spatial-model-editor

Sep 10, 2020

Getting started

1 Getting Started **1**
1.1 Installation . . . 1

2 Importing a Model **3**

3 Importing Geometry **5**

4 Mesh generation **7**

5 Species properties **9**

6 Running a Simulation **11**

7 dune-copasi simulator **13**
7.1 Simulation options . . . 13

8 Pixel simulator **15**
8.1 Simulation options . . . 15
8.2 Spatial discretization . . . 17
8.3 Time integration . . . 17
8.4 Adaptive timestep . . . 20
8.5 Maximum timestep . . . 21
8.6 Boundary Conditions . . . 21

9 Mesh generation **23**
9.1 Contour detection . . . 23
9.2 Contour splitting . . . 26
9.3 Contour merging . . . 27
9.4 End point merging . . . 28
9.5 Boundary line simplification . . . 29
9.6 Steiner points . . . 30
9.7 End point splitting . . . 31
9.8 Membrane boundaries . . . 32
9.9 Mesh generation . . . 33

10 Python Interface **35**

11 Command Line Interface **37**

11.1 Use . . . 37
11.2 Command line parameters . . . 37

12 Maths **39**
12.1 Reaction-Diffusion . . . 39
12.2 Compartment Reactions . . . 39
12.3 Membrane reactions . . . 40
12.4 Boundary Conditions . . . 40

13 Units **41**
13.1 Fundamental Units . . . 41
13.2 Derived Units . . . 41

14 Source Code **43**

15 Diffusion **45**
15.1 Model . . . 45
15.2 Analytic solution . . . 45

CHAPTER 1

Getting Started

1.1 Installation

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

-  linux
-  macOS
-  windows

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

CHAPTER 2

Importing a Model

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

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

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

CHAPTER 3

Importing Geometry

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

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

For each compartment in the model:

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

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

CHAPTER 4

Mesh generation

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

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

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

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

CHAPTER 5

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 6

Running a Simulation

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

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

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

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

CHAPTER 7

dune-copasi simulator

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

7.1 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

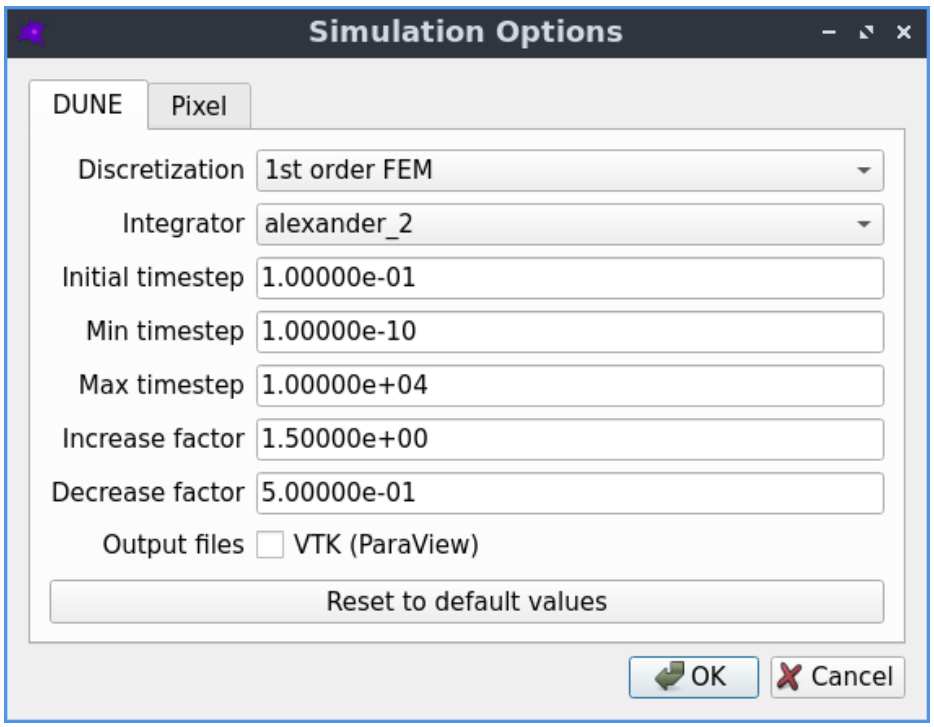

Fig. 1: The simulation options that can be used to fine tune-the dune-copasi solver.

- **Increase factor**
  - after a successful integration step, the timestep is multiplied by this factor
  - this must be greater than or equal to 1
  - if equal to 1, the timestep is never increased between integration steps
  - the larger the value, the more the timestep is increased after successful integration steps
- **Decrease factor**
  - if an integration step is unsuccessful, the timestep is multiplied by this factor and the step is repeated
  - this must be less than 1
  - the smaller the value, the more the timestep is decreased in the case of an unsuccessful integration step
- **Output files**
  - VTK files of the species concentrations throughout the simulation can be generated
  - these files can be viewed using ParaView

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

CHAPTER 8

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.

8.1 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

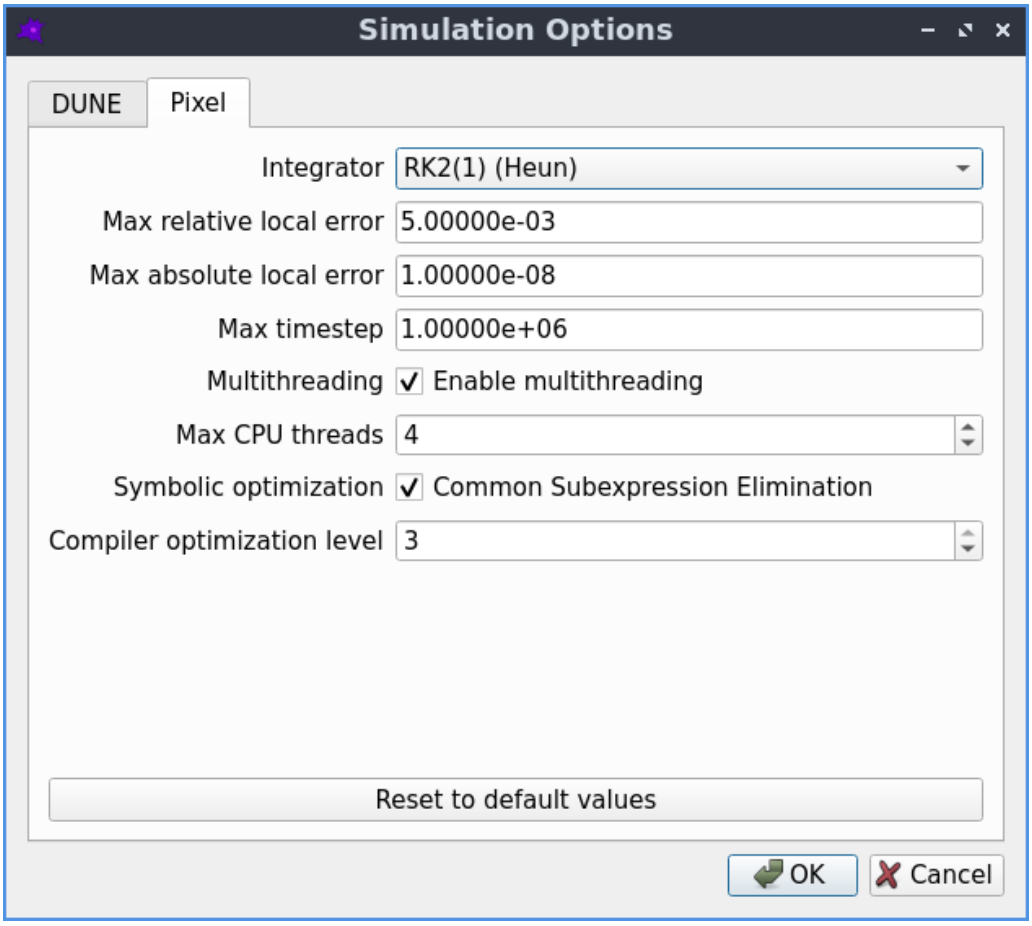

Fig. 1: The simulation options that can be used to fine tune-the Pixel solver.

