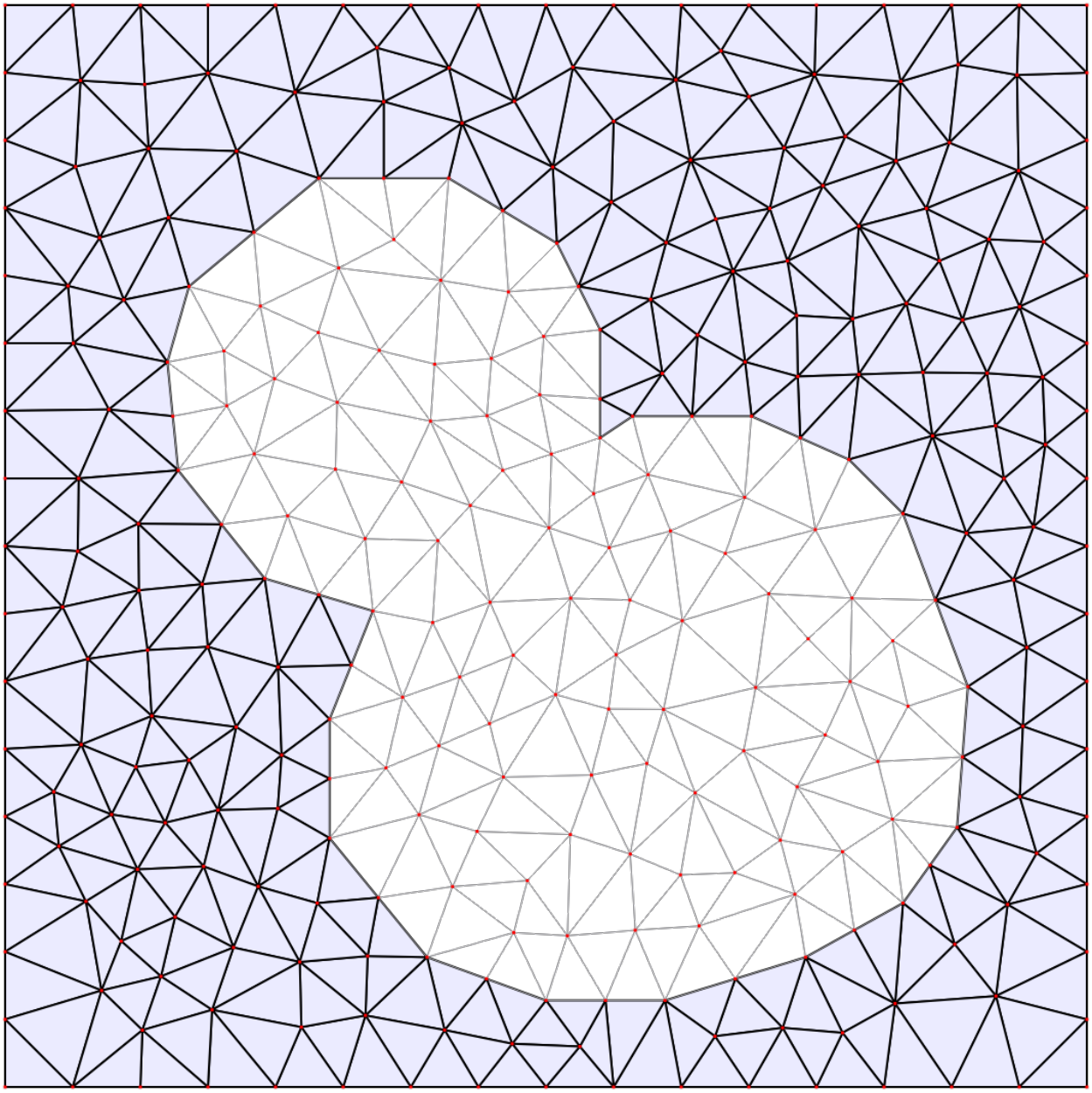


Fig. 6: Final mesh.

CHAPTER

SEVENTEEN

COMMAND LINE INTERFACE

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

-  linux
-  macOS
-  windows

17.1 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
```

17.2 Command line parameters

```
Spatial Model Editor CLI v1.6.0
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 program version information and exit
  -d,--dump-config            Dump the default config ini file and exit
  -c,--config                 Read an ini file containing simulation options
```

17.3 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

EIGHTEEN

MATHS

18.1 Reaction-Diffusion

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

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

where

- c_s is the concentration of species s at position (x, y, z) 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.

18.2 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. They are evaluated at every point inside the compartment

18.3 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.

18.4 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

NINETEEN

UNITS

19.1 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*.

19.2 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`

19.3 More information

For more information see `units.pdf`

CHAPTER

TWENTY

CHANGELOG

20.1 [1.6.0] - 2024-03-26

20.1.1 Added

- support for using the DuneCopasi simulator with 3d models #937
- 3d volume rendering for voxel visualization #926
- visualization of 3d model geometry meshes #882
- import of geometry image files by drag and drop onto the GUI #933

20.2 [1.5.0] - 2023-12-01

20.2.1 Added

- support for dune-copasi 2 #908
- Python 3.12 wheels #897
- Apple Silicon wheels and binaries ssciwr/sme-osx-arm64

20.3 [1.4.0] - 2023-09-04

20.3.1 Added

- initial support for 3d spatial models #850
- support for anisotropic voxels #879

20.3.2 Changed

- disable OK button while selecting geometry image colours #878.
- image arrays in sme (Python interface) now include a z dimension #850

20.3.3 Fixed

- lack of geometry image could cause a crash in some cases #872

20.4 [1.3.6] - 2023-05-23

20.4.1 Added

- support for importing RGBA TIFF image files #862

20.5 [1.3.5] - 2023-03-02

20.5.1 Fixed

- incorrect function evaluations in some cases #855, #856, #857

20.6 [1.3.4] - 2023-02-23

20.6.1 Fixed

- incorrect initial concentrations with DUNE simulator for some models #852

20.7 [1.3.3] - 2022-12-05

20.7.1 Changed

- export format of Image Sampled Field from colors to indices #710 and #830.

20.8 [1.3.2] - 2022-10-13

20.8.1 Added

- support for importing Combine archives #828

20.9 [1.3.1] - 2022-08-10

20.9.1 Changed

- offer to use alternative simulator if simulation setup fails #814

20.9.2 Fixed

- crash caused by analytic initial species concentrations involving dividing by zero #805
- disappearing simulation progress bar dialog #752

20.10 [1.3.0] - 2022-06-10

20.10.1 Added

- parameter fitting functionality to GUI #757
- support for reading SBML files compressed with bzip2 #246

20.10.2 Changed

- GUI no longer writes logging output to console #771

20.10.3 Fixed

- analytic initial species concentrations were not updated when a parameter value was modified #776

20.11 [1.2.2] - 2022-04-28

20.11.1 Added

- support for reading compressed sampledFields #709

20.11.2 Changed

- TBB is now used everywhere for multithreading (previously linux Python wheels used OpenMP) #732

20.11.3 Fixed

- improved multi-threading performance #742
- spurious simulation times are no longer appended upon simulation #751

20.12 [1.2.1] - 2021-09-30

20.12.1 Added

- topology-preserving line simplification option #328
- python interface: `ctrl+c` now cancels a currently running simulation #665

20.12.2 Changed

- model reactions with invalid locations can now be seen and removed or relocated #615

20.12.3 Fixed

- functions with zero arguments are now fully supported #674
- python interface: setting `n_threads=1` for simulate takes precedence over `n_threads` value in model #672
- bug where membrane reactions could disappear when geometry was changed #679
- bug where removing a compartment could result in an invalid model or a crash #685
- SBML export includes any ModifierSpecies in reactions #133

20.12.4 Removed

- incomplete Parametric geometry sbml export support #599

20.13 [1.2.0] - 2021-09-13

20.13.1 Added

- non-spatial model guided import with automatic reaction rate rescaling #607
- python interface: optional multithreading (via OpenMP) for Pixel simulations on linux #662

20.13.2 Fixed

- bug where loading a model with simulation data and changing model units could cause a crash #666
- show warning message before dune simulation of model with non-spatial species #664
- minimum supported version of MacOS is now 10.14 instead of 10.15 #669

20.14 [1.1.5] - 2021-09-01

20.14.1 Added

- python interface: access to all built-in example models #637
- python interface: view and edit uniform/analytic/image species initial concentrations #644

20.14.2 Changed

- compartment colour assignments are now preserved when the geometry image is resized #587

20.14.3 Fixed

- python interface: bug where compartment geometry mask was not updated after geometry image changed #630
- slow loading of models with large geometry images #632
- avoid constructing mesh twice on model load #597
- crash when importing geometry image after importing non-spatial model #651

20.15 [1.1.4] - 2021-08-17

20.15.1 Added

- python interface: `Model.simulation_results()` provides access to existing model simulation data #622

20.15.2 Changed

- python interface: simulation results now provided in numpy.ndarray format #610

20.15.3 Fixed

- bug where opening a corrupted sme file caused a crash #618

20.16 [1.1.3] - 2021-08-10

20.16.1 Added

- python interface: add option to simulate without returning the results to reduce RAM usage #610
- more detailed error message when a model cannot be loaded #449

20.16.2 Fixed

- bug where invalid reaction rate expression caused simulation to crash #609

20.17 [1.1.2] - 2021-07-13

20.17.1 Added

- autocomplete when editing maths #559
- support for all L3 sbml math in reaction rates (with Pixel simulator) #569
- option to invert y-axis of images #568
- option to set size of geometry image in third dimension #582
- `umol` unit of amount added to built-in units #600

20.17.2 Changed

- simulation length/intervals only loaded from model on 'reset' click or new model load #565
- models exported as 3d SBML models (to be consistent with our 3d units for species concentrations and volumes) #588

20.17.3 Fixed

- bug when clicking on simulation results plot selected wrong timepoint #570
- reaction rates involving species from unrelated compartments now displays an error, instead of only giving an error on simulation #552
- simulation crash when species is changed to constant & simulation data are re-loaded #561
- inaccuracy when re-starting a simulation that was stopped early #388
- bug where compartment sizes were not updated when compartment image resolution was altered #583
- bug where invalid mesh could cause a crash #585
- bug where simulating after changing geometry image size could cause a crash #591
- incorrect units used for compartment volumes when referenced in reaction rates #584
- bug where membrane area could be incorrect on model load #595

20.18 [1.1.1] - 2021-05-30

20.18.1 Added

- COPASI cps file import support (if COPASI is installed and on path) #492
- shift + mouse scroll to zoom in and out of images #532
- support for spatial models with reactions whose compartment is not specified #481

