SyMBac

Release 0.2

Georgeos Hardo

Sep 26, 2023

INTRODUCTION

SyMBac (Synthetic Micrographs of Bacteria) is a tool to generate synthetic phase contrast or fluorescence images of bacteria. Currently the tool only supports bacteria growing in the mother machine and on agar pads.

Coming soon:

- Agar pad simulations (working, code being added)
- Command line interface (currently testing)

Read the preprint, SyMBac: Synthetic Micrographs for Accurate Segmentation of Bacterial Cells using Deep Neural Networks, Georgeos Hardo, Maximillian Noka, Somenath Bakshi

---

Note: This project is under active development. Please report issues on GitHub, or if you want specific usage help, please email me.

CHAPTER

ONE

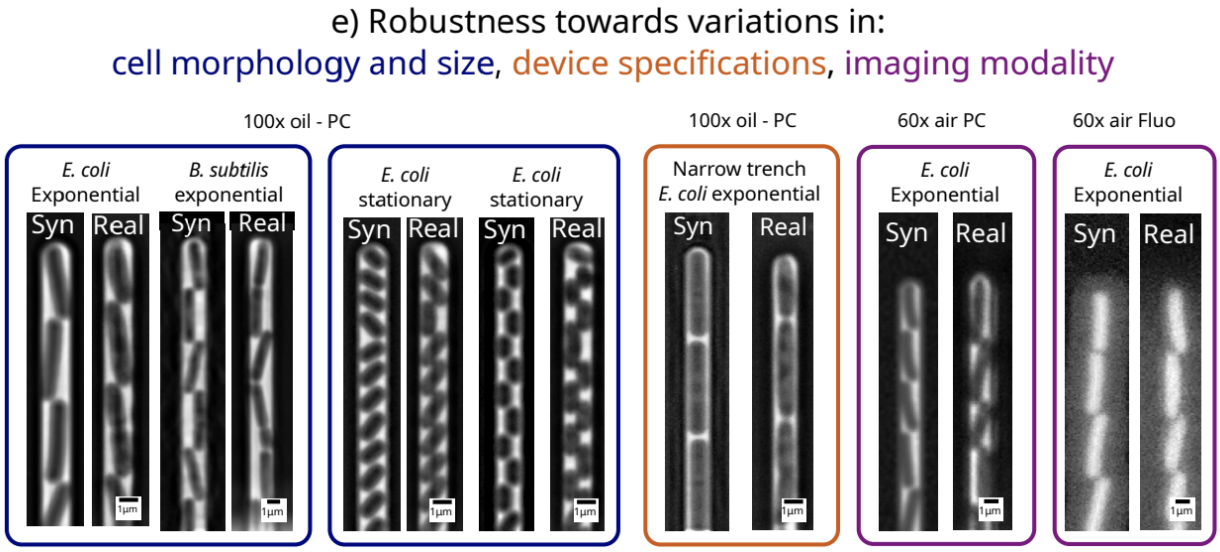
CONTENTS

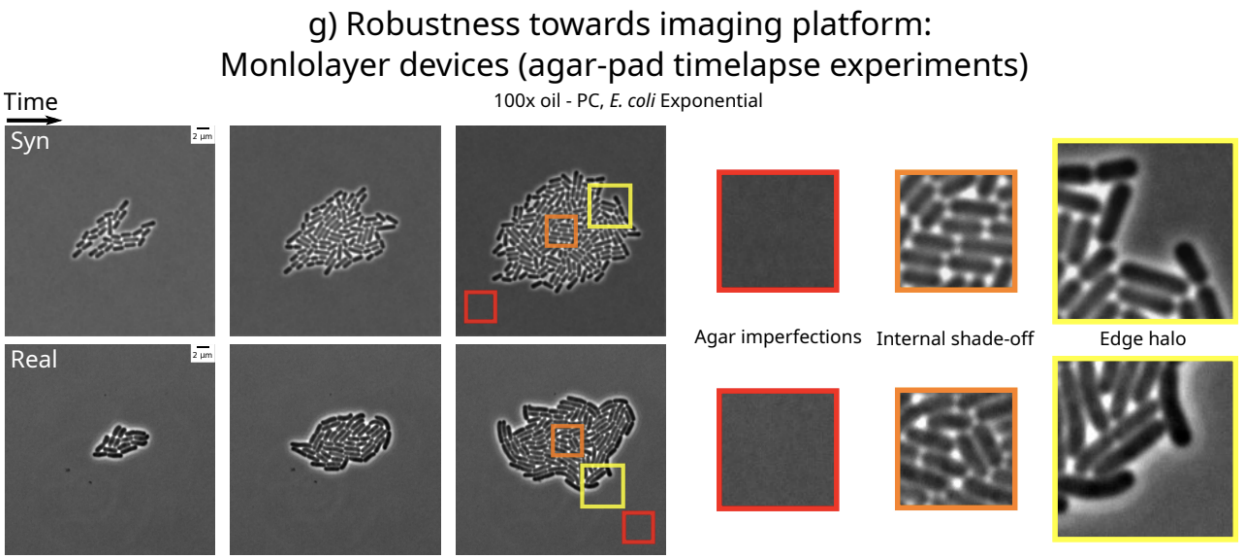
1.1 Introduction

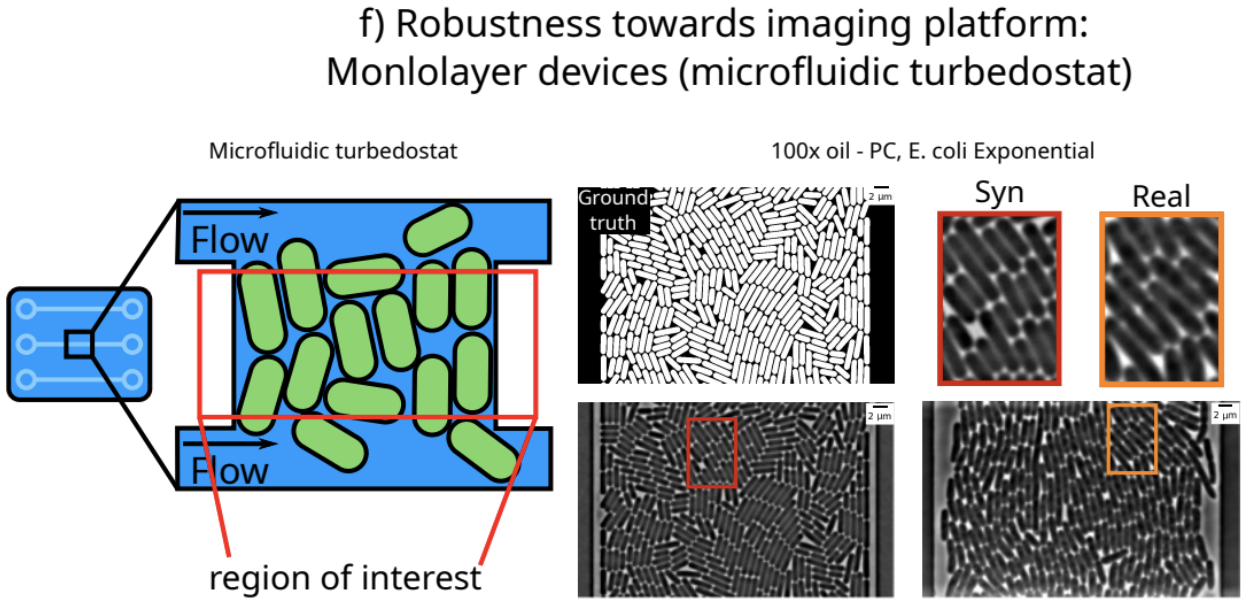
For a quick understanding of what SyMBac is, without reading the preprint, then read this page.