  - 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**
  - 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

8.2 Spatial discretization

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

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

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

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

8.3 Time integration

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

Integrators differ in their:

- order of truncation error

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

Implemented integrators:

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

These integrators have three sources of error:

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

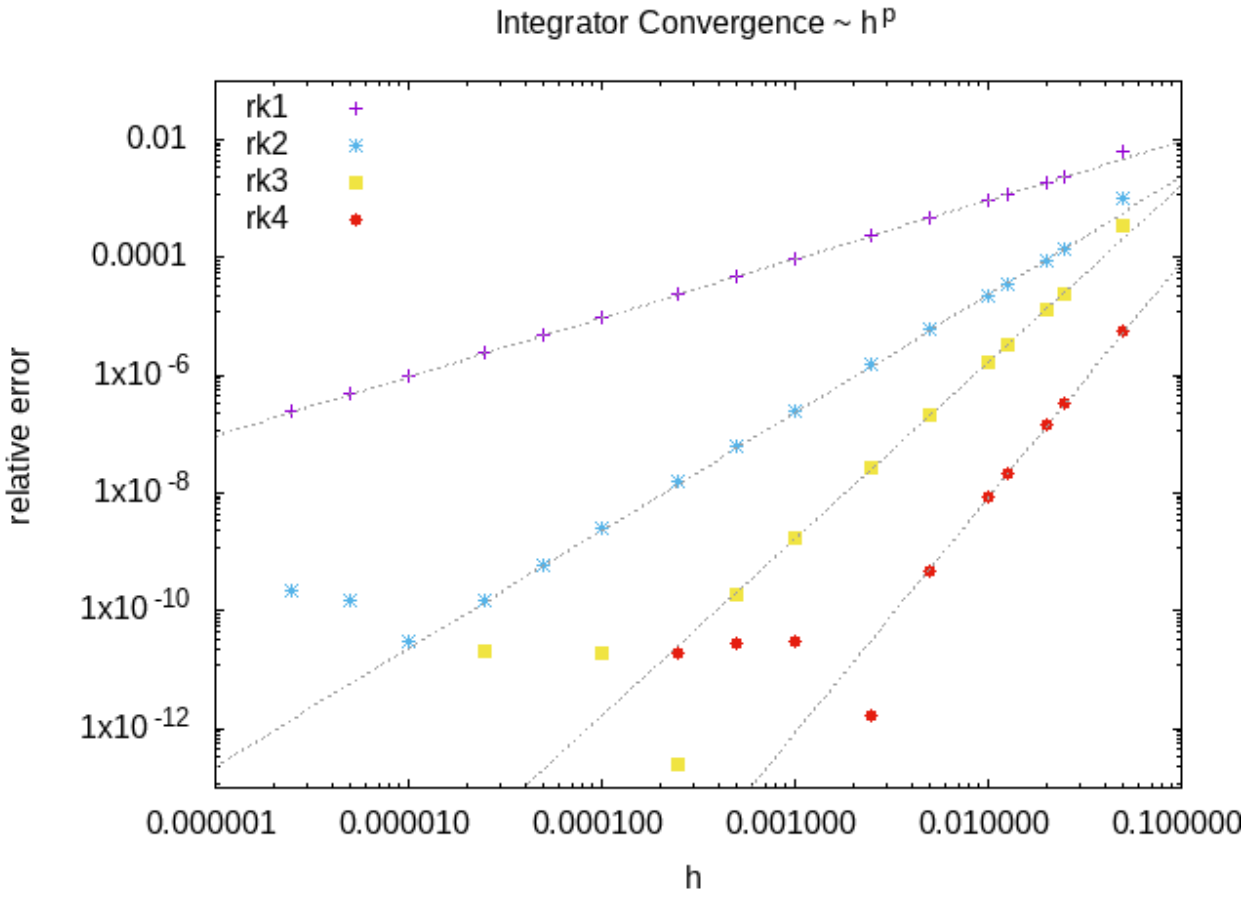

Fig. 2: An example of the convergence of the included RK integrators: relative error of the solution at a particular pixel as a function of the stepsize.

8.4 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

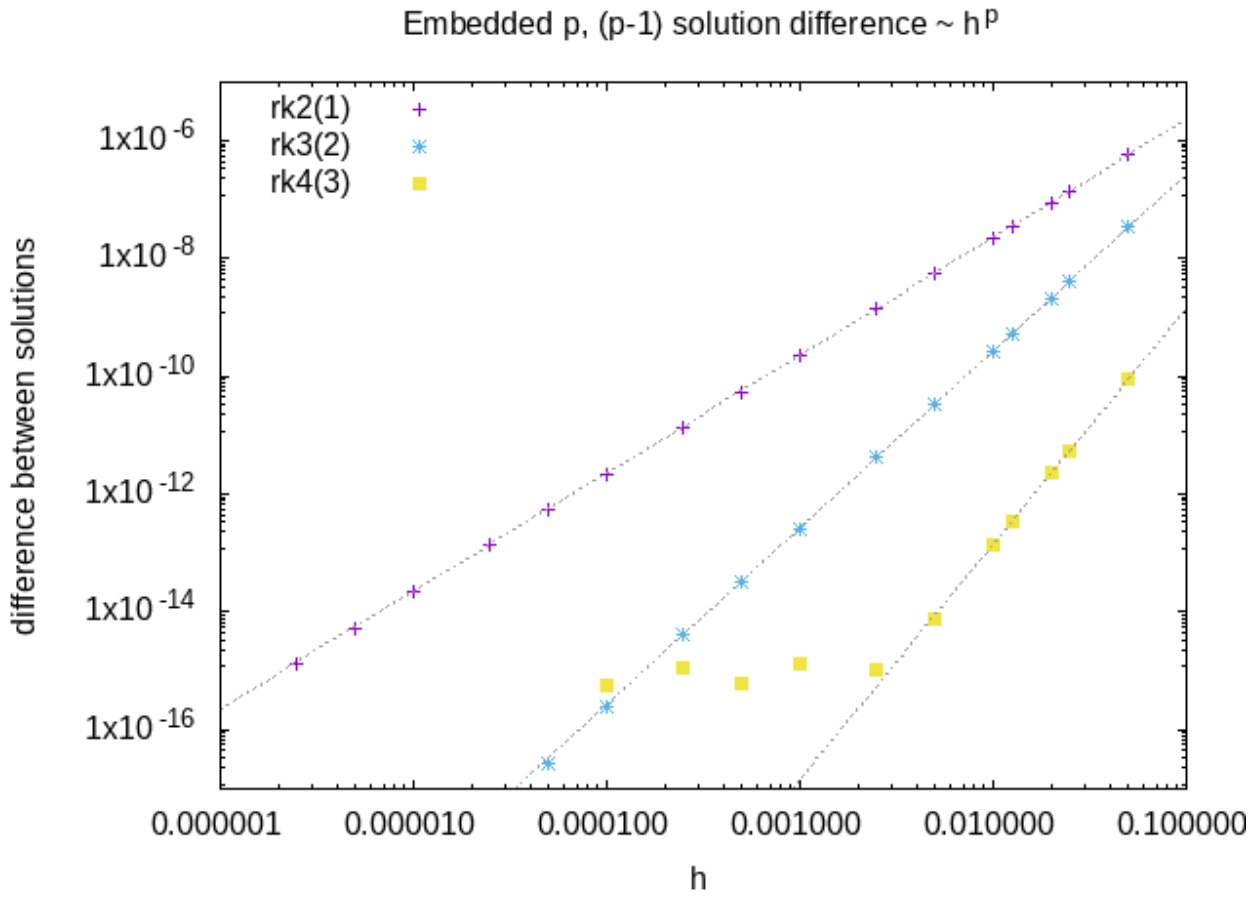

Fig. 3: An example of the difference between order p and order p-1 solutions from embedded schemes as a function of the stepsize. This quantity is a measure of the local integration error, and scales like h^p