20.18.2 Changed

- mouse scroll no longer changes species stoichiometry in reactions #549
- membrane reaction rates automatically rescaled by membrane area on non-spatial model import #557
- reactions moved between compartments and membranes rescaled by ratio of area to volume #558

20.18.3 Fixed

- bug loading simulation settings depending on system locale setting #535
- slow loading of simulation data #504
- bug where simulation doesn't run again after being stopped #545
- bug where selected reaction sometimes changes when reaction location is changed #548
- ensure species involved in a reaction are valid when reaction location is changed #551
- bug where loading existing simulation uses initial concentrations instead of latest simulation concentrations #541

20.19 [1.1.0] - 2021-05-04

20.19.1 Added

- sme file format, contains model, simulation settings and simulation results #482
- support multiple simulation lengths as comma delimited lists #464
- support for simulating models with empty compartments with the Dune-Copasi simulator #435
- option to resize the number of pixels in the geometry image #462
- option to reduce the number of colours in the geometry image #280
- support for non-integer stoichiometries in reactions #495
- optional grid and scale to geometry image #497
- python interface: DUNE simulator can now be used #276
- python interface: existing simulations can be continued #514
- zoom option to geometry image #516

20.19.2 Changed

- use v1.1.0 of Dune-Copasi simulator

20.19.3 Fixed

- python interface: revert change in default simulate() behaviour when doing a second simulation of a model #475
- multiple event related simulation bugs #484, #485, #486
- incorrect compartment name displayed in simulate display options #487
- simulation image intervals changing length of simulation #492
- bug where removing compartment could cause a crash #506
- artefacts in import of species concentrations into simulation #508
- DUNE maximum iteraction count simulation error #490

20.20 [1.0.9] - 2021-04-06

20.20.1 Added

- open/save as `.sme` filetype: contains both the model and the simulation results #461
- support for time-based events: parameters and species concentrations can be set at specified times in the simulation
- diagnostic info & image when a simulation fails #432
- zoom option for Boundary and Mesh images in the Geometry Tab
- new option in Simulation Tab to export concentrations to model as initial concentrations

20.20.2 Changed

- simulation is no longer reset when the Simulate tab is left in the GUI
- simulations can be continued using a different simulator
- use v1.0.0 of Dune-Copasi simulator
- GUI warns before closing a model with unsaved changes #346
- Membrane names can now be edited in the GUI and in the python interface

20.20.3 Fixed

- diffusion constant bug in the Pixel simulator #468
- bug in dune-copasi mesh generation that could cause a crash #434
- bug where unused diffusion constant parameter was sometimes not removed from sbml document
- data race in generating simulation concentration images #434

20.20.4 Removed

- incomplete Parametric geometry sbml import support #452

20.21 [1.0.8] - 2021-02-10

20.21.1 Added

- python library can now import a new geometry image using `sme.Model.import_geometry_from_image()`

20.21.2 Changed

- mesh generation is now done using the CGAL library

20.21.3 Removed

- dependence on Triangle mesh library removed due to its non-free license

20.22 [1.0.7] - 2021-02-07

20.22.1 Fixed

- illegal instruction crash on old CPUs #422

20.23 [1.0.6] - 2021-01-29

20.23.1 Added

- python library `sme.Model.simulate`:
  - simulation results now also contain rate of change of concentrations
  - option to return partial results instead of throwing on simulation timeout

20.23.2 Changed

- more accurate initial concentration interpolation for dune-copasi simulations
  - applies to both GUI simulations and exported TIFF files for stand-alone dune-copasi simulation
- improved performance of interior point determination in mesh construction

20.23.3 Fixed

- meshing bug where invalid 2-point boundary line could cause crash

20.24 [1.0.5] - 2021-01-02

20.24.1 Added

- user can edit dune-copasi Newton solver parameters
- Pixel simulator supports reactions that explicitly depend on t, x, y

20.24.2 Changed

- dune-copasi simulations now support membrane flux terms
  - rectangular membrane compartments no longer required
  - simpler mesh generation
  - faster and more accurate simulations
- improved boundary construction
  - use pixel edges as boundary edges instead of pixels
  - shared boundaries and their end points now unambiguously determined
- improved line simplification

20.24.3 Fixed

- bug in scaling of membrane reactions in Pixel simulations
  - added missing `1/a` factor, where `a` is the width of a pixel in model units

20.24.4 Removed

- inconsistent `RateRule` support in SBML import

20.25 [1.0.4] - 2020-10-22

20.25.1 Added

- simulation display options are now saved

20.25.2 Changed

- python library API
  - lists now have name-based look-up, old name-based look-up functions removed
  - examples of old -> new change required in user code:
    * `model.compartment("name")` -> `model.compartments["name"]`
    * `model.specie("name")` -> `model.species["name"]`

20.25.3 Fixed

- species removal bug

20.26 [1.0.3] - 2020-10-19

20.26.1 Added

- python library example notebooks

20.26.2 Changed

- python library performance improvements

20.26.3 Fixed

- python library segfault / missing data issue on some linux platforms

20.27 [1.0.2] - 2020-10-16

20.27.1 Added

- python library documentation and example notebooks
- support for import of geometry image with alpha channel / transparency
- display compartment size and membrane length

20.27.2 Changed

- python library performance improvements

20.27.3 Fixed

- normalisation of simulation concentration images
- geometry imported from another model not saved bug
- python library segfault / missing data issue on linux
- reaction removal bug

20.28 [1.0.1] - 2020-10-5

20.28.1 Added

- membranes to python library
- compartment pixel mask to python library
- timeout parameter to simulate function in python library

20.29 [1.0.0] - 2020-10-2

First official release.

Changelog format based on Keep a Changelog.

CHAPTER

TWENTYONE

SOURCE CODE

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

The spatial-model-editor organization on GitHub contains more repositories including the source code for the website, spatial-model-editor.github.io, the continuous deployment scripts, and other related components.

CHAPTER

TWENTYTWO

LICENSE

The source code for Spatial Model Editor is released under the MIT license, which is a permissive GPL-compatible license.

The open source libraries that it uses are listed here, and are either also released under a permissive GPL-compatible license, or under a GPL license. As described in the gpl-faq, this means that the work as a whole is then licensed under the GPL.

CHAPTER

TWENTYTHREE

CORE

The core library implements all the functionality, which is then exposed to the user via the GUI, Python library, or CLI.

It is split into four (somewhat interdependent) parts: *sme::model*, *sme::mesh*, *sme::simulate* and *sme::common*.

The public headers for each part are in `/include/sme`, and the private implementation details are in `/src`.

For each component X there is

- `X.hpp` the public interface
- `X.cpp` the private implementation
- `X_t.cpp` the tests
- `X_bench.cpp` the benchmarks (optional)

23.1 sme::model

Importing, exporting and editing spatial models.