1.1.1 What is it?

SyMBac is a tool to generate synthetic phase contrast or fluorescence images of bacteria. Currently the tool only supports bacteria growing in the mother machine, we also have preliminary support for bacteria growing in 2D monolayers on agar pads (again, both in fluorescence and phase contrast).







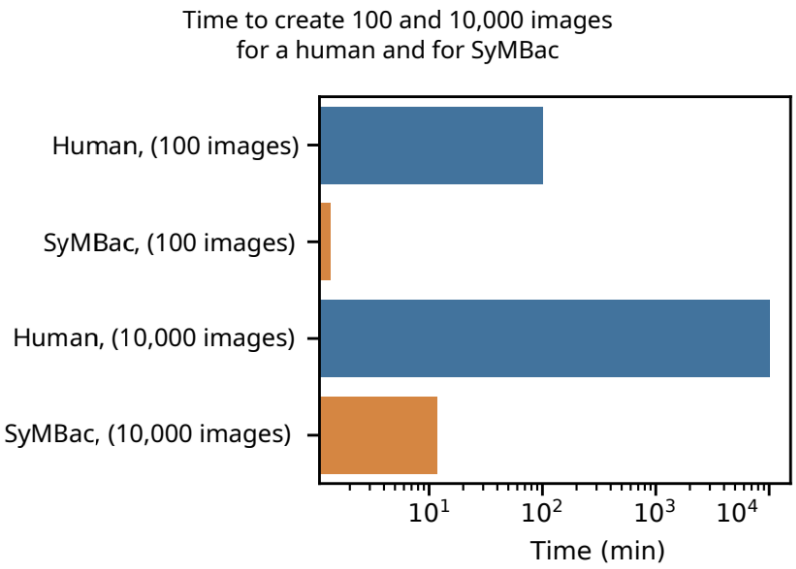
1.1.2 Why would I want to generate synthetic images?

Because you're sick of generating your own training data by hand! Synthetic images provide an instant source of high quality and unlimited training data for machine learning image segmentation algorithms!

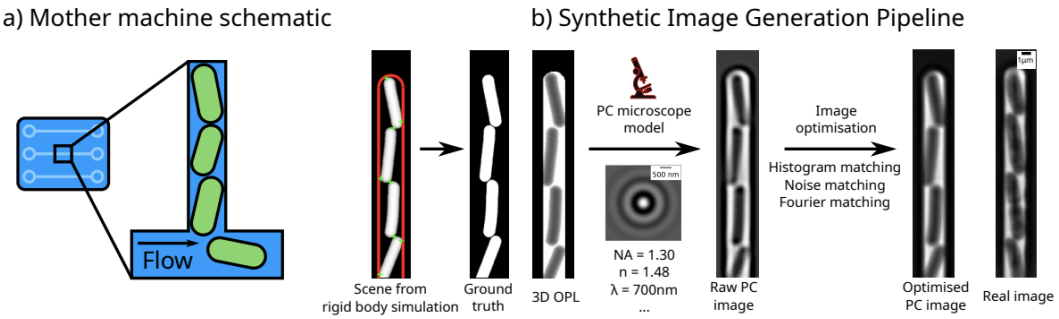
The images are tuned to perfectly replicate your experimental setup, no matter what your microscope's objective is (we have tested 20x air all the way to 100x oil), no matter your imaging modality (phase contrast/fluorescence), and no matter the geometry of your microfluidic device.