8.5 Maximum timestep

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

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

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

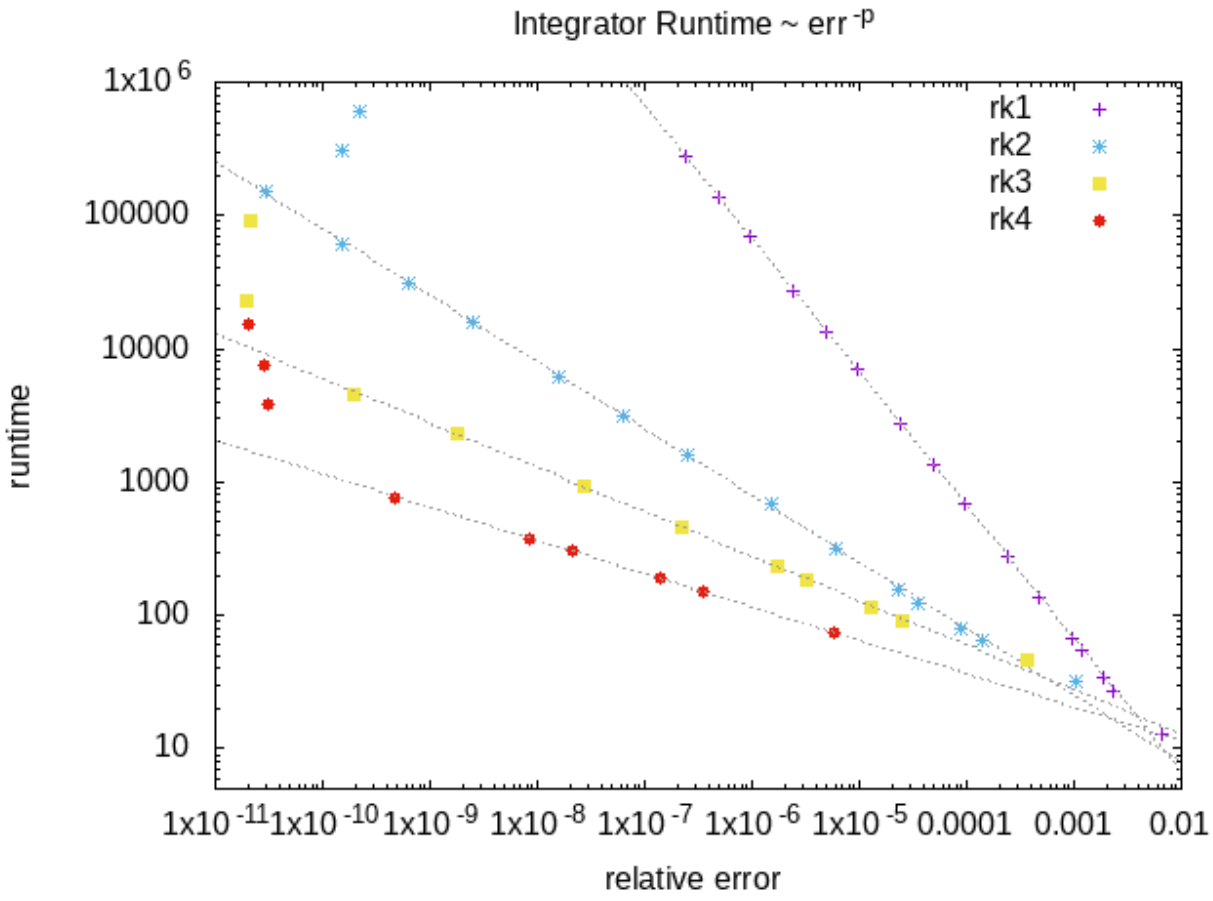

Fig. 4: An example of the runtime of the RK integrators as a function of the relative error on the final solution. The higher order integrators offer better performance if a very accurate solution is required, but at lower accuracy the lower order integrators are much faster.

8.6 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.

8.6.1 Compartments

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

8.6.2 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.

8.6.3 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 9

Mesh generation

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

- identify contours of compartment boundaries
- split contours into lines
- merge adjacent lines
- merge adjacent end points
- simplify boundary lines by removing points
- add Steiner points
- split membrane end points
- generate membrane compartments
- triangulate

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

9.1 Contour detection

The first step in generating the mesh is to identify the set of contours that make up the boundaries of each compartment. Each contour is a closed loop of connected pixels that makes up a boundary between a compartment and the rest of the model. 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.

The contour tracing is done using the findContours routine from the OpenCV library, which implements the method described in Suzuki et. al.. This method returns an ordered set of pixels for each contour. The outer contour traces the outer boundary in the anti-clockwise direction, while the inner contours trace each inner boundary in a clockwise direction.

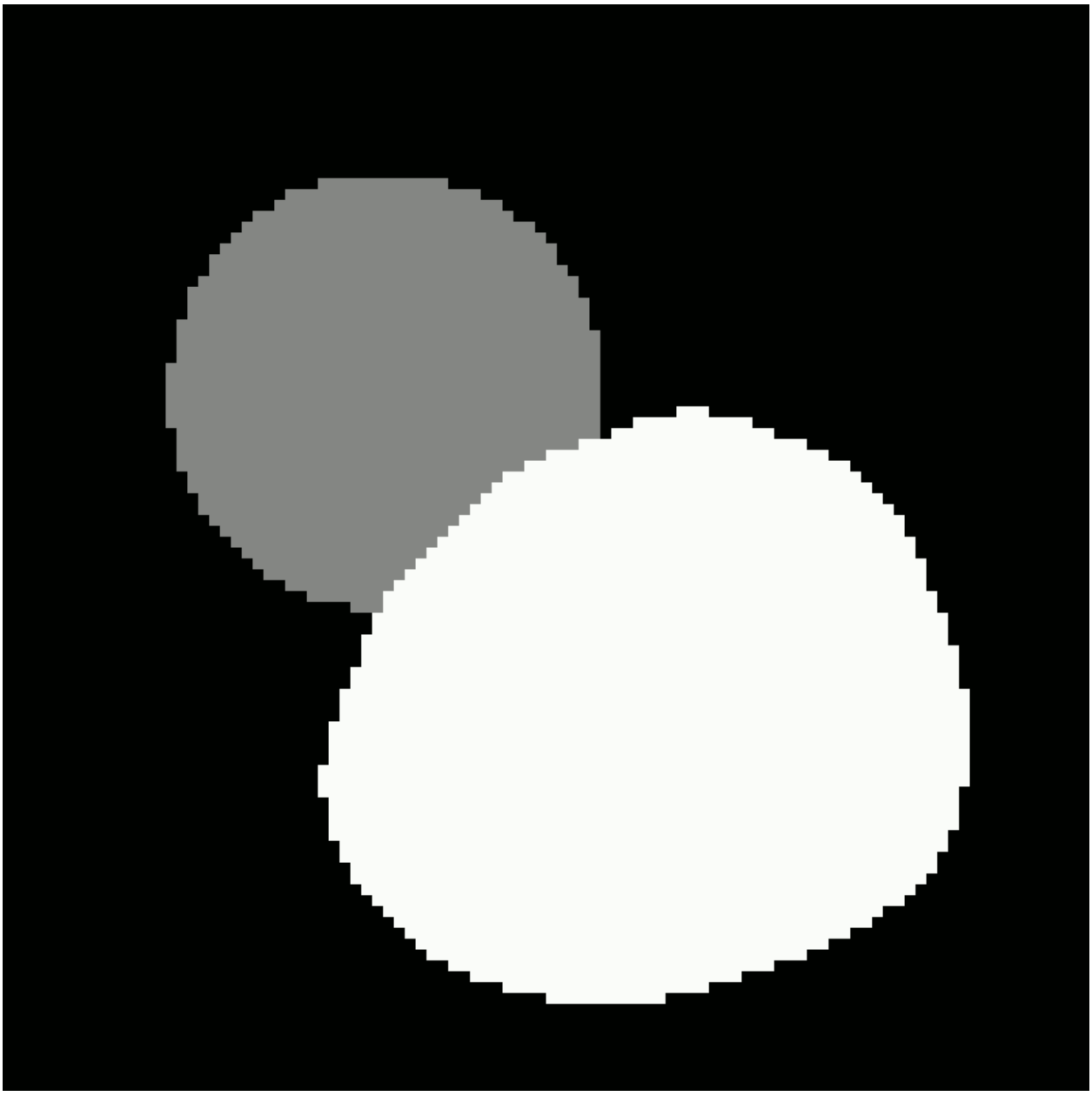

Fig. 1: Initial geometry image.