23.1.1 inc

Model

class **Model**

Public Functions

bool **getIsValid**() const

const QString &**getErrorMessage**() const

bool **getHasUnsavedChanges**() const

const QString &**getCurrentFilename**() const

void **setName**(const QString &name)

QString **getName**() const

ModelCompartments &**getCompartments**()

const *ModelCompartments* &**getCompartments**() const

ModelGeometry &**getGeometry**()

const *ModelGeometry* &**getGeometry**() const

ModelMembranes &**getMembranes**()

const *ModelMembranes* &**getMembranes**() const

ModelSpecies &**getSpecies**()

const *ModelSpecies* &**getSpecies**() const

ModelReactions &**getReactions**()

const *ModelReactions* &**getReactions**() const

ModelFunctions &**getFunctions**()

const *ModelFunctions* &**getFunctions**() const

ModelParameters &**getParameters**()

const *ModelParameters* &**getParameters**() const

ModelEvents &**getEvents**()

const *ModelEvents* &**getEvents**() const

ModelUnits &**getUnits**()

const *ModelUnits* &**getUnits**() const

ModelMath &**getMath**()

const *ModelMath* &**getMath**() const

simulate::*SimulationData* &**getSimulationData**()

const simulate::*SimulationData* &**getSimulationData**() const

SimulationSettings &**getSimulationSettings**()

const *SimulationSettings* &**getSimulationSettings**() const

MeshParameters &**getMeshParameters**()

const *MeshParameters* &**getMeshParameters**() const

simulate::*OptimizeOptions* &**getOptimizeOptions**()

const simulate::*OptimizeOptions* &**getOptimizeOptions**() const

const std::vector<QRgb> &**getSampledFieldColours**() const

explicit **Model**()

Model(*Model*&&) noexcept = default

Model &**operator=**(*Model*&&) noexcept = default

Model &**operator=**(const *Model*&) = delete

Model(const *Model*&) = delete

~Model()

void **createSBMLFile**(const std::string &name)

void **importSBMLFile**(const std::string &filename)

void **importSBMLString**(const std::string &xml, const std::string &filename = {})

void **exportSBMLFile**(const std::string &filename)

void **importFile**(const std::string &filename)

void **exportSMEFile**(const std::string &filename)

QString **getXml**()

void **clear**()

SpeciesGeometry **getSpeciesGeometry**(const QString &speciesID) const

std::string **inlineExpr**(const std::string &mathExpression) const

DisplayOptions **getDisplayOptions**() const

void **setDisplayOptions**(const *DisplayOptions* &displayOptions)

ModelCompartments

class **ModelCompartments**

Public Functions

ModelCompartments()

ModelCompartments(libsbml::Model *model, *ModelMembranes* *membranes, const *ModelUnits* *units, simulate::*SimulationData* *data)

void **setGeometryPtr**(*ModelGeometry* *geometry)

void **setSpeciesPtr**(*ModelSpecies* *species)

void **setReactionsPtr**(*ModelReactions* *reactions)

void **setSimulationDataPtr**(simulate::*SimulationData* *data)

const QStringList &**getIds**() const

const QStringList &**getNames**() const

const QVector<QRgb> &**getColours**() const

QString **add**(const QString &name)

bool **remove**(const QString &id)

QString **getName**(const QString &id) const

QString **setName**(const QString &id, const QString &name)

std::optional<std::vector<QPointF>> **getInteriorPoints**(const QString &id) const

void **setInteriorPoints**(const QString &id, const std::vector<QPointF> &points)

void **setColour**(const QString &id, QRgb colour)

QRgb **getColour**(const QString &id) const

QString **getIdFromColour**(QRgb colour) const

const std::vector<std::unique_ptr<geometry::Compartment>> &**getCompartments**() const

geometry::Compartment ***getCompartment**(const QString &id)

const geometry::Compartment ***getCompartment**(const QString &id) const

double **getSize**(const QString &id) const

const std::map<std::string, double, std::less<>> &**getInitialCompartmentSizes**() const

void **clear**()

bool **getHasUnsavedChanges**() const

void **setHasUnsavedChanges**(bool unsavedChanges)

ModelEvents

class **ModelEvents**

Public Functions

ModelEvents()

explicit **ModelEvents**(libsbml::Model *model, *ModelParameters* *parameters = nullptr, *ModelSpecies* *species = nullptr)

const QStringList &**getIds**() const

const QStringList &**getNames**() const

QString **setName**(const QString &id, const QString &name)

QString **getName**(const QString &id) const

void **setVariable**(const QString &id, const QString &variable)

QString **getVariable**(const QString &id) const

void **setTime**(const QString &id, double time)

double **getTime**(const QString &id) const

void **setExpression**(const QString &id, const QString &expr)

QString **getExpression**(const QString &id) const

QString **add**(const QString &name, const QString &variable)

bool **isParameter**(const QString &id) const

double **getValue**(const QString &id) const

void **remove**(const QString &id)

void **removeAnyUsingVariable**(const QString &variable)

void **applyEvent**(const QString &id)

bool **getHasUnsavedChanges**() const

void **setHasUnsavedChanges**(bool unsavedChanges)

ModelFunctions

class **ModelFunctions**

Public Functions

ModelFunctions()

explicit **ModelFunctions**(libsbml::Model *model)

const QStringList &**getIds**() const

const QStringList &**getNames**() const

QString **setName**(const QString &id, const QString &name)

QString **getName**(const QString &id) const

void **setExpression**(const QString &id, const QString &expression)

QString **getExpression**(const QString &id) const

QStringList **getArguments**(const QString &id) const

QString **addArgument**(const QString &functionId, const QString &argumentId)

void **removeArgument**(const QString &functionId, const QString &argumentId)

QString **add**(const QString &name)

void **remove**(const QString &id)

std::vector<*common*::SymbolicFunction> **getSymbolicFunctions**() const

bool **getHasUnsavedChanges**() const

void **setHasUnsavedChanges**(bool unsavedChanges)

ModelGeometry

class **ModelGeometry**

Public Functions

ModelGeometry()

explicit **ModelGeometry**(libsbml::Model *model, *ModelCompartments* *compartments, *ModelMembranes* *membranes, const *ModelUnits* *units, *Settings* *annotation)

void **importSampledFieldGeometry**(const libsbml::Model *model)

void **importSampledFieldGeometry**(const QString &filename)

void **importGeometryFromImages**(const *common*::ImageStack &imgs, bool keepColourAssignments)

void **updateMesh**()

void **clear**()

int **getNumDimensions**() const

const *common*::VolumeF &**getVoxelSize**() const

void **setVoxelSize**(const *common*::VolumeF &newVoxelSize, bool updateSBML = true)

const *common*::VoxelF &**getPhysicalOrigin**() const

const *common*::VolumeF &**getPhysicalSize**() const

common::VoxelF **getPhysicalPoint**(const *common*::Voxel &voxel) const

QString **getPhysicalPointAsString**(const *common*::Voxel &voxel) const

const *common*::ImageStack &**getImages**() const

mesh::*Mesh2d* ***getMesh2d**() const

mesh::*Mesh3d* ***getMesh3d**() const

bool **getIsValid**() const

bool **getIsMeshValid**() const

bool **getHasImage**() const

void **writeGeometryToSBML**() const

bool **getHasUnsavedChanges**() const

void **setHasUnsavedChanges**(bool unsavedChanges)

ModelMath

class **ModelMath**

Public Functions

ModelMath()

explicit **ModelMath**(const libsbml::Model *model)

void **parse**(const std::string &expr)

double **eval**(const std::map<const std::string, std::pair<double, bool>> &vars = {}) const

bool **isValid**() const

const std::string &**getErrorMessage**() const

ModelMembranes

class **ModelMembranes**

Public Functions

const QStringList &**getIds**() const

const QStringList &**getNames**() const

QString **setName**(const QString &id, const QString &name)

QString **getName**(const QString &id) const

const std::vector<geometry::Membrane> &**getMembranes**() const

const geometry::Membrane *__getMembrane__(const QString &id) const

const std::vector<std::pair<std::string, std::pair<QRgb, QRgb>>> &**getIdColourPairs**() const

double **getSize**(const QString &id) const

void **updateCompartmentNames**(const QStringList &compartmentNames)

void **updateCompartments**(const std::vector<std::unique_ptr<geometry::Compartment>> &compartments)

void **updateCompartmentImages**(const *common*::ImageStack &imgs)

void **importMembraneIdsAndNames**()

void **exportToSBML**(const *common*::VolumeF &voxelSize)

explicit **ModelMembranes**(libsbml::Model *model = nullptr)

~ModelMembranes()

bool **getHasUnsavedChanges**() const

void **setHasUnsavedChanges**(bool unsavedChanges)

ModelParameters

class **ModelParameters**

Public Functions

ModelParameters()

explicit **ModelParameters**(libsbml::Model *model)

void **setEventsPtr**(*ModelEvents* *events)

void **setSpeciesPtr**(*ModelSpecies* *species)

const QStringList &**getIds**() const

const QStringList &**getNames**() const

QString **setName**(const QString &id, const QString &name)

QString **getName**(const QString &id) const

void **setExpression**(const QString &id, const QString &expr)

QString **getExpression**(const QString &id) const

QString **add**(const QString &name)

void **remove**(const QString &id)