Additionally:

- SyMBac is very fast at generating training data compared to humans:

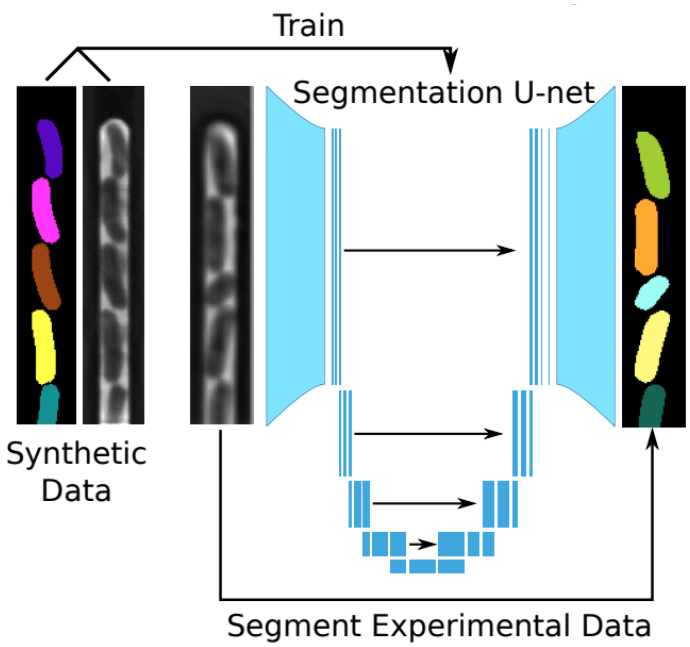


- The image generation process uses a rigid body physics model to simulate bacterial growth, 3D cell geometry to calculate the light's optical path, and a model of the phase contrast/fluorescence optics (point spread function), with some post-rendering optimisation to match image similarity:



1.1.3 How do I use these synthetic images?

That is up to you. SyMBac is not a machine learning tool. It is a tool to generate unlimited free training data which accurately represents your experiment. It is up to you to train a machine learning network on these synthetic images. We do however provide example notebooks for how to train a U-net (as implemented by DeLTA), and for training Omnipose.



1.2 Installation

1.2.1 Prerequisites

Please make sure you have an NVIDIA GPU and a working installation of CUDA and cudNN. If you don't have an NVIDIA GPU then the convolution will default to the CPU, and be very slow.

SyMBac is meant to be run interactively (in a notebook + with a small Qt/GTK interface), so make sure that you are running this on a local machine (you should have access to the machine's display).

If you are running SyMBac on a remote machine, say through an SSH tunnel, you can still use it, but you will need to ensure you have an active VNC screen available, as SyMBac needs access to a screen to render the live simulation. You do not need to be actively accessing the VNC session, it just needs to be running.

1.2.2 Installation

```
pip install SyMBac
```

Or to install the development version (recommended for now), run:

```
pip install git+https://github.com/georgeoshardo/SyMBac
```

Activate the Jupyter widgets extension. This is needed to interact with slides in the notebooks to optimise images.

```
jupyter nbextension enable --py widgetsnbextension
```

If you're using a GPU

Check the version of CUDA you have installed using nvcc –version and install the appropriate version of cupy. For example, if you have CUDA 11.4 you would install as follows:

```
pip install cupy-cuda114
```

If you installed CUDA on Ubuntu 18.04+ using the new Nvidia supplied repositories, it is a real possibility that nvcc won't work. Instead check your CUDA version using nvidia-smi.

If you aren't using a GPU

See FAQs "Do I need to have a GPU?"

1.3 FAQs

- **Do I need to have a GPU?**
  - No, although image synthesis will be around 40x slower on the CPU. SyMBac will detect that you do not have CuPy installed and default to using CPU convolution.
  - Interactive image optimisation will be very painful on the CPU. By default I turn off slider interactivity if you are using the CPU, so that you can move a slider without the CPU being maxed out. This means that every time you move a slider you must click the button to update the image (do a convolution).
- **Can I generate fluorescence images as well?**
  - Yes, you can do fluorescence image generation, just make sure that in the interactive image generation part of the code, you select fluorescence.
  - Since our fluorescence kernel is defined to be a subset of the phase contrast kernel, you can choose **any** condenser, and your fluorescence kernel should be correct. Just ensure that the imaging wavelength, numerical aperture, refractive index, and pixel size are set correctly.
- **What format do my images need to be in?**
  - The real images you are trying to replicate should be in the format of single-trench timeseries images. If you are unsure what this is, you can call `get_sample_images()["E. coli 100x"]` from `SyMBac.misc` for an example image.
- **I'm getting libGL MESA-LOADER/swarst driver errors**
  - Try `conda install -c conda-forge libstdcxx-ng`
  - See this StackExchange link.
- **I'm getting a libGL error**
  - E.g:

```
libGL error: MESA-LOADER: failed to open swrast: /usr/lib/dri/swrast_dri.so: cannot␣
↪open shared object file: No such file or directory (search paths /usr/lib/x86_64-
↪linux-gnu/dri:\$${ORIGIN}/dri:/usr/lib/dri, suffix _dri) libGL error: failed to␣
↪load driver: swrast
```

  - Try running:

```
export LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libstdc++.so.6
```

1.4 Mother machine simulations

[1]:
```
%load_ext autoreload
%autoreload 2
```

[3]:
```
import sys
sys.path.insert(1, '/home/georgeos/Documents/GitHub/SyMBac/') # Not needed if you␣
↪installed SyMBac using pip

from SyMBac.simulation import Simulation
from SyMBac.PSF import PSF_generator
from SyMBac.renderer import Renderer
from SyMBac.PSF import Camera
from SyMBac.misc import get_sample_images
real_image = get_sample_images()["E. coli 100x"]
```

```
/home/georgeos/Documents/GitHub/SyMBac/SyMBac/cell_simulation.py:10:␣
↪TqdmExperimentalWarning: Using `tqdm.autonotebook.tqdm` in notebook mode. Use `tqdm.
↪tqdm` instead to force console mode (e.g. in jupyter console)
  from tqdm.autonotebook import tqdm
TiffPage 0: TypeError: read_bytes() missing 3 required positional arguments: 'dtype',
↪'count', and 'offsetsize'
```

1.4.1 Running a simulation

Mother machine simulations are handled with the `Simulation` class.