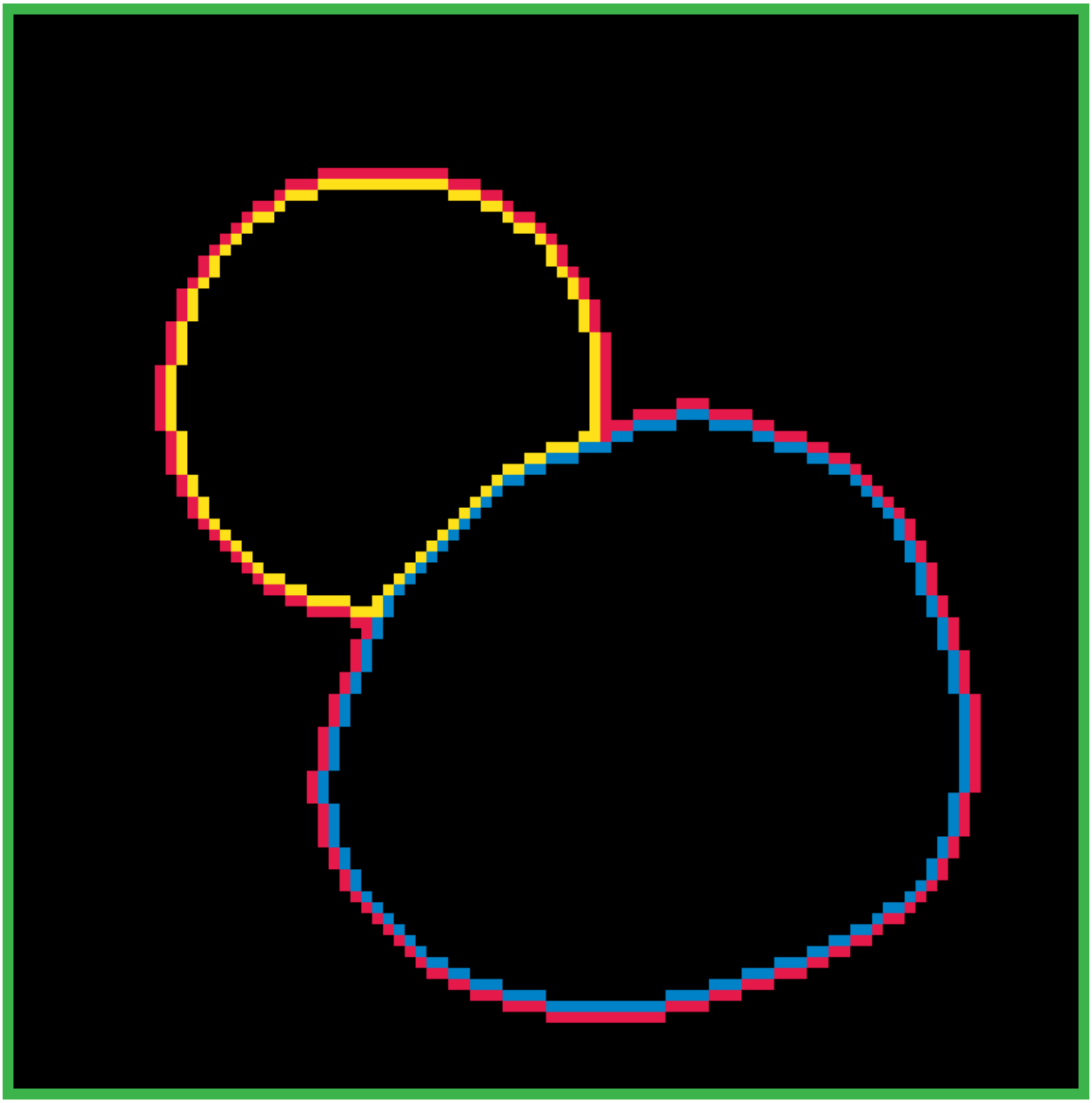

Fig. 2: Boundary contours.

9.2 Contour splitting

The contours are then split into lines which have a consistent neighbouring contour (including not having a neighbouring contour). For example, if part of contour *A* is adjacent to contour *B*, and the rest of contour A has no neighbouring contours, it will be split into two sections, one line for the part adjacent to contour *B*, and another line for the rest of the contour.

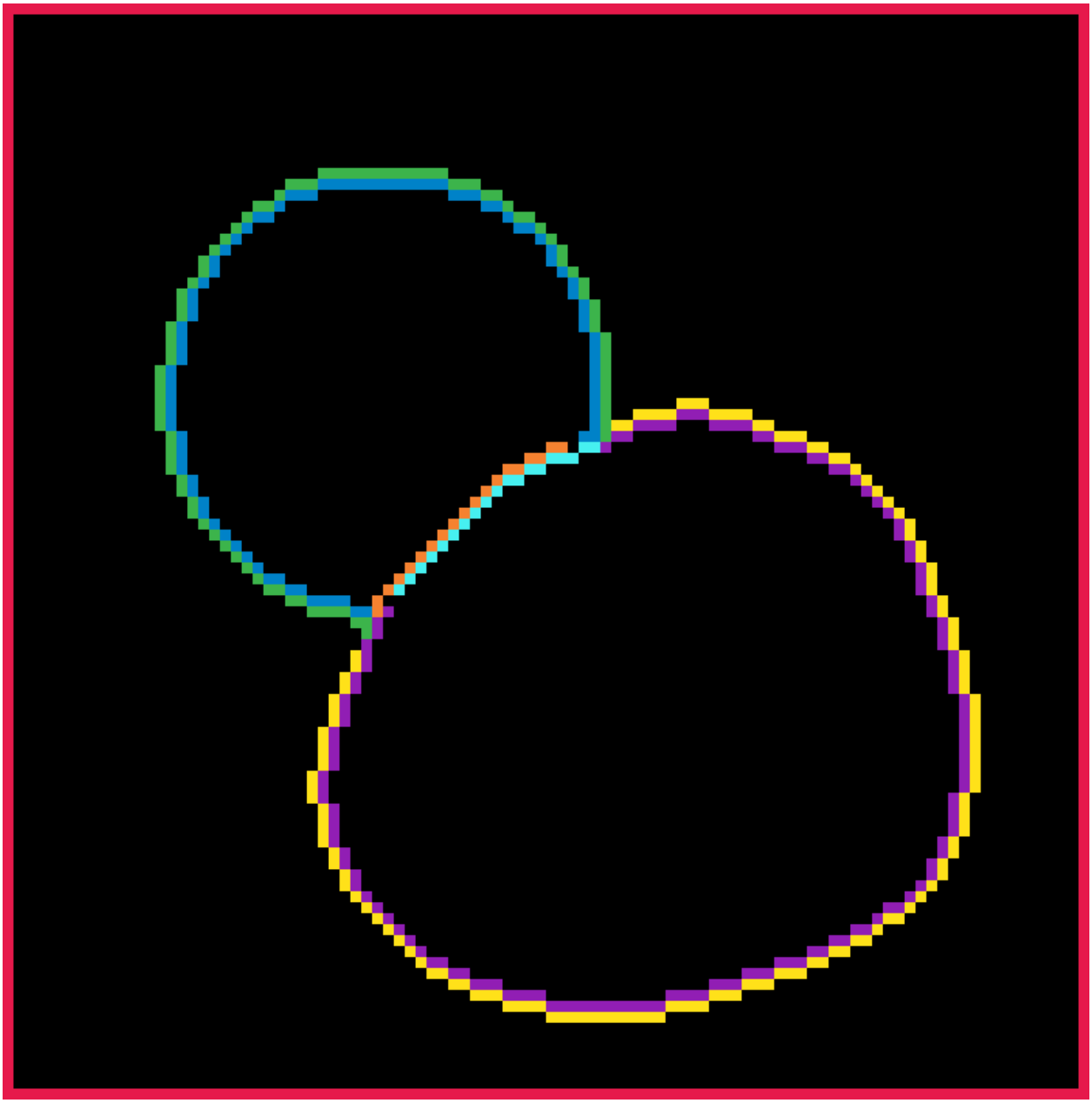

Fig. 3: Split boundary contours.