const SpatialCoordinates &**getSpatialCoordinates**() const

void **setSpatialCoordinates**(SpatialCoordinates coords)

std::vector<IdName> **getSymbols**(const QStringList &compartments = {}) const

std::vector<IdNameValue> **getGlobalConstants**() const

std::vector<IdNameExpr> **getNonConstantParameters**() const

bool **getHasUnsavedChanges**() const

void **setHasUnsavedChanges**(bool unsavedChanges)

ModelReactions

class **ModelReactions**

Public Functions

ModelReactions()

explicit **ModelReactions**(libsbml::Model *model, const *ModelCompartments* *compartments, const *ModelMembranes* *membranes, bool isNonSpatialModel)

void **makeReactionLocationsValid**()

void **applySpatialReactionRescalings**(const std::vector<ReactionRescaling> &reactionRescalings)

std::vector<ReactionRescaling> **getSpatialReactionRescalings**() const

bool **getIsIncompleteODEImport**() const

QStringList **getIds**(const QString &locationId) const

QStringList **getIds**(const ReactionLocation &reactionLocation) const

std::vector<ReactionLocation> **getReactionLocations**() const

QString **add**(const QString &name, const QString &locationId, const QString &rateExpression = "1")

void **remove**(const QString &id)

void **removeAllInvolvingSpecies**(const QString &speciesId)

QString **setName**(const QString &id, const QString &name)

QString **getName**(const QString &id) const

QString **getScheme**(const QString &id) const

void **setLocation**(const QString &id, const QString &locationId)

QString **getLocation**(const QString &id) const

double **getSpeciesStoichiometry**(const QString &id, const QString &speciesId) const

void **setSpeciesStoichiometry**(const QString &id, const QString &speciesId, double stoichiometry)

QString **getRateExpression**(const QString &id) const

void **setRateExpression**(const QString &id, const QString &expression)

QStringList **getParameterIds**(const QString &id) const

QString **setParameterName**(const QString &reactionId, const QString &parameterId, const QString &name)

QString **getParameterName**(const QString &reactionId, const QString &parameterId) const

void **setParameterValue**(const QString &reactionId, const QString &parameterId, double value)

double **getParameterValue**(const QString &reactionId, const QString &parameterId) const

QString **addParameter**(const QString &reactionId, const QString &name, double value)

void **removeParameter**(const QString &reactionId, const QString &id)

bool **dependOnVariable**(const QString &variableId) const

bool **getHasUnsavedChanges**() const

void **setHasUnsavedChanges**(bool unsavedChanges)

ModelSpecies

class **ModelSpecies**

Public Functions

ModelSpecies()

ModelSpecies(libsbml::Model *model, const *ModelCompartments* *compartments, const *ModelGeometry* *geometry, const *ModelParameters* *parameters, const *ModelFunctions* *functions, simulate::*SimulationData* *data, *Settings* *annotation)

void **setReactionsPtr**(*ModelReactions* *reactions)

void **setSimulationDataPtr**(simulate::*SimulationData* *data)

bool **containsNonSpatialReactiveSpecies**() const

QString **add**(const QString &name, const QString &compartmentId)

void **remove**(const QString &id)

QString **setName**(const QString &id, const QString &name)

QString **getName**(const QString &id) const

void **updateCompartmentGeometry**(const QString &compartmentId)

void **setCompartment**(const QString &id, const QString &compartmentId)

QString **getCompartment**(const QString &id) const

QStringList **getIds**(const QString &compartmentId) const

QStringList **getNames**(const QString &compartmentId) const

void **setIsSpatial**(const QString &id, bool isSpatial)

bool **getIsSpatial**(const QString &id) const

void **setDiffusionConstant**(const QString &id, double diffusionConstant)

double **getDiffusionConstant**(const QString &id) const

ConcentrationType **getInitialConcentrationType**(const QString &id) const

void **setInitialConcentration**(const QString &id, double concentration)

double **getInitialConcentration**(const QString &id) const

void **setAnalyticConcentration**(const QString &id, const QString &analyticExpression)

void **setFieldConcAnalytic**(geometry::Field &field, const std::string &expr, const std::map<std::string, double, std::less<>> &substitutions = {})

QString **getAnalyticConcentration**(const QString &id) const

void **updateAllAnalyticConcentrations**()

void **setSampledFieldConcentration**(const QString &id, const std::vector<double> &concentrationArray)

std::vector<double> **getSampledFieldConcentration**(const QString &id, bool maskAndInvertY = false) const

common::ImageStack **getConcentrationImages**(const QString &id) const

void **setColour**(const QString &id, QRgb colour)

QRgb **getColour**(const QString &id) const

void **setIsConstant**(const QString &id, bool constant)

bool **getIsConstant**(const QString &id) const

bool **isReactive**(const QString &id) const

void **removeInitialAssignments**()

QString **getSampledFieldInitialAssignment**(const QString &id) const

geometry::Field ***getField**(const QString &id)

const geometry::Field ***getField**(const QString &id) const

bool **getHasUnsavedChanges**() const

void **setHasUnsavedChanges**(bool unsavedChanges)

ModelUnits

class **ModelUnits**

Public Functions

explicit **ModelUnits**(libsbml::Model *model = nullptr)

const Unit &**getTime**() const

int **getTimeIndex**() const

const QVector<Unit> &**getTimeUnits**() const

QVector<Unit> &**getTimeUnits**()

void **setTimeIndex**(int index)

const Unit &**getLength**() const

int **getLengthIndex**() const

const QVector<Unit> &**getLengthUnits**() const

QVector<Unit> &**getLengthUnits**()

void **setLengthIndex**(int index)

const Unit &**getVolume**() const

int **getVolumeIndex**() const

const QVector<Unit> &**getVolumeUnits**() const

QVector<Unit> &**getVolumeUnits**()

void **setVolumeIndex**(int index)

const Unit &**getAmount**() const

int **getAmountIndex**() const

const QVector<Unit> &**getAmountUnits**() const

QVector<Unit> &**getAmountUnits**()

void **setAmountIndex**(int index)

const QString &**getConcentration**() const

const QString &**getDiffusion**() const

const QString &**getCompartmentReaction**() const

const QString &**getMembraneReaction**() const

bool **getHasUnsavedChanges**() const

void **setHasUnsavedChanges**(bool unsavedChanges)

Settings

struct **Settings**

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

SimulationSettings **simulationSettings** = {}

DisplayOptions **displayOptions** = {}

MeshParameters **meshParameters** = {}

std::map<std::string, QRgb> **speciesColours** = {}

sme::simulate::*OptimizeOptions* **optimizeOptions** = {}

std::vector<QRgb> **sampledFieldColours** = {}

struct **SimulationSettings**

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

std::vector<std::pair<std::size_t, double>> **times** = {}

simulate::*Options* **options** = {}

sme::simulate::*SimulatorType* **simulatorType** = {}

struct **DisplayOptions**

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

std::vector<bool> **showSpecies** = {}

bool **showMinMax** = {true}

bool **normaliseOverAllTimepoints** = {true}

bool **normaliseOverAllSpecies** = {true}

bool **showGeometryGrid** = {false}

bool **showGeometryScale** = {false}

bool **invertYAxis** = {false}

struct **MeshParameters**

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

std::vector<std::size_t> **maxPoints** = {}

std::vector<std::size_t> **maxAreas** = {}

std::size_t **boundarySimplifierType** = {0}

23.1.2 src

23.2 sme::mesh

Constructing the simplified boundary lines and triangular mesh approximation to the geometry.

23.2.1 inc

Mesh

class **Mesh2d**

Constructs a triangular mesh from a geometry image.

Given a segmented geometry image, and the colours that correspond to compartments in the image, the compartment boundaries are identified and simplified to a set of connected straight lines, and the resulting PLSG is triangulated to give a triangular mesh.

The mesh is provided as an image, in GMSH format, and as flat arrays of indices and vertices for SBML.

The number of points used for each boundary line and the maximum triangle area allowed for each compartment can then be adjusted.

Public Functions

Mesh2d()

explicit **Mesh2d**(const QImage &image, std::vector<std::size_t> maxPoints = {}, std::vector<std::size_t> maxTriangleArea = {}, const *common*::VolumeF &voxelSize = {1.0, 1.0, 1.0}, const *common*::VoxelF &originPoint = {0.0, 0.0, 0.0}, const std::vector<QRgb> &compartmentColours = {}, std::size_t boundarySimplificationType = 0)

Constructs a mesh from the supplied image.

Parameters

- **image** – **[in]** the segmented geometry image
- **maxPoints** – **[in]** the max points allowed for each boundary line
- **maxTriangleArea** – **[in]** the max triangle area allowed for each compartment
- **voxelSize** – **[in]** the physical size of a pixel
- **originPoint** – **[in]** the physical location of the (0,0) pixel
- **compartmentColours** – **[in]** the colours of compartments in the image

~Mesh2d()

bool **isValid**() const

Returns true if the mesh is valid.