Instantiating a `Simulation` object requires the following arguments to be specified:

- *trench_length*: The length of the trench.
- *trench_width*: The width of the trench.
- *cell_max_length*: The maximum allowable length of a cell in the simulation.
- *cell_width*: The average width of cells in the simulation.
- *sim_length*: The number of timesteps to run the simulation for.
- *pix_mic_conv*: The number of microns per pixel in the simulation.
- *gravity*: The strength of the arbitrary gravity force in the simulations.
- *phys_iters*: The number of iterations of the rigid body physics solver to run each timestep. Note that this affects how gravity works in the simulation, as gravity is applied every physics iteration, higher values of *phys_iters* will result in more gravity if it is turned on.
- *max_length_var*: The variance applied to the normal distribution which has mean *cell_max_length*, from which maximum cell lengths are sampled.
- *width_var*: The variance applied to the normal distribution of cell widths which has mean *cell_width*.

- *save_dir*: The save location of the return value of the function. The output will be pickled and saved here, so that the simulation can be reloaded later withuot having to rerun it, for reproducibility. If you don't want to save it, just leave it as /tmp/`.

More details about this class can be found at the API reference: *SyMBac.simulation.Simulation()*

[4]:
```
my_simulation = Simulation(
    trench_length=15,
    trench_width=1.3,
    cell_max_length=6.65, #6, long cells # 1.65 short cells
    cell_width= 1, #1 long cells # 0.95 short cells
    sim_length = 100,
    pix_mic_conv = 0.065,
    gravity=0,
    phys_iters=15,
    max_length_var = 0.,
    width_var = 0.,
    lysis_p = 0.,
    save_dir="/tmp/",
    resize_amount = 3
)
```

We can then run the simulation by calling the *SyMBac.simulation.Simulation.run_simulation()* method on our instantiated object. Setting *show_window=True* will bring up a *pyglet* window, allowing you to watch the simulation in real time. If you run SyMBac headless, then keep this setting set to *False*, you will be able to visualise the simulation in the next step.

[5]:
```
my_simulation.run_simulation(show_window=False)
```

```
  0%|          | 0/100 [00:00<?, ?it/s]
```

Next we call `SyMBac.simulation.Simulation.draw_simulation_OPL()`, which will convert the output of the simulation to a tuple of two arrays, the first being the optical path length (OPL) images, and the second being the masks.

This method takes two arguments.

- *do_transformation* - Whether or not to bend or morph the cells to increase realism.
- *label_masks* - This controls whether the output training masks will be binary or labeled. Binary masks are used to train U-net (e.g DeLTA), wheras labeled masks are used to train Omnipose

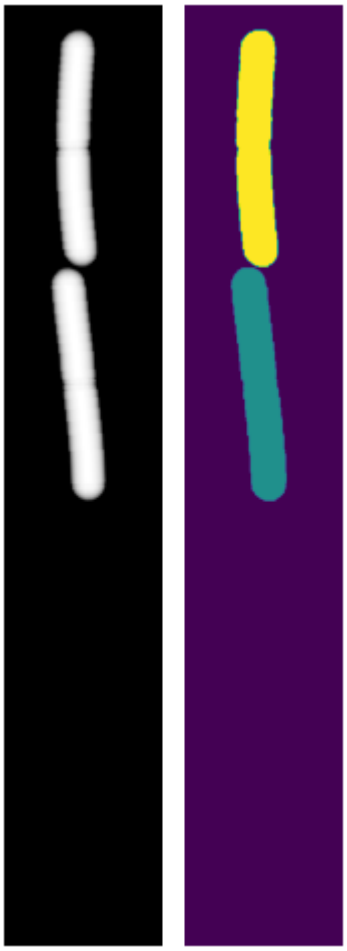
[6]:
```
my_simulation.draw_simulation_OPL(do_transformation=True, label_masks=True)
```

```
Timeseries Properties:   0%|          | 0/98 [00:00<?, ?it/s]
Scene Draw::   0%|          | 0/98 [00:00<?, ?it/s]
```

1.4.2 Simulation visualisation

We can visualise one of the OPL images and masks from the simulation:

[7]:
```
import matplotlib.pyplot as plt
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(2,5))
ax1.imshow(my_simulation.OPL_scenes[-1], cmap="Greys_r")
ax1.axis("off")
ax2.imshow(my_simulation.masks[-1])
ax2.axis("off")
plt.tight_layout()
```



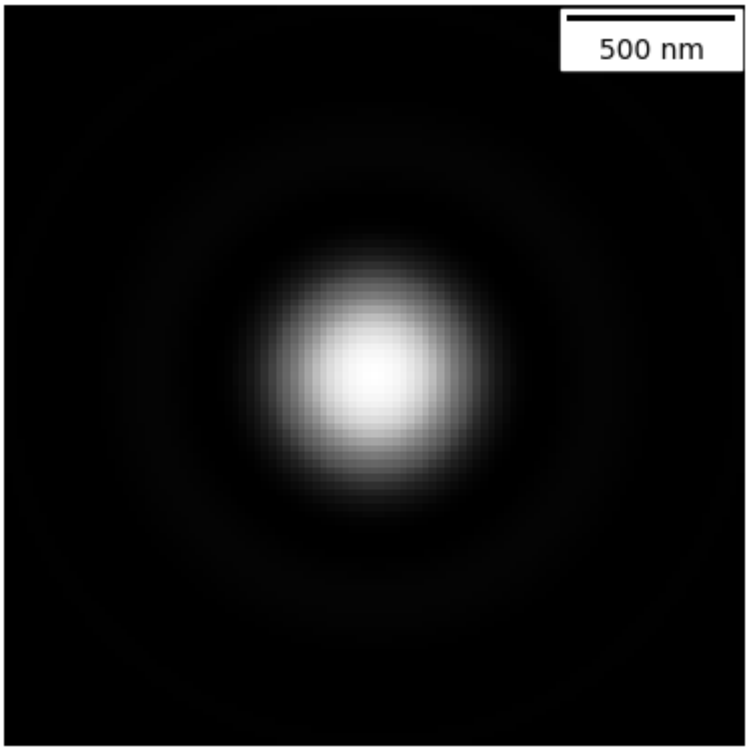
Alternatively we can bring up a napari window to visualise the simulation interactively using *SyMBac.simulation.Simulation.visualise_in_napari()*