9.3 Contour merging

Next, adjacent contours need to be merged into a single boundary line. Adjacent contours are determined as the section of the appropriate contour with the smallest distance between the oriented pixels, and once a pair has been identified, one of the lines is removed to leave a single boundary line.

At the end of this step, we are left with a collection of boundary lines, some of which are closed loops, some not. All instances of adjacent contours have been replaced with a single boundary line of some kind.

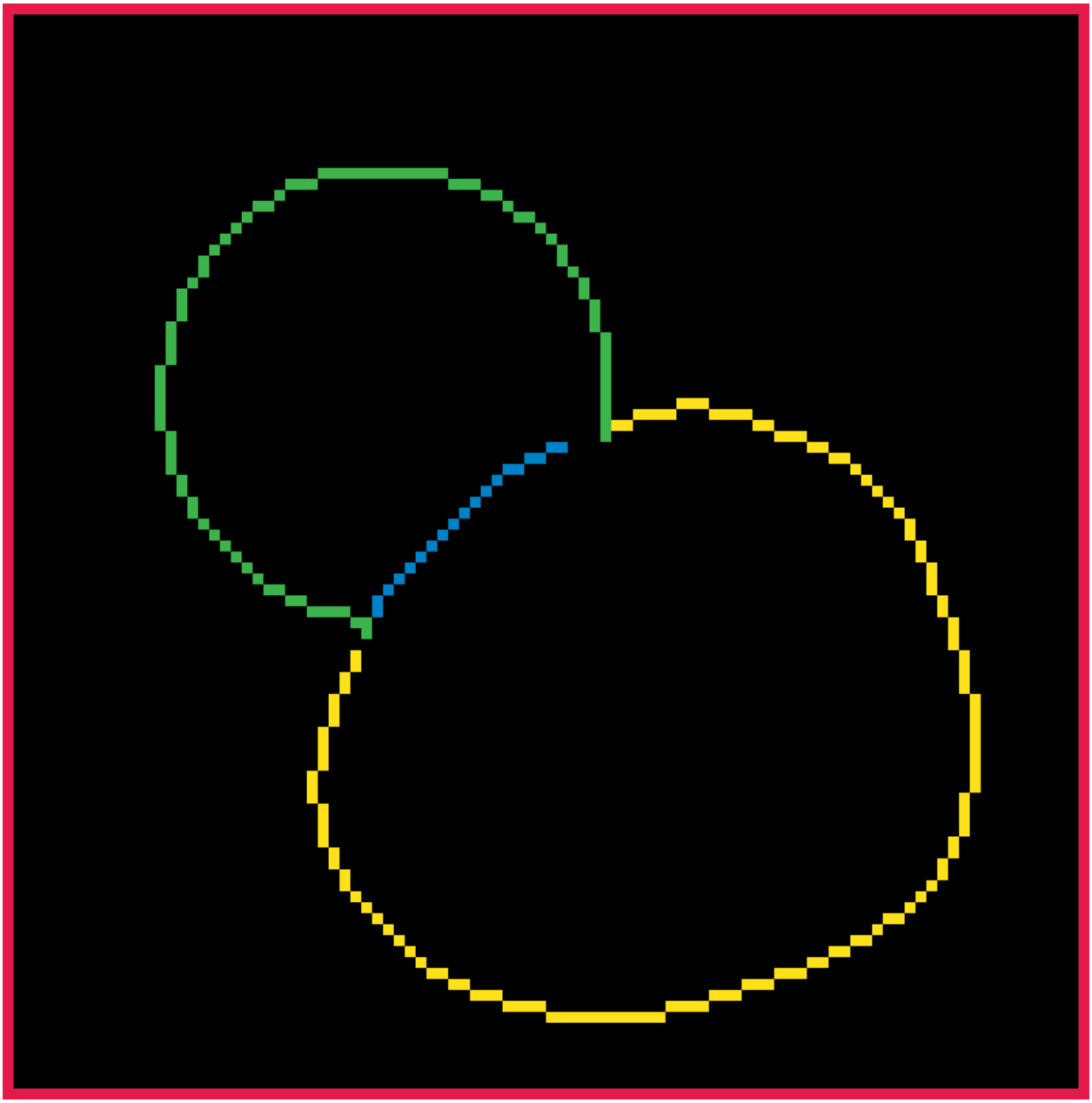

Fig. 4: Split and merged boundary contours.

9.4 End point merging

At places where three compartments meet, we would like to have three non-loop boundary lines meeting at a point. However, from the previous step, we will have three such lines that terminate near to each other, but typically separated by a pixel or two.

Here we take an end point, then determine the two closest other end points to it, and merge the three points. This is done until all such end point triplets have been merged. If the new end point is not adjacent to the end of the line, pixels are added in a straight line to connect the line to its new end point.

Note: currently we assume all end points come in triplets - ideally we should take into account possible 2- or 4-point merging.

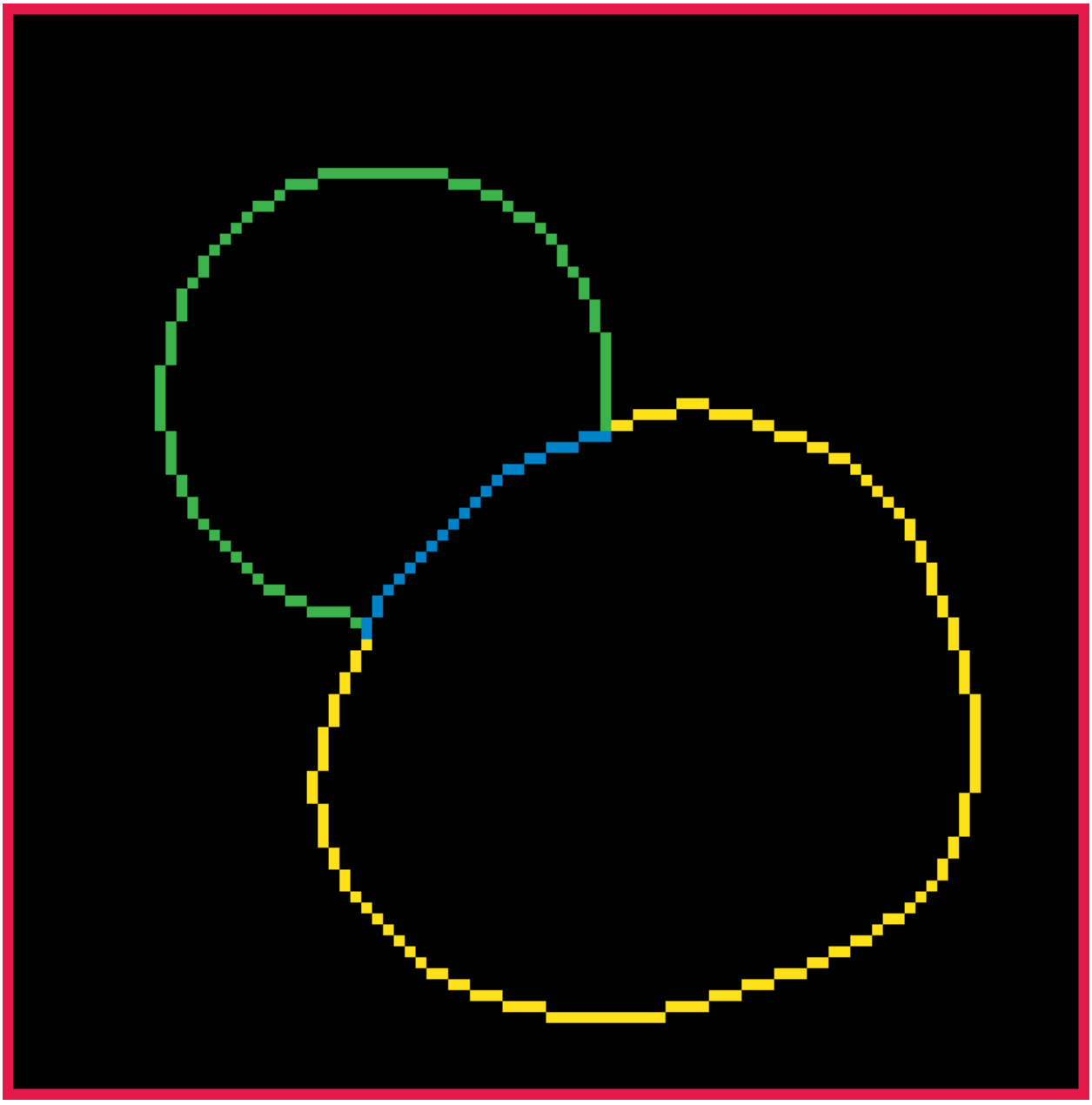

Fig. 5: Merged boundary contours.