const std::string &**getErrorMessage**() const

Returns an error message if the mesh is invalid.

std::size_t **getNumBoundaries**() const

The number of boundary lines in the mesh.

std::size_t **getBoundarySimplificationType**() const

Get the type of boundary simplification.

void **setBoundarySimplificationType**(std::size_t boundarySimplificationType)

Set the type of boundary simplification.

void **setBoundaryMaxPoints**(std::size_t boundaryIndex, std::size_t maxPoints)

Set the maximum number of allowed points.

If the boundary simplification type is one where each boundary line can be independently simplified, maxPoints applies only to the specified boundary line, and only this boundary line is potentially altered by this method.

If the boundary simplification type is a topology-preserving type that simplifies all boundaries simultaneously, maxPoints is the total number of allowed points summed over all boundaries, and all boundary lines can potentially be altered by this method.

Parameters

- **boundaryIndex** – **[in]** the index of the boundary
- **maxPoints** – **[in]** the maximum number of points allowed

std::size_t **getBoundaryMaxPoints**(std::size_t boundaryIndex) const

Get the maximum number of allowed points for a given boundary.

Parameters

boundaryIndex – **[in]** the index of the boundary

std::vector<std::size_t> **getBoundaryMaxPoints**() const

The maximum allowed points for each boundary in the mesh.

void **setCompartmentMaxTriangleArea**(std::size_t compartmentIndex, std::size_t maxTriangleArea)

Set the maximum allowed triangle area for a given compartment.

Parameters

- **compartmentIndex** – **[in]** the index of the compartment
- **maxTriangleArea** – **[in]** the maximum allowed triangle area

std::size_t **getCompartmentMaxTriangleArea**(std::size_t compartmentIndex) const

Get the maximum allowed triangle area for a given compartment.

Parameters

compartmentIndex – **[in]** the index of the compartment

const std::vector<std::size_t> &**getCompartmentMaxTriangleArea**() const

The maximum allowed triangle areas for each compartment in the mesh.

const std::vector<std::vector<QPointF>> &**getCompartmentInteriorPoints**() const

The interior points for each compartment in the mesh.

Each interior point is chosen to be as far as possible from the edge of the region, to reduce the chance that it ends up outside the meshed approximation to the region as the boundary lines are simplified.

Note: There may be multiple interior points for a single compartment, for example if that compartment has two disconnected regions

---

void **setPhysicalGeometry**(const *common*::VolumeF &voxelSize, const *common*::VoxelF &originPoint = {0.0, 0.0, 0.0})

The physical volume and origin to use.

The boundary lines and mesh use pixel units internally, and are rescaled to physical values using the supplied physical origin and pixel width.

Parameters

- **voxelSize** – **[in]** the physical size of a voxel
- **originPoint** – **[in]** the physical location of the `(0,0,0)` voxel

std::vector<double> **getVerticesAsFlatArray**() const

The physical mesh vertices as a flat array of doubles.

For saving to the SBML document.

std::vector<int> **getTriangleIndicesAsFlatArray**(std::size_t compartmentIndex) const

The mesh triangle indices as a flat array of ints.

For saving to the SBML document.

const std::vector<std::vector<TriangulateTriangleIndex>> &**getTriangleIndices**() const

The mesh triangle indices.

The indices of the triangles used in the mesh for each compartment

std::pair<*common*::ImageStack, *common*::ImageStack> **getBoundariesImages**(const QSize &size, std::size_t boldBoundaryIndex) const

An image of the compartment boundary lines.

A pair of images are returned, the first is the image of the boundary lines, and the second is a map from each pixel to the corresponding boundary index, which can be used to identify which boundary was clicked on by the user.

Parameters

- **size** – **[in]** the desired size of the image
- **boldBoundaryIndex** – **[in]** the boundary line to emphasize in the image

std::pair<*common*::ImageStack, *common*::ImageStack> **getMeshImages**(const QSize &size, std::size_t compartmentIndex) const

An image of the mesh.

A pair of images are returned, the first is the image of the mesh, and the second is a map from each pixel to the corresponding compartment index, which can be used to identify which compartment was clicked on by the user.

Parameters

- **size** – **[in]** the desired size of the image
- **compartmentIndex** – **[in]** the compartment to emphasize in the image

Returns

a pair of images: mesh, compartment index

QString **getGMSH**() const

The mesh in GMSH format.

Returns
the mesh in GMSH format

Public Static Attributes

static constexpr std::size_t **dim** = 2

class **Mesh3d**

Public Functions

Mesh3d()

explicit **Mesh3d**(const *sme*::*common*::ImageStack &imageStack, std::vector<std::size_t> maxCellVolume = {}, const *common*::VolumeF &voxelSize = {1.0, 1.0, 1.0}, const *common*::VoxelF &originPoint = {0.0, 0.0, 0.0}, const std::vector<QRgb> &compartmentColours = {})

Constructs a 3d mesh from the supplied ImageStack.

Parameters

- **imageStack** – **[in]** the segmented geometry ImageStack
- **maxCellVolume** – **[in]** the max volume (in voxels) of a cell for each compartment
- **voxelSize** – **[in]** the physical size of a voxel
- **originPoint** – **[in]** the physical location of the **(0,0,0)** voxel
- **compartmentColours** – **[in]** the colours of compartments in the image

~Mesh3d()

bool **isValid**() const

Returns true if the mesh is valid.

const std::string &**getErrorMessage**() const

Returns an error message if the mesh is invalid.

void **setCompartmentMaxCellVolume**(std::size_t compartmentIndex, std::size_t maxCellVolume)

Set the maximum allowed cell volume in voxels for a given compartment.

Parameters

- **compartmentIndex** – **[in]** the index of the compartment
- **maxTriangleArea** – **[in]** the maximum allowed cell volume in voxels

std::size_t **getCompartmentMaxCellVolume**(std::size_t compartmentIndex) const

Get the maximum allowed cell volume for a given compartment.

Parameters
compartmentIndex – **[in]** the index of the compartment

const std::vector<std::size_t> &**getCompartmentMaxCellVolume**() const

The maximum allowed cell volume in voxels for each compartment in the mesh.

void **setPhysicalGeometry**(const *common*::VolumeF &voxelSize, const *common*::VoxelF &originPoint = {0.0, 0.0, 0.0})

The physical volume and origin to use.

The boundary lines and mesh use pixel units internally, and are rescaled to physical values using the supplied physical origin and pixel width.

Parameters

- **voxelSize** – **[in]** the physical size of a voxel
- **originPoint** – **[in]** the physical location of the (0,0,0) voxel

std::vector<double> **getVerticesAsFlatArray**() const

The physical mesh vertices as a flat array of doubles.

For saving to the SBML document.

std::vector<QVector4D> **getVerticesAsQVector4DArrayInHomogeneousCoord**() const

The physical mesh vertices as an array of QVector4D (homogeneous floating value)

Used as input in the rendering system

std::size_t **getNumberOfCompartment**() const

Returns
number of compartments available.

std::vector<int> **getTetrahedronIndicesAsFlatArray**(std::size_t compartmentIndex) const

The mesh tetrahedron indices as a flat array of ints.

For saving to the SBML document.

std::vector<uint32_t> **getMeshTrianglesIndicesAsFlatArray**(std::size_t compartmentIndex) const

A flat array of triangle indices for a particular compartment

Used by the rendering system.

const std::vector<std::vector<TetrahedronVertexIndices>> &**getTetrahedronIndices**() const

The mesh tetrahedron indices.

The indices of the vertices of each tetrahedron used in the mesh for each compartment

QString **getGMSH**() const

The mesh in GMSH format.

Returns
the mesh in GMSH format

const std::vector<QColor> &**getColors**() const

Get the colors of the compartments.

QVector3D **getOffset**() const

returns offset used for centering the mesh.

Public Static Attributes

static constexpr std::size_t **dim** = 3

23.2.2 src

Boundary

class **Boundary**

Approximate boundary line with adjustable number of points.

Given an ordered set of pixels that form a line or a closed loop, constructs an approximation to the line. The number of points used can be chosen automatically or set by the user.

Public Functions

bool **isLoop**() const

Is the line a closed loop.

bool **isValid**() const

Is the line valid.

const std::vector<QPoint> &**getPoints**() const

The simplified line.

const std::vector<QPoint> &**getAllPoints**() const

The original un-simplified line.

std::size_t **getMaxPoints**() const

The maximum number of points used by the approximate line.

void **setPoints**(std::vector<QPoint> &&simplifiedPoints)

Set the simplified line points explicitly.

void **setMaxPoints**(std::size_t maxPoints)

Set the maximum number of points to use in the approximate line.

std::size_t **setMaxPoints**()

Automatically choose and return the maximum number of points to use in the approximate line.