[8]:
```
my_simulation.visualise_in_napari()
```

1.4.3 Point spread function (PSF) generation

The next thing we must do is define the optical parameters which define the microscope simulation. To do this we instantiate a *PSF_generator*, which will create our point spread functions for us. We do this by passing in the parameters defined in *SyMBac.PSF.PSF_generator* but below we show 3 examples:

[9]:
```
# A 2D simple fluorescence kernel based on Airy
my_kernel = PSF_generator(
    radius = 50,
    wavelength = 0.75,
    NA = 1.2,
    n = 1.3,
    resize_amount = 3,
    pix_mic_conv = 0.065,
    apo_sigma = None,
    mode="simple fluo")
my_kernel.calculate_PSF()
my_kernel.plot_PSF()
```

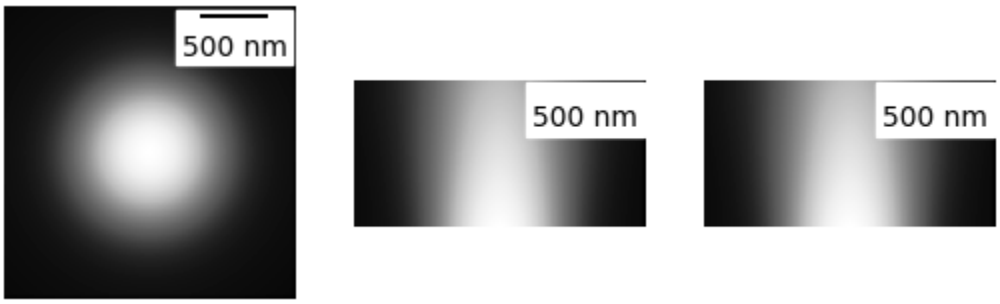


[10]:
```
# A 3D fluorescence kernel based on the psfmodels library
# (note the larger looking PSF due to the summed projection)
# (During convolution, each slice of the image is convolved with the relevent
# volume slice of the cell)
my_kernel = PSF_generator(
    z_height=50,
    radius = 50,
    wavelength = 0.75,
    NA = 1.2,
```

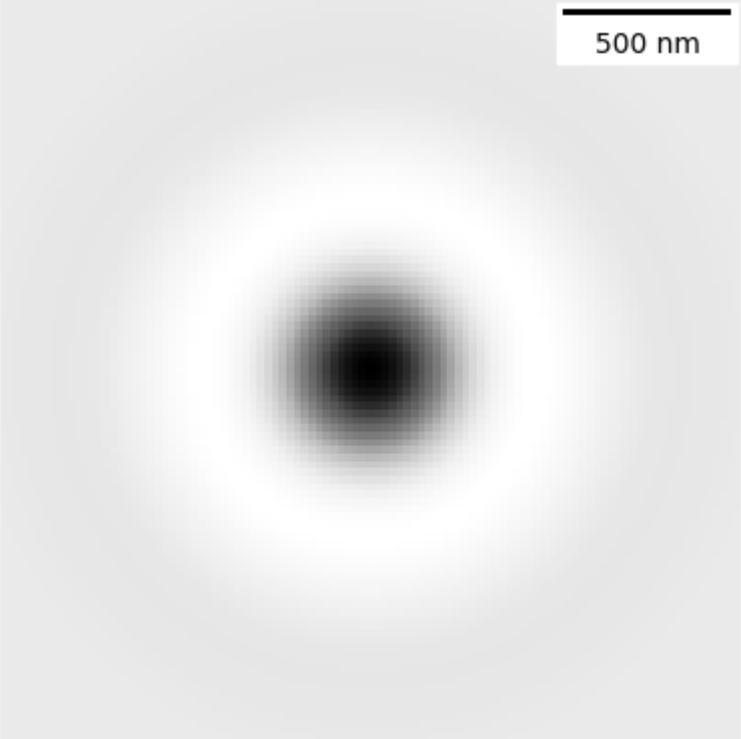
(continues on next page)

(continued from previous page)

```
    n = 1.3,
    resize_amount = 3,
    pix_mic_conv = 0.065,
    apo_sigma = None,
    mode="3D fluo")
my_kernel.calculate_PSF()
my_kernel.plot_PSF()
```