9.5 Boundary line simplification

Once we have identified all of our boundaries, we want to simplify them by removing points from the boundary. We do this using Visvalingam-Whyatt polyline simplification. The algorithm starts by calculating the area of the triangle formed by each point on the boundary with its two nearest neighbouring points. Then at each step: - the point with the smallest area is removed - the two neighbouring points areas are recalculated - the larger of the previous and the new area is used This allows us to order the points in the boundary by their order of importance, and then the user can adjust the number of points used for each boundary according to how accurately they wish the represent the original boundaries.

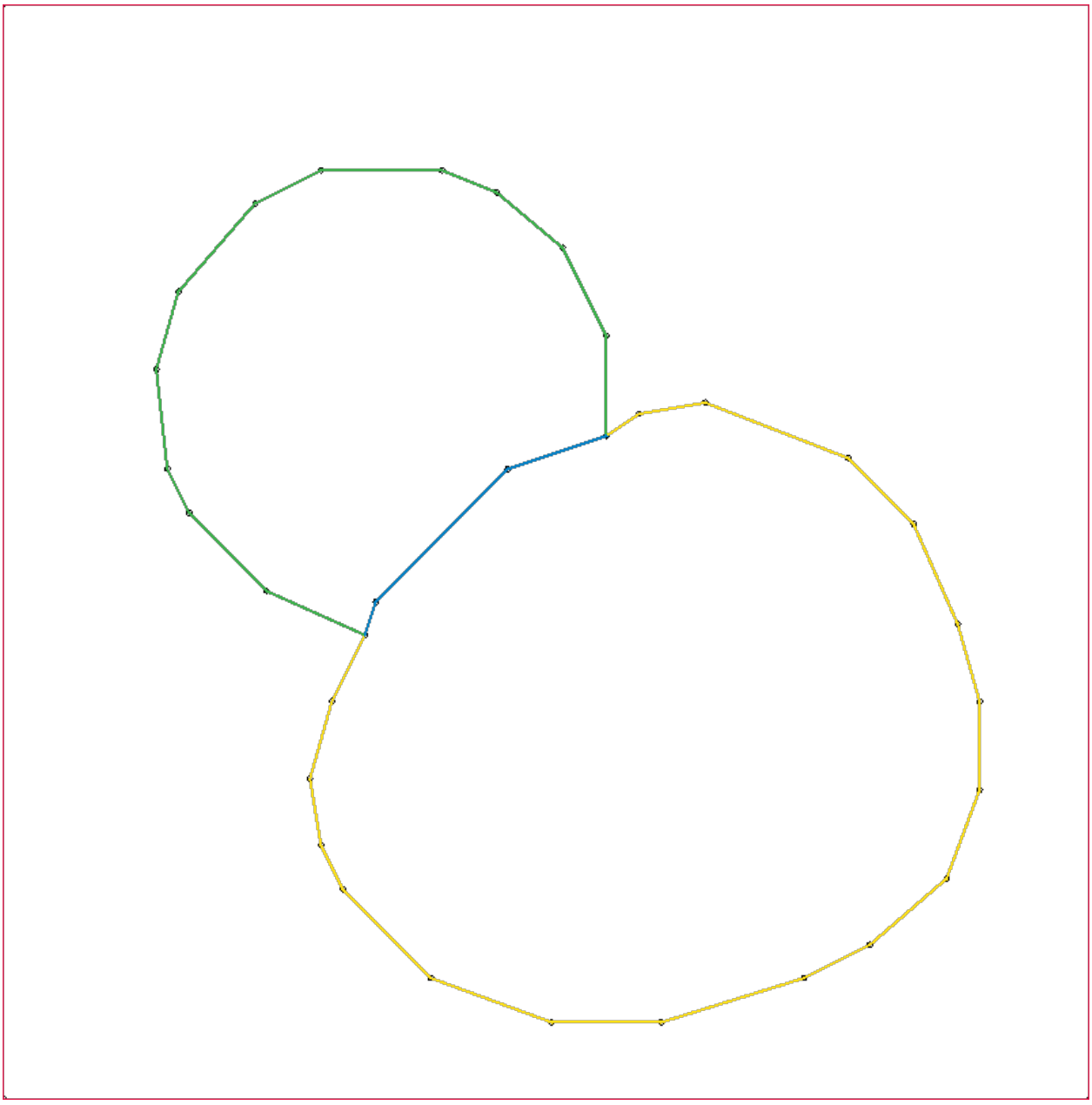

Fig. 6: Simplified boundary lines.

9.6 Steiner points

A long straight section of boundary will only have a start and end point. When we triangulate the area, we may need to insert more boundary points along these straight sections to satisfy limits on maximum triangle area. These points do not change the shape of the boundary, and are known as *Steiner points*.

Currently we determine the need for such points by performing a preliminary triangulation using our current boundary lines, and adding any Steiner points that were inserted as part of the triangulation. We use the Triangle library to generate the triangular mesh. This generates a constrained conforming Delaunay triangulation (CCDT) from the boundary lines, by inserting points inside the compartments and triangulating them. If necessary it will also add aditional points on the boundary lines (known as Steiner points).