explicit **Boundary**(const std::vector<QPoint> &boundaryPoints, bool isClosedLoop = false)

Construct a simplified line from on ordered set of points.

Parameters

- **boundaryPoints** – **[in]** the boundary as an ordered set of points
- **isClosedLoop** – **[in]** true if the last point implicitly connects to the first point to form a closed loop

ContourMap

class **ContourMap**

Public Functions

ContourMap(const QSize &size, const *Contours* &contours)

const ContourIndices &**getContourIndices**(const cv::Point &p) const

bool **isFixedPoint**(const cv::Point &p) const

struct **Contours**

Public Members

std::vector<std::vector<cv::Point>> **compartmentEdges**

std::vector<std::vector<cv::Point>> **domainEdges**

getInteriorPoints

std::vector<std::vector<QPointF>> *sme*::mesh::**getInteriorPoints**(const QImage &img, const std::vector<QRgb> &cols)

Line Simplifier

class **LineSimplifier**

Simplify a line using Visvalingam-Whyatt polyline simplification.

Given an ordered set of pixels that form a line or a closed loop, construct an n-point approximation to the line.

Note: Each point in the line is ranked according to the size of the triangle formed by itself and its two nearest neighbours, and points are removed in reverse order of importance. See 10.1179/000870493786962263 for more details.

Public Functions

void **getSimplifiedLine**(std::vector<QPoint> &line, const *LineError* &allowedError = {0.0, 0.5}) const

Construct a simplified line given a maximum allowed error.

Parameters

- **line** – **[out]** the simplified lines
- **allowedError** – **[in]** the maximum allowed error

void **getSimplifiedLine**(std::vector<QPoint> &line, std::size_t nPoints) const

Construct a simplified line given a maximum number of points.

Parameters

- **line** – **[out]** the simplified lines
- **nPoints** – **[in]** the maximum allowed number of points

const std::vector<QPoint> &**getAllVertices**() const

The original un-simplified line.

std::size_t **maxPoints**() const

The number of points in the original un-simplified line.

bool **isValid**() const

Is the line valid.

bool **isLoop**() const

Is the line a closed loop.

explicit **LineSimplifier**(const std::vector<QPoint> &points, bool isClosedLoop = false)

Construct a simplified version of the supplied line.

Parameters

- **points** – **[in]** the ordered points that define the line
- **isClosedLoop** – **[in]** true if the line represents a closed loop, i.e. the last point implicitly connects to the first point

struct **LineError**

Public Members

double **total** = std::numeric_limits<double>::max()

double **average** = std::numeric_limits<double>::max()

PixelCornerIterator

class **PixelCornerIterator**

Iterate around the edge vertices of a pixel contour.

Given a closed loop 8-connected pixel contour generated by Suzuki contour tracing, this class iterates around the edge vertices of the pixels to to produce a 4-connected contour of vertices.

Note: The OpenCV function cv::findContours() implements Suzuki contour tracing.

Note: Edge vertices are shifted by (-1/2,-1/2) relative to the pixel centre, e.g the (0,0) vertex corresponds to the top-left corner of the (0,0) pixel.

Public Functions

PixelCornerIterator(const std::vector<cv::Point> &points, bool outer)

Iterate around the edge vertices of a pixel contour.

Given a closed loop 8-connected pixel contour generated by Suzuki contour tracing, this class iterates around the edge vertices of the pixels to to produce a 4-connected contour of vertices.

Parameters

- **points** – **[in]** 8-connected pixel contour as returned by cv::findContours()
- **outer** – **[in]** true if the contour is an outer contour (has no parent)

cv::Point **vertex**() const

The current vertex.

bool **done**() const

Returns true if at the end of the contour.

PixelCornerIterator &**operator++**()

Progress to the next vertex in the contour.

Triangulate

class **Triangulate**

Triangulate a set of boundary lines.

Given a set of boundary lines, a set of interior points for each compartment, and a maximum allowed triangle area for each compartment, constructs a Constrained Delauney Triangulation of the geometry, with the triangles labelled according to the compartment they belong to.

Public Functions

explicit **Triangulate**(const std::vector<*Boundary*> &boundaries, const std::vector<std::vector<QPointF>> &interiorPoints, const std::vector<std::size_t> &maxTriangleAreas)

Triangulate a set of boundary lines.

Given a set of boundary lines, a set of interior points for each compartment,

Parameters

- **boundaries** – **[in]** the boundary lines separating the compartments
- **interiorPoints** – **[in]** the interior point(s) for each compartment
- **maxTriangleAreas** – **[in]** the maximum allowed triangle area for each compartment

const std::vector<QPointF> &**getPoints**() const

The vertices or points in the mesh.

Returns
The vertices in the mesh

const std::vector<std::vector<TriangulateTriangleIndex>> &**getTriangleIndices**() const

The triangle vertex indices.

For each compartment, the vertices of the triangles that define this compartment in the mesh.

Returns
The triangle vertex indices

23.3 sme::simulate

Simulating the model, either with Pixel or dune-copasi.

23.3.1 inc

DuneConverter

class **DuneConverter**

Public Functions

explicit **DuneConverter**(const *model*::Model &model, const std::map<std::string, double, std::less<>> &substitutions = {}, bool forExternalUse = false, const QString &outputIniFile = {}, int doublePrecision = 18)

QString **getIniFile**() const

const mesh::*Mesh2d* ***getMesh**() const

const mesh::*Mesh3d* ***getMesh3d**() const

const std::unordered_map<std::string, std::vector<double>> &**getConcentrations**() const

const std::unordered_map<std::string, std::vector<std::string>> &**getSpeciesNames**() const

const std::vector<std::string> &**getCompartmentNames**() const

common::VoxelF **getOrigin**() const

common::VolumeF **getVoxelSize**() const

common::Volume **getImageSize**() const

Pde

class **Pde**

Public Functions

explicit **Pde**(const model::*Model* *doc_ptr, const std::vector<std::string> &speciesIDs, const std::vector<std::string> &reactionIDs, const std::vector<std::string> &relabelledSpeciesIDs = {}, const PdeScaleFactors &pdeScaleFactors = {}, const std::vector<std::string> &extraVariables = {}, const std::vector<std::string> &relabelledExtraVariables = {}, const std::map<std::string, double, std::less<>> &substitutions = {})

const std::vector<std::string> &**getRHS**() const

const std::vector<std::vector<std::string>> &**getJacobian**() const

Simulation

class **Simulation**

Public Functions

explicit **Simulation**(model::*Model* &smeModel)

~Simulation()

std::size_t **doTimesteps**(double time, std::size_t nSteps = 1, double timeout_ms = -1.0)

std::size_t **doMultipleTimesteps**(const std::vector<std::pair<std::size_t, double>> &timesteps, double timeout_ms = -1.0, const std::function<bool()> &stopRunningCallback = {})

const std::string &**errorMessage**() const

const *common*::ImageStack &**errorImages**() const

const std::vector<std::string> &**getCompartmentIds**() const

const std::vector<std::string> &**getSpeciesIds**(std::size_t compartmentIndex) const

const std::vector<QRgb> &**getSpeciesColors**(std::size_t compartmentIndex) const

const std::vector<double> &**getTimePoints**() const

const *AvgMinMax* &**getAvgMinMax**(std::size_t timeIndex, std::size_t compartmentIndex, std::size_t speciesIndex) const

std::vector<double> **getConc**(std::size_t timeIndex, std::size_t compartmentIndex, std::size_t speciesIndex) const

std::vector<double> **getConcArray**(std::size_t timeIndex, std::size_t compartmentIndex, std::size_t speciesIndex) const

void **applyConcsToModel**(model::*Model* &m, std::size_t timeIndex) const

std::vector<double> **getDcdt**(std::size_t compartmentIndex, std::size_t speciesIndex) const

std::vector<double> **getDcdtArray**(std::size_t compartmentIndex, std::size_t speciesIndex) const

double **getLowerOrderConc**(std::size_t compartmentIndex, std::size_t speciesIndex, std::size_t pixelIndex) const

common::ImageStack **getConcImage**(std::size_t timeIndex, const std::vector<std::vector<std::size_t>> &speciesToDraw = {}, bool normaliseOverAllTimepoints = false, bool normaliseOverAllSpecies = false) const

const std::vector<std::string> &**getPyNames**(std::size_t compartmentIndex) const

std::vector<std::vector<double>> **getPyConcs**(std::size_t timeIndex, std::size_t compartmentIndex) const

std::vector<std::vector<double>> **getPyDcdts**(std::size_t compartmentIndex) const

std::size_t **getNCompletedTimesteps**() const

const *SimulationData* &**getSimulationData**() const

bool **getIsRunning**() const

bool **getIsStopping**() const

void **requestStop**()

SimulationData

class **SimulationData**

Public Functions

void **clear**()

std::size_t **size**() const

void **reserve**(std::size_t n)

void **pop_back**()

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

std::vector<double> **timePoints**

std::vector<std::vector<std::vector<double>>> **concentration**

std::vector<std::vector<std::vector<*AvgMinMax*>>> **avgMinMax**

std::vector<std::vector<std::vector<double>>> **concentrationMax**

std::vector<std::size_t> **concPadding**

std::string **xmlModel**

struct **AvgMinMax**

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

double **avg** = 0

double **min** = std::numeric_limits<double>::max()

double **max** = 0

SimulationOptions

struct **Options**

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

DuneOptions **dune**

PixelOptions **pixel**

enum class *sme*::simulate::**SimulatorType**

Values:

enumerator **DUNE**

enumerator **Pixel**

struct **PixelOptions**

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

PixelIntegratorType **integrator** = {*PixelIntegratorType*::*RK212*}

PixelIntegratorError **maxErr**

double **maxTimestep** = {std::numeric_limits<double>::max()}

bool **enableMultiThreading** = {false}

std::size_t **maxThreads** = {0}

bool **doCSE** = {true}

unsigned **optLevel** = {3}

struct **PixelIntegratorError**

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

double **abs** = {std::numeric_limits<double>::max()}

double **rel** = {0.005}

enum class *sme*::simulate::**PixelIntegratorType**

Values:

enumerator **RK101**

enumerator **RK212**

enumerator **RK323**

enumerator **RK435**

struct **DuneOptions**

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

DuneDiscretizationType **discretization** = {*DuneDiscretizationType*::*FEM1*}

std::string **integrator** = {"Alexander2"}

double **dt** = {1e-1}

double **minDt** = {1e-10}

double **maxDt** = {1e4}

double **increase** = {1.5}

double **decrease** = {0.5}

bool **writeVTKfiles** = {false}

double **newtonRelErr** = {1e-8}

double **newtonAbsErr** = {0.0}

std::string **linearSolver** = {"RestartedGMRes"}

enum class *sme*::simulate::**DuneDiscretizationType**

Values:

enumerator **FEM1**

Optimization

class **Optimization**

Optimize model parameters.