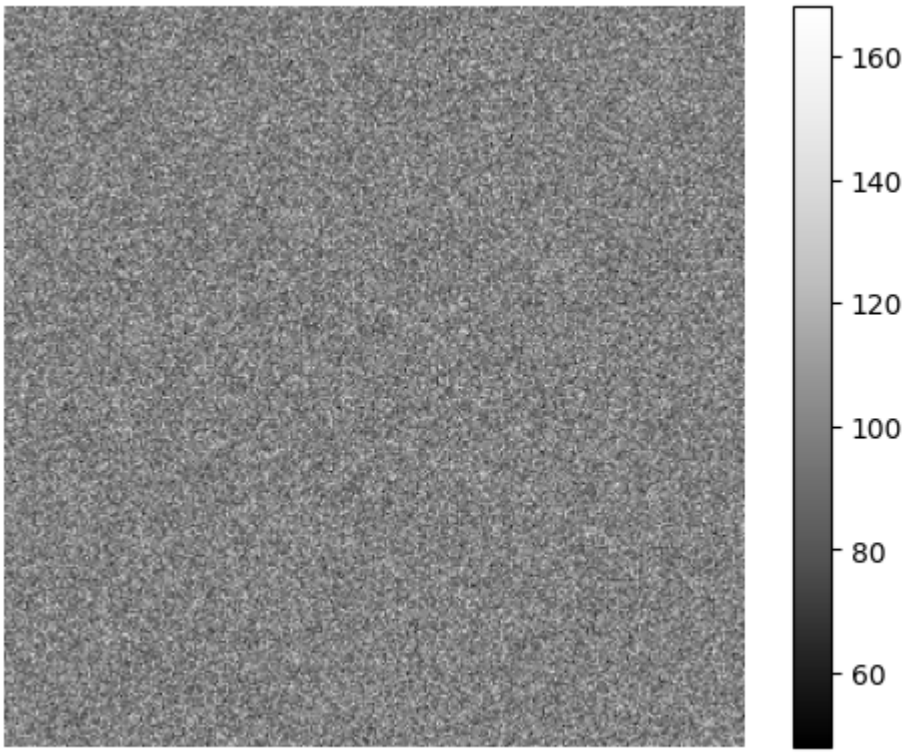
[11]:
```
# A phase contrast kernel
my_kernel = PSF_generator(
    radius = 50,
    wavelength = 0.75,
    NA = 1.2,
    n = 1.3,
    resize_amount = 3,
    pix_mic_conv = 0.065,
    apo_sigma = 20,
    mode="phase contrast",
    condenser = "Ph3")
my_kernel.calculate_PSF()
my_kernel.plot_PSF()
```



1.4.4 Camera model

Next we optionally define a camera based on `SyMBac.camera.Camera()`, if your camera properties are known. If you do not know them, then you will be allowed to do *ad-hoc* noise matching during image optimisation.

[12]:
```
my_camera = Camera(baseline=100, sensitivity=2.9, dark_noise=8)
my_camera.render_dark_image(size=(300,300));
```



1.4.5 Rendering

Next we will create a renderer with *SyMBac.renderer.Renderer()*, this will take our simulation, our PSF (in this case phase contrast), a real image, and optionally our camera.

[13]:
```
my_renderer = Renderer(simulation = my_simulation, PSF = my_kernel, real_image = real_
↪image, camera = my_camera)
```

Next we shall extract some pixels from the real image which we will use to optimise the synthetic image. We will extract the pixel intensities and variances from the 3 important regions of the image. The cells, the device, and the media. These are the same three aforementioned intensities for which we "guessed" some parameters in the previous code block.

We use napari to load the real image, and create three layers above it, called `media_label`, `cell_label`, and `device_label`. We will then select each layer and draw over the relevant regions of the image.

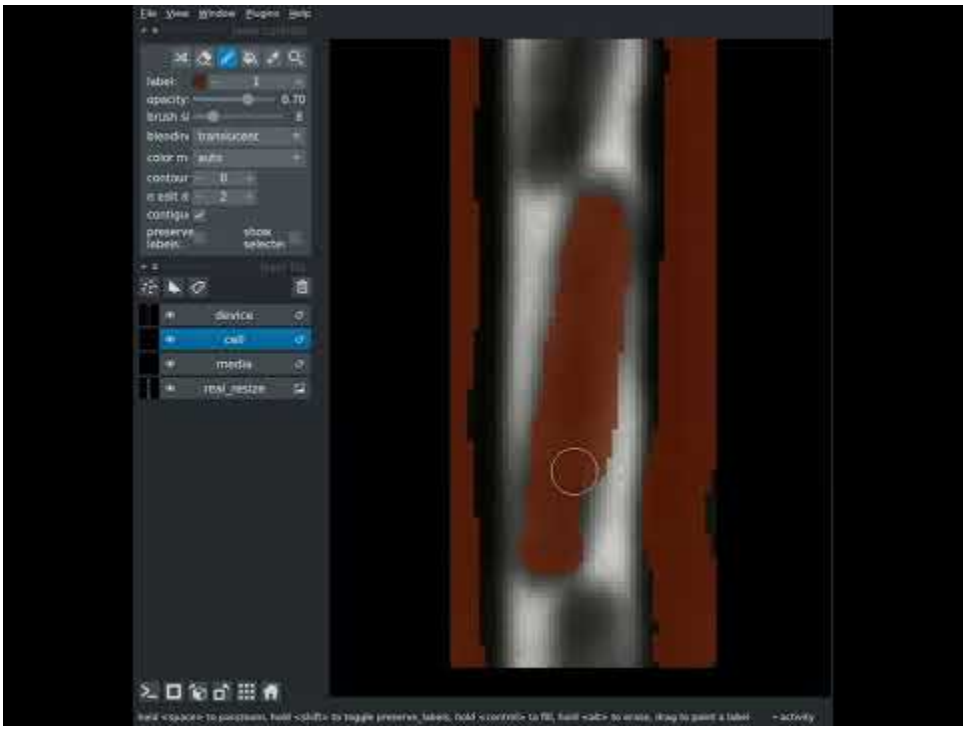
A video below shows how it's done:

Note: You do not need to completely draw over all the cells, the entire device, or all the media gaps between the cells. Simply getting a representative sample of pixels is generally enough. See the video below for a visual demonstration.