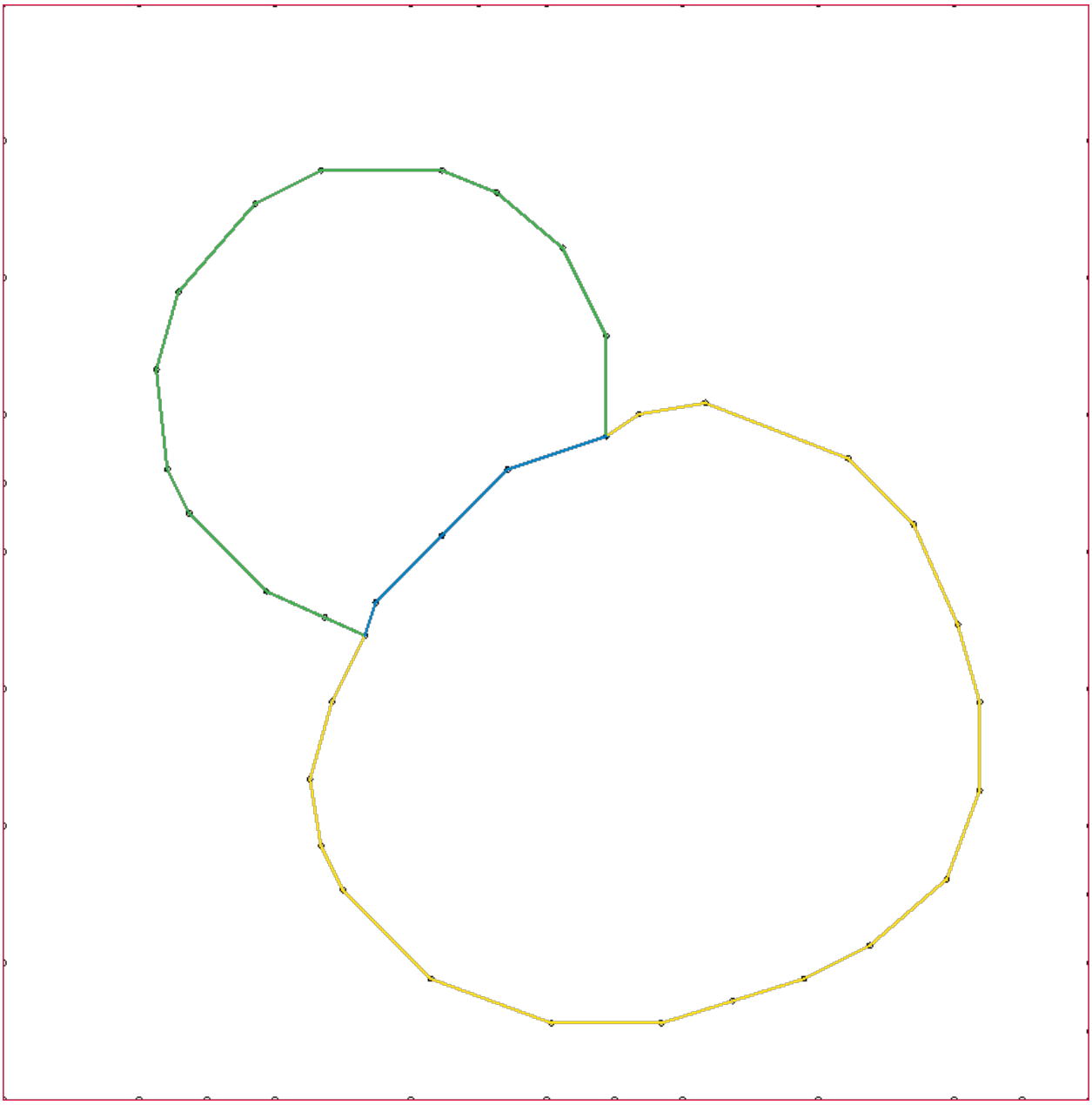

Fig. 7: Simplified boundary lines with Steiner points added.

9.7 End point splitting

We currently require membrane boundaries to be represented by a quadrilateral mesh of single-element thickness. To construct the boundary lines for these membrane compartments, we first replace each end point of a membrane boundary line with a triplet of points equally distributed around the original point. The triangle formed by connecting these three points will form a void around the original end point in the final mesh, and is where the three membrane compartments will meet.

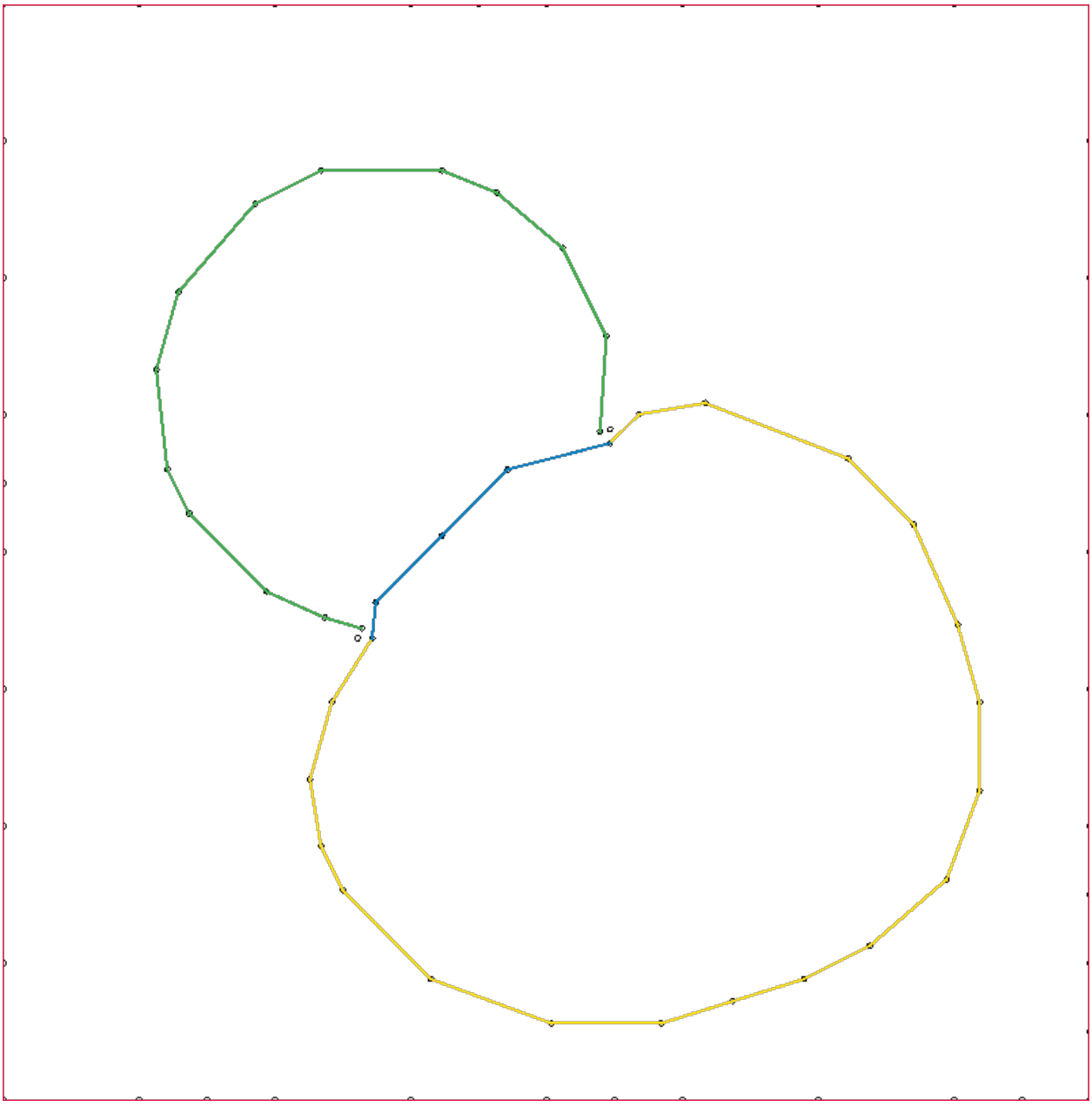

Fig. 8: Simplified boundary lines with split end points.

9.8 Membrane boundaries

Now we can duplicate and shift each membrane boundary line in a perpendicular direction to form the boundaries of the membrane compartment, and then connect the start and end points of each of these lines to the appropriate point in the new triplet of points around each previous end point. They then form the boundaries of the membrane compartments.