Optimizes the supplied model parameters to minimize the supplied cost functions.

Public Functions

explicit **Optimization**(*sme*::model::*Model* &model)

Constructs an *Optimization* object from the supplied model.

Parameters
model – **[in]** the model to optimize

std::size_t **evolve**(std::size_t n = 1)

Do n iterations of parameter optimization.

bool **applyParametersToModel**(*sme*::model::*Model* *model) const

Apply the current best parameter values to the supplied model.

const std::vector<std::vector<double>> &**getParams**() const

The best set of parameters from each iteration.

std::vector<QString> **getParamNames**() const

The names of the parameters being optimized.

const std::vector<double> &**getFitness**() const

The best fitness from each iteration.

bool **setBestResults**(double fitness, std::vector<std::vector<double>> &&results)

Try to set a new set of best results for each target.

The best results are only updated if `fitness` is lower than the current `bestResultsFitness`

common::ImageStack **getTargetImage**(std::size_t index) const

Get an image of the a target.

std::optional<*common*::ImageStack> **getUpdatedBestResultImage**(std::size_t index)

Get an image of the current best result for a target.

std::size_t **getIterations**() const

The number of completed evolve iterations.

bool **getIsRunning**() const

True if the optimization is currently running.

bool **getIsStopping**() const

True if *requestStop()* has been called.

void **requestStop**()

Stop the evolution as soon as possible.

const std::string &**getErrorMessage**() const

Returns a message if an error occurred - empty if no errors occurred.

OptimizationOptions

struct **OptimizeOptions**

Optimization options.

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

OptAlgorithm **optAlgorithm**

The algorithm to use.

std::vector<*OptParam*> **optParams**

The parameters to optimize.

std::vector<*OptCost*> **optCosts**

The costs to minimize.

struct **OptParam**

Defines a parameter to be used in optimization.

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

OptParamType **optParamType**

The type of parameter.

std::string **name**

The name of this *OptParam* as displayed to the user.

std::string **id**

The id of the parameter in the model.

std::string **parentId**

The id of the parent of the parameter in the model (optional)

double **lowerBound**

The lower bound on the allowed values of the parameter.

double **upperBound**

The upper bound on the allowed values of the parameter.

enum class *sme*::simulate::**OptParamType**

Types of model parameters that can be used in optimization.

Values:

enumerator **ModelParameter**

enumerator **ReactionParameter**

struct **OptCost**

Defines a cost function to be minimized.

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

OptCostType **optCostType**

The type of cost function.

OptCostDiffType **optCostDiffType**

The type of difference (e.g. absolute, relative) used in the cost function.

std::string **name**

The name of this *OptCost* as displayed to the user.

std::string **id**

The id of the species in the model.

double **simulationTime** = {1.0}

The simulation time at which the cost function should be calculated.

double **weight** = {1.0}

The scale factor to multiply the cost by.

This assigns a relative weight to this cost when the optimization involves the sum of multiple cost functions.

std::size_t **compartmentIndex**

The index of the compartment containing the species.

std::size_t **speciesIndex**

The index of the species.

std::vector<double> **targetValues**

The target values to compare with the species concentration or dcdt.

Should contain a value for each pixel in the image, including those outside of the compartment the species is located in. If empty, the target values are assumed to be zero everywhere.

double **epsilon** = {1e-15}

A small number to avoid dividing by zero in relative differences.

A small value to be added to the denominator of relative differences to avoid numerical issues caused by dividing by zero.

enum class *sme*::simulate::**OptCostType**

Types of costs that can be used in optimization.

Values:

enumerator **Concentration**

enumerator **ConcentrationDcdt**

enum class *sme*::simulate::**OptCostDiffType**

Types of differences that can be used in costs.

Values:

enumerator **Absolute**

enumerator **Relative**

struct **OptAlgorithm**

Optimization algorithm options.

Public Functions

template<class **Archive**>
inline void **serialize**(*Archive* &ar, std::uint32_t const version)

Public Members

OptAlgorithmType **optAlgorithmType** = {*OptAlgorithmType*::*PSO*}

The algorithm to use.

std::size_t **islands** = {1}

The number of islands.

std::size_t **population** = {2}

The population size in each island.

enum class *sme*::simulate::**OptAlgorithmType**

Types of algorithms that can be used in optimization.

Values:

enumerator **PSO**

enumerator **GPSO**

enumerator **DE**

enumerator **iDE**

enumerator **jDE**

enumerator **pDE**

enumerator **ABC**

enumerator **gaco**

23.3.2 src

BaseSim

class **BaseSim**

Subclassed by *sme::simulate::DuneSim*, *sme::simulate::PixelSim*

Public Functions

virtual **~BaseSim**() = default

virtual std::size_t **run**(double time, double timeout_ms, const std::function<bool()> &stopRunningCallback) = 0

virtual const std::vector<double> &**getConcentrations**(std::size_t compartmentIndex) const = 0

virtual std::size_t **getConcentrationPadding**() const = 0

virtual const std::string &**errorMessage**() const = 0

virtual const *common*::ImageStack &**errorImages**() const = 0

virtual void **setStopRequested**(bool stop) = 0

GridFunction

Warning: doxygenclass: Cannot find class "sme::simulate::GridFunction" in doxygen xml output for project "sme" from directory: build/xml

IniFile

class **IniFile**

Public Functions

const QString &**getText**() const

void **addSection**(const QString &str)

void **addSection**(const QString &str1, const QString &str2)

void **addSection**(const QString &str1, const QString &str2, const QString &str3)

void **addSection**(const QString &str1, const QString &str2, const QString &str3, const QString &str4)

void **addSection**(const QString &str1, const QString &str2, const QString &str3, const QString &str4, const QString &str5)

void **addSection**(const QString &str1, const QString &str2, const QString &str3, const QString &str4, const QString &str5, const QString &str6)

void **addValue**(const QString &var, const QString &value)

void **addValue**(const QString &var, int value)

void **addValue**(const QString &var, double value, int precision)

void **clear**()

DuneSim

class **DuneSim** : public *sme*::simulate::*BaseSim*

Public Functions

explicit **DuneSim**(const model::*Model* &sbmlDoc, const std::vector<std::string> &compartmentIds, const std::map<std::string, double, std::less<>> &substitutions = {})

~DuneSim() override

virtual std::size_t **run**(double time, double timeout_ms, const std::function<bool()> &stopRunningCallback) override

virtual const std::vector<double> &**getConcentrations**(std::size_t compartmentIndex) const override

virtual std::size_t **getConcentrationPadding**() const override

virtual const std::string &**errorMessage**() const override

virtual const *common*::ImageStack &**errorImages**() const override

virtual void **setStopRequested**(bool stop) override

DuneImpl

template<int **DuneDimensions**>

class **DuneImpl**

Public Functions

inline explicit **DuneImpl**(const *DuneConverter* &dc, const *DuneOptions* &options, const model::*Model* &sbmlDoc, const std::vector<std::string> &compartmentIds)

~DuneImpl() = default

inline void **run**(double time)

inline const std::vector<double> &**getConcentrations**(std::size_t compartmentIndex) const

PixelSim

class **PixelSim** : public *sme*::simulate::*BaseSim*

Public Functions

explicit **PixelSim**(const model::*Model* &sbmlDoc, const std::vector<std::string> &compartmentIds, const std::vector<std::vector<std::string>> &compartmentSpeciesIds, const std::map<std::string, double, std::less<>> &substitutions = {})

~PixelSim() override

virtual std::size_t **run**(double time, double timeout_ms, const std::function<bool()> &stopRunningCallback) override

virtual const std::vector<double> &**getConcentrations**(std::size_t compartmentIndex) const override

virtual std::size_t **getConcentrationPadding**() const override

const std::vector<double> &**getDcdt**(std::size_t compartmentIndex) const

double **getLowerOrderConcentration**(std::size_t compartmentIndex, std::size_t speciesIndex, std::size_t pixelIndex) const

virtual const std::string &**errorMessage**() const override

virtual const *common*::ImageStack &**errorImages**() const override

virtual void **setStopRequested**(bool stop) override

PagmoUDP

class **PagmoUDP**

Implements a Pagmo User Defined Problem to evolve.