---

[14]:
```
from IPython.display import YouTubeVideo
YouTubeVideo("sPC3nV_5DfM")
```

[14]:

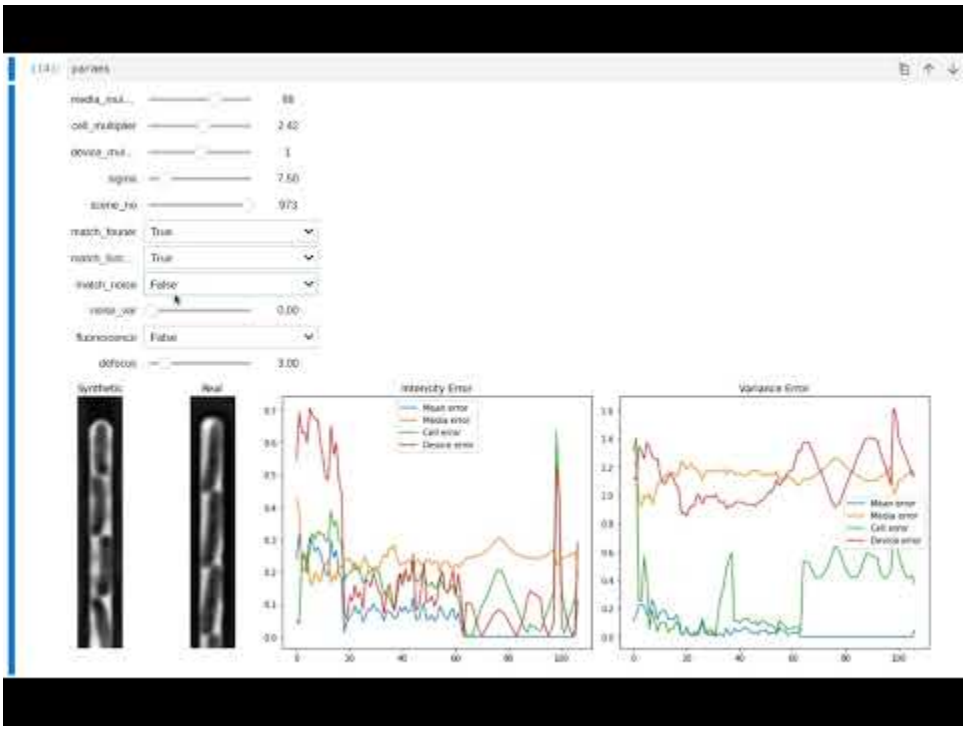


[15]:
```
my_renderer.select_intensity_napari()
```

Finally, we will use the manual optimiser (*SyMBac.renderer.Renderer.optimise_synth_image()*) to generate a realistic image. The output from the optimiser will then be used to generate an entire dataset of synthetic images. Below the code is a video demonstrating the optimisation process.

[16]:
```
YouTubeVideo("PeeyotMQAQU")
```

[16]:



[18]:
```
my_renderer.optimise_synth_image(manual_update=False)
```

```
interactive(children=(IntSlider(value=75, description='media_multiplier', max=300, min=-
↪300), FloatSlider(valu...
```

Finally, we generate our training data using *SyMBac.renderer.Renderer.generate_training_data()*.

The important parameters to recognise are:

- *sample_amount*: This is a percentage by which all continuous parameters `manual_optimise()` can be randomly scaled during the synthesis process. For example, a value of 0.05 will randomly scale all continuous parameters by $X \sim U(0.95, 1.05)$ Higher values will generate more variety in the training data (but too high values may result in unrealistic images).
- *randomise_hist_match*: Whether to randomise the switching on and off of histogram matching.
- *randomise_noise_match*: Whether to randomise the switching on and off of noise matching.
- *sim_length*: The length of the simulation
- *burn_in*: The number of frames at the beginning of the simulation to ignore. Useful if you do not want images of single cells to appear in your training data.
- *n_samples*: The number of random training samples to generate.
- *save_dir*: The save directory of the images. Will output two folders, `convolutions` and `masks`.
- *in_series*: Whether or not to shuffle the simulation while generating training samples.

Note: When running this, you may choose to set `in_series=True`. This will generate training data whereby each image is taken sequentially from the simulation. This useful if you want train a tracking model, where you need the frames to be in order. If you choose to set `in_series=True`, then it is a good idea to choose a low value of `sample_amount`, typically less than 0.05 is sensible. This reduces the frame-to-frame variability.

---

[19]:
```
my_renderer.generate_training_data(sample_amount=0.1, randomise_hist_match=True,␣
↪randomise_noise_match=True, burn_in=40, n_samples = 500, save_dir="/tmp/test/", in_
↪series=False)
```

```
Sample generation: 100%|| 500/500 [00:50<00:00,  9.81it/s]
```

[ ]:
```
```

1.5 Agar pad simulations

1.5.1 This notebook is currently being updated, and is unfinished!

[1]:
```
%load_ext autoreload
```

[2]:
```
%autoreload 2
```

[1]:
```
import numpy as np
import os
import pickle
from skimage.transform import rescale, rotate
import noise
import matplotlib.pyplot as plt
```

[2]:
```
import sys
sys.path.insert(1, '/home/georgeos/Documents/GitHub/SyMBac/') # Not needed if you␣
↪installed SyMBac using pip
from SyMBac.drawing import raster_cell
from SyMBac.colony_simulation import ColonySimulation
```

```
/home/georgeos/Documents/GitHub/SyMBac/SyMBac/colony_simulation.py:8:␣
↪TqdmExperimentalWarning: Using `tqdm.autonotebook.tqdm` in notebook mode. Use `tqdm.
↪tqdm` instead to force console mode (e.g. in jupyter console)
  from tqdm.autonotebook import tqdm
```

For microcolony simulations, we prefer to use CellModeller[1] instead of the inbuilt SyMBac simulator. We will be updating the following page with example models written for CellModeller, which cover simple microcolonies, and other microfluidic geometries: Example models.