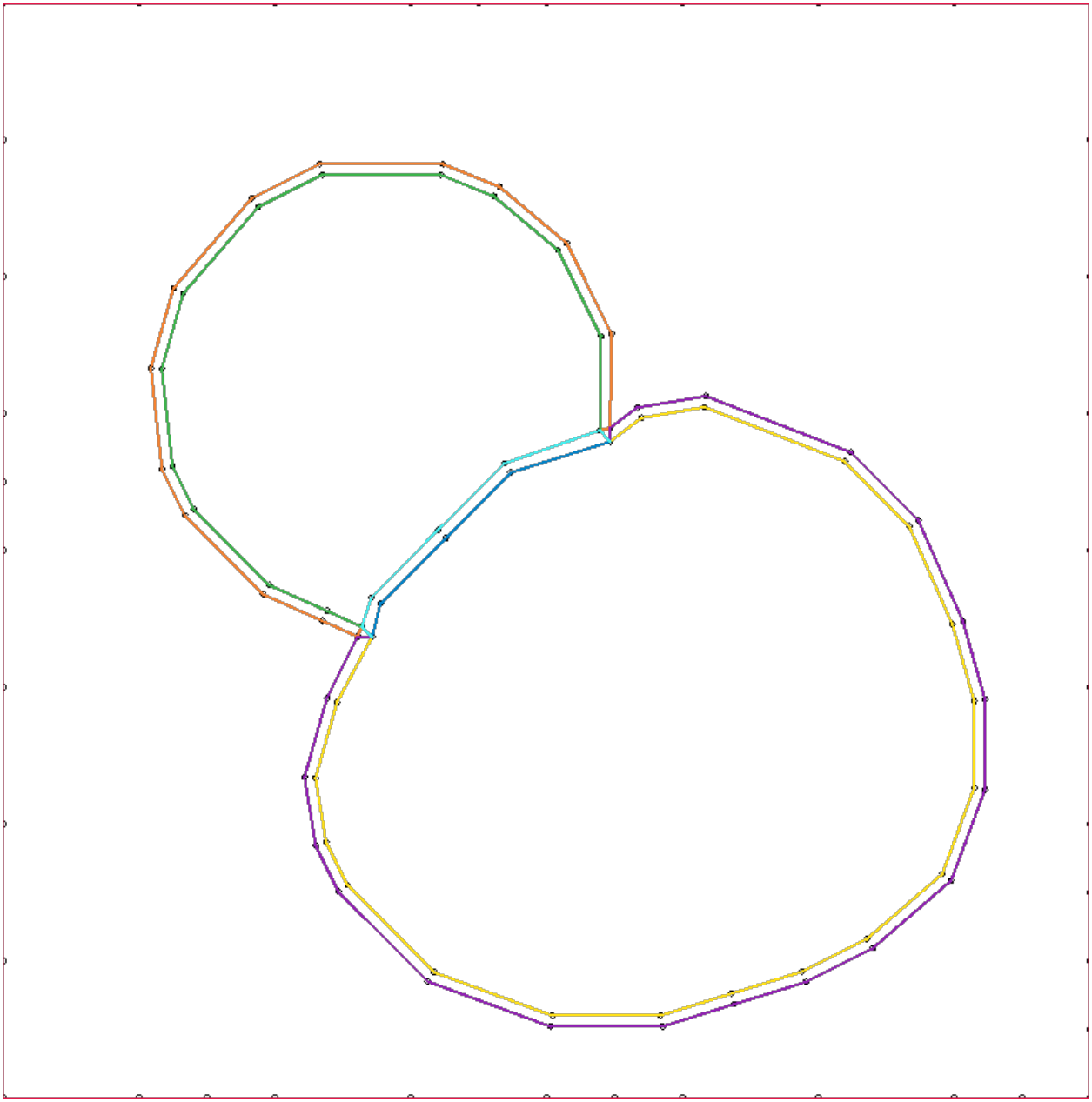

Fig. 9: Simplified boundary lines with membrane compartment boundaries.

9.9 Mesh generation

Finally we are ready to generate the triangular mesh. We do this again using the Triangle library, with the same maximum triangle area constraints as previously, but this time disallowing the creation of new Steiner points, to ensure that the points on the membrane boundary lines which we will use to construct the quadrilateral elements are not altered. We also insert holes in each membrane compartment such that no triangles are constructed inside them, and instead we fill them with rectangular elements after triangulation.

The user can then adjust the maximum allowed triangle area for each compartment, the number of points used to approximate a boundary, and the width of each membrane compartment.

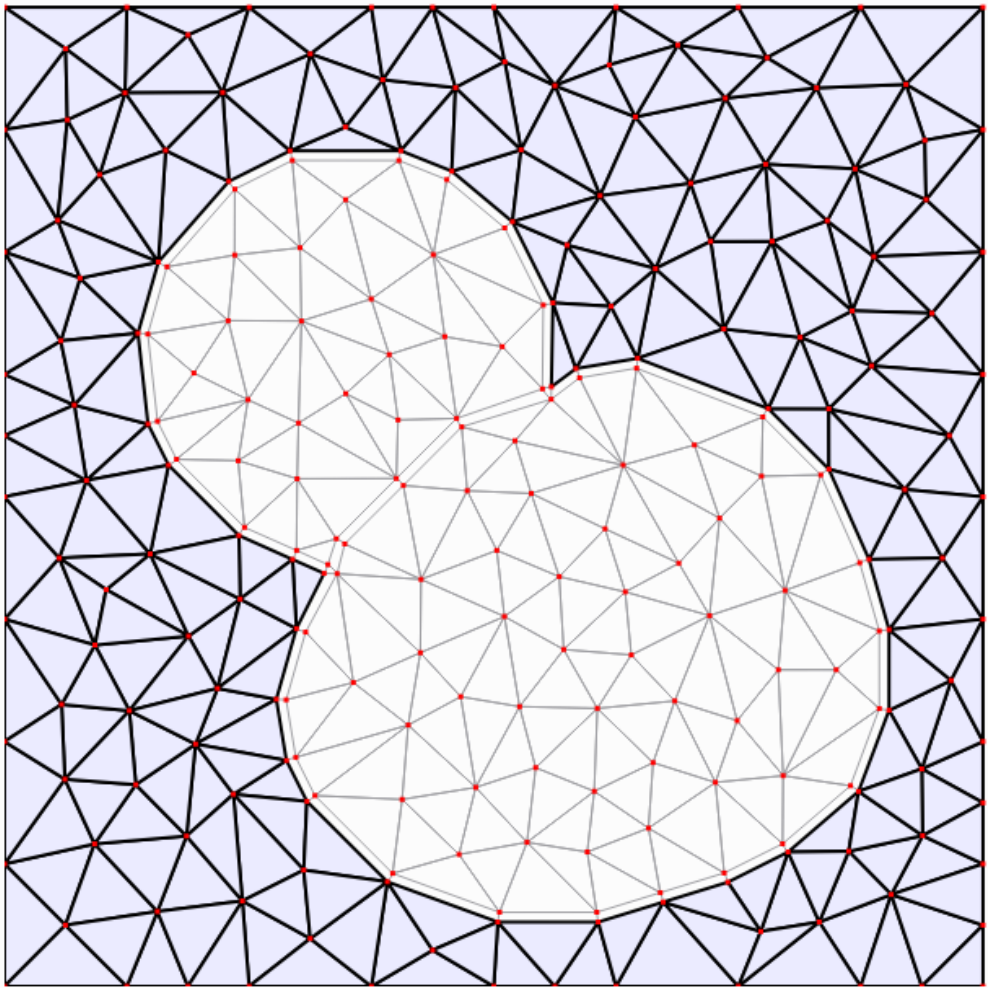

Fig. 10: Generated triangular mesh with rectangular membrane compartments.

CHAPTER 10

Python Interface

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

```
pip install sme
```

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

Once it is installed, you should be able to import the `sme` module and load the built-in example model:

```
  $ python
  Python 3.8.5 (default, Aug 13 2020, 08:28:54)
  [GCC 10.0.1 20200416 (experimental) [master revision 3c3f12e2a76:dcee354ce56:44 on␣
↪linux
  Type "help", "copyright", "credits" or "license" for more information.
  >>> import sme
  >>> model = sme.open_example_model()
  >>> print(model)
  <sme.Model>
    - name: 'Very Simple Model'
    - compartments:
       - Outside
       - Cell
       - Nucleus
```

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

As shown in the getting started notebook, `sme` allows you to import an existing model, change the value of parameters in the model, simulate the model and produce images of the species concentrations from the simulation. To get help on an object, its methods and properties, use the help function, e.g. `help(sme.Model)`.

CHAPTER 11

Command Line Interface

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

-  linux
-  macOS
-  windows

11.1 Use

It can be used to simulate an existing model and save the resulting images:

```
./spatial-cli filename [simulator] [simulation_time] [image_interval] [max_cpu_
↪threads]
```

11.2 Command line parameters

- **`filename`**
  - the SBML file containing the model to be simulated
- **`simulator`**
  - this can be `dune` or `pixel`
  - default: `dune`
- **`simulation_time`**
  - the total length of the simulation (in model units of time)
  - default: `100`

- **`image_interval`**
  - the interval (in model units of time) at which concentration images should be saved
  - default: `10`
- **`max_cpu_threads`**
  - the maximum number of cpu threads that should be used
  - default: `0` (which means unlimitied / use all available threads)
  - note: this parameter only applies to the `pixel` simulator

CHAPTER 12

Maths

12.1 Reaction-Diffusion

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

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

where

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

and we assume that

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

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

12.2 Compartment Reactions

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

12.3 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. They are evaluated on the membrane, and can be implemented as interface conditions between the PDEs in neighbouring compartments, or as additional reaction terms localised on these membranes.

12.4 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 13

Units

13.1 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*.

13.2 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`

CHAPTER 14

Source Code

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

CHAPTER 15

Diffusion

15.1 Model

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

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

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

15.2 Analytic solution

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

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

where

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

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

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

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

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

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

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

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

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