Note: Needs to be (cheaply) copy-constructible, implement `fitness()` and `get_bounds()` functions, and be thread-safe, see https://esa.github.io/pagmo2/docs/cpp/problem.html

---

Public Functions

PagmoUDP() = default

explicit **PagmoUDP**(const OptConstData *optConstData, ThreadsafeModelQueue *modelQueue, *Optimization* *optimization)

pagmo::vector_double **fitness**(const pagmo::vector_double &dv) const

std::pair<pagmo::vector_double, pagmo::vector_double> **get_bounds**() const

23.4 sme::common

Symbolic math, TIFF import/export, other utility functions.

23.4.1 inc

Symbolic

class **Symbolic**

Public Functions

Symbolic()

explicit **Symbolic**(const std::vector<std::string> &expressions, const std::vector<std::string> &variables = {}, const std::vector<std::pair<std::string, double>> &constants = {}, const std::vector<SymbolicFunction> &functions = {}, bool allow_unknown_symbols = false)

explicit **Symbolic**(const std::string &expression, const std::vector<std::string> &variables = {}, const std::vector<std::pair<std::string, double>> &constants = {}, const std::vector<SymbolicFunction> &functions = {}, bool allow_unknown_symbols = false)

bool **parse**(const std::vector<std::string> &expressions, const std::vector<std::string> &variables = {}, const std::vector<std::pair<std::string, double>> &constants = {}, const std::vector<SymbolicFunction> &functions = {}, bool allow_unknown_symbols = false)

bool **parse**(const std::string &expression, const std::vector<std::string> &variables = {}, const std::vector<std::pair<std::string, double>> &constants = {}, const std::vector<SymbolicFunction> &functions = {}, bool allow_unknown_symbols = false)

bool **compile**(bool doCSE = true, unsigned optLevel = 3)

std::string **expr**(std::size_t i = 0) const

std::string **inlinedExpr**(std::size_t i = 0) const

std::string **diff**(const std::string &var, std::size_t i = 0) const

void **relabel**(const std::vector<std::string> &newVariables)

void **rescale**(double factor, const std::vector<std::string> &exclusions = {})

void **eval**(std::vector<double> &results, const std::vector<double> &vars = {}) const

void **eval**(double *results, const double *vars) const

bool **isValid**() const

bool **isCompiled**() const

const std::string &**getErrorMessage**() const

void **clear**()

TiffReader

class **TiffReader**

Public Functions

explicit **TiffReader**(const std::string &filename)

bool **empty**() const

sme::*common*::ImageStack **getImages**() const

const QString &**getErrorMessage**() const

serialization.hpp

namespace **sme**

namespace **common**

struct **SmeFileContents**

Public Members

std::string **xmlModel** = {}

std::unique_ptr<simulate::*SimulationData*> **simulationData** = {}

utils.hpp

namespace **sme**

namespace **common**

Typedefs

template<typename **T**>

using **unique_C_ptr** = std::unique_ptr<*T*, *free_deleter*>

Functions

template<typename **Container**>
Container::value_type **sum**(const *Container* &c)

The sum of all elements in a container.

template<typename **Container**>
Container::value_type **average**(const *Container* &c)

The average of all elements in a container.

template<typename **Container**>
Container::value_type **min**(const *Container* &c)

The minimum value in a container.

template<typename **Container**>
Container::value_type **max**(const *Container* &c)

The maximum value in a container.

template<typename **Container**>
std::vector<typename *Container*::value_type> **get_unique_values**(const *Container* &c)

The unique values from a container.

template<typename **Container**>
bool **isItIndexes**(const *Container* &c, std::size_t length)

Are the numbers in the container indexes?

template<typename **Container**>
std::pair<typename *Container*::value_type, typename *Container*::value_type> **minmax**(const *Container* &c)

The minimum and maximum values in a container.

template<typename **Container**, typename **Element**>
std::size_t **element_index**(const *Container* &c, const *Element* &e, std::size_t index_if_not_found = 0)

The index in the container of the matching element.

template<typename **T**>
constexpr auto **decltypeStr**()

The type of an object as a string.

Note: Based on https://stackoverflow.com/a/56766138

---

template<typename **T**>
std::vector<*T*> **stringToVector**(const std::string &str)

Convert a string to a vector of values.

The values in the string are separated by spaces.

Template Parameters

T – the type of the values

template<typename **T**>
std::string **vectorToString**(const std::vector<*T*> &vec)

Convert a vector of values to a string.

The values in the string are separated by spaces. Doubles are printed in scientific notation with 17 significant digits.

Template Parameters
T – the type of the values

template<typename **T**>
void **cyclicErase**(std::vector<*T*> &v, std::size_t first, std::size_t last)

Cyclic erase of elements from a vector.

Erase the elements with index [first, last) from the vector v, with cyclic indices, i.e. the next element after the last element is the first element of the vector

Template Parameters
T – the value type

template<typename **T**>
bool **isCyclicPermutation**(const std::vector<*T*> &a, const std::vector<*T*> &b)

Check if a vector is a cyclic permutation of another vector.

Note: Assumes the elements of the vector are unique

Template Parameters
T – the value type

class **indexedColours**

#include <utils.hpp> Default set of colours.

A vector of default colours

Public Functions

const QColor &**operator[]**(std::size_t i) const

Private Static Attributes

static const std::vector<QColor> **colours** = std::vector<QColor>{{230, 25, 75}, {60, 180, 75}, {255, 225, 25}, {0, 130, 200}, {245, 130, 48}, {145, 30, 180}, {70, 240, 240}, {240, 50, 230}, {210, 245, 60}, {250, 190, 190}, {0, 128, 128}, {230, 190, 255}, {170, 110, 40}, {255, 250, 200}, {128, 0, 0}, {170, 255, 195}, {128, 128, 0}, {255, 215, 180}, {0, 0, 128}, {128, 128, 128}}

struct **free_deleter**

Public Functions

template<typename **T**>
inline void **operator()**(*T* *p) const

23.4.2 src

CHAPTER

TWENTYFOUR

TESTS

The tests for a component `X.hpp` are located next to the corresponding `X.cpp` file in `X_t.cpp`.

Generally they are self contained, but some require additional explanation, which is given here:

24.1 Diffusion

24.1.1 2d 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.

24.1.2 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 initial condition can then be found with an overall rescaling and a shift in t:

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

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,

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

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

24.1.3 3d Model

The example model single-compartment-diffusion-3d is a single compartment that contains two species: 'fast' and 'slow', each with the same analytic initial distribution.

Key differences from the 2d example above are that the origin of the geometry lies in the centre of a 100cm^3 cube, with initial species concentration given by:

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

and the heat kernel in 3d is given by:

$$c_s(t) = \frac{1}{(4\pi D_s t)^{3/2}} e^{-(x^2+y^2+z^2)/(4D_s t)}$$

which gives the solution at time t (again ignoring boundary effects):

$$c_s(t) = (\frac{t_0}{t+t_0})^{3/2} e^{-(x^2+y^2+z^2)/(4D_s(t+t_0))}$$

where $t_0 = 9/D_s$, with total amount of species in the compartment a conserved quantity:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} c_s(t)dxdydz = (36\pi)^{3/2}$$

PYTHON MODULE INDEX

INDEX

Symbols

A

C

D

E

G

I

M

N

P

R

S

T

U

V