[3]:
```
colonysim = ColonySimulation(
    cellmodeller_model= 'cellmodeller_ex1_simpleGrowth_modified.py',
    max_cells = 100,
    pix_mic_conv = 0.065,
    resize_amount = 3,
    save_dir = "test/",
)
```

[6]:
```
colonysim.run_cellmodeller_sim(num_sim=1)
```

[1] Computational Modeling of Synthetic Microbial Biofilms Timothy J. Rudge, Paul J. Steiner, Andrew Phillips, and Jim Haseloff ACS Synthetic Biology 2012 1 (8), 345-352 DOI: 10.1021/sb300031n

```
Set up OpenCL context:
  Platform: NVIDIA CUDA
  Device: Quadro RTX 3000
Importing model cellmodeller_ex1_simpleGrowth_modified
      10           2 cells           0 contacts    0.000034 hour(s) or 0.002035␣
↪minute(s) or 0.122123 second(s)
      20           2 cells           0 contacts    0.000041 hour(s) or 0.002452␣
↪minute(s) or 0.147110 second(s)
      30           2 cells           0 contacts    0.000048 hour(s) or 0.002906␣
↪minute(s) or 0.174348 second(s)
      40           2 cells           0 contacts    0.000056 hour(s) or 0.003365␣
↪minute(s) or 0.201879 second(s)
      50           2 cells           0 contacts    0.000063 hour(s) or 0.003795␣
↪minute(s) or 0.227723 second(s)
      60           3 cells           1 contacts    0.000075 hour(s) or 0.004480␣
↪minute(s) or 0.268788 second(s)
   60     3 cells       2 cts       3 iterations  residual = 0.000513
      70           5 cells           4 contacts    0.000092 hour(s) or 0.005502␣
↪minute(s) or 0.330109 second(s)
   70     5 cells       4 cts       4 iterations  residual = 0.001992
      80           5 cells           4 contacts    0.000110 hour(s) or 0.006609␣
↪minute(s) or 0.396537 second(s)
   80     5 cells       4 cts       3 iterations  residual = 0.003237
      90           5 cells           5 contacts    0.000128 hour(s) or 0.007680␣
↪minute(s) or 0.460783 second(s)
   90     5 cells       4 cts       2 iterations  residual = 0.003866
     100           8 cells           7 contacts    0.000146 hour(s) or 0.008754␣
↪minute(s) or 0.525262 second(s)
  100     8 cells       7 cts       4 iterations  residual = 0.002115
     110           9 cells           8 contacts    0.000164 hour(s) or 0.009836␣
↪minute(s) or 0.590187 second(s)
  110     9 cells      10 cts       5 iterations  residual = 0.003253
     120          10 cells          14 contacts    0.000188 hour(s) or 0.011260␣
↪minute(s) or 0.675616 second(s)
  120    10 cells      14 cts       7 iterations  residual = 0.003114
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Input In [6], in <cell line: 1>()
----> 1 colonysim.run_cellmodeller_sim(num_sim=1)

File ~/Documents/GitHub/SyMBac/SyMBac/colony_simulation.py:56, in ColonySimulation.run_
↪cellmodeller_sim(self, num_sim)
     54 # Run the simulation to ~n cells
     55 while len(self.simulation.cellStates) < self.max_cells:
---> 56     self.simulation.step()
     57 self.n_simulations += 1

File ~/miniconda3/envs/symbac/lib/python3.9/site-packages/CellModeller/Simulator.py:382,␣
↪in Simulator.step(self)
    379             self.kill(state)
    381 self.phys.set_cells()
--> 382 while not self.phys.step(self.dt): #neighbours are current here
```

(continues on next page)

(continued from previous page)

```
    383     pass
    384 if self.sig:

File ~/miniconda3/envs/symbac/lib/python3.9/site-packages/CellModeller/Biophysics/
↪BacterialModels/CLBacterium.py:590, in CLBacterium.step(self, dt)
    588 if not self.progress_initialised:
    589     self.progress_init(dt)
--> 590 if self.progress():
    591     self.progress_finalise()
    592     return True

File ~/miniconda3/envs/symbac/lib/python3.9/site-packages/CellModeller/Biophysics/
↪BacterialModels/CLBacterium.py:557, in CLBacterium.progress(self)
    555 def progress(self):
    556     if self.n_ticks:
--> 557         if self.tick(self.actual_dt):
    558             self.n_ticks -= 1
    559         return False

File ~/miniconda3/envs/symbac/lib/python3.9/site-packages/CellModeller/Biophysics/
↪BacterialModels/CLBacterium.py:623, in CLBacterium.tick(self, dt)
    621 if not self.sub_tick_initialised:
    622     self.sub_tick_init(dt)
--> 623 if self.sub_tick(dt):
    624     self.sub_tick_finalise()
    625     return True

File ~/miniconda3/envs/symbac/lib/python3.9/site-packages/CellModeller/Biophysics/
↪BacterialModels/CLBacterium.py:643, in CLBacterium.sub_tick(self, dt)
    641 self.build_matrix() # Calculate entries of the matrix
    642 #print "max cell contacts = %i"%cl_array.max(self.cell_n_cts_dev).get()
--> 643 self.CGSSolve(dt, alpha) # invert MTMx to find deltap
    644 self.add_impulse()
    645 return False

File ~/miniconda3/envs/symbac/lib/python3.9/site-packages/CellModeller/Biophysics/
↪BacterialModels/CLBacterium.py:1030, in CLBacterium.CGSSolve(self, dt, alpha, substep)
   1026 max_iters = self.n_cells*7
   1028 for iter in range(max_iters):
   1029     # Ap
-> 1030     ␣
↪self.calculate_Ax(self.Ap_dev[0:self.n_cells], self.p_dev[0:self.n_cells], dt, alpha)
   1032     # p^TAp
   1033     pAp = self.vdot(self.p_dev[0:self.n_cells], self.Ap_dev[0:self.n_cells]).
↪get()

File ~/miniconda3/envs/symbac/lib/python3.9/site-packages/CellModeller/Biophysics/
↪BacterialModels/CLBacterium.py:951, in CLBacterium.calculate_Ax(self, Ax, x, dt, alpha)
    939 def calculate_Ax(self, Ax, x, dt, alpha):
    941     self.program.calculate_Bx(self.queue,
    942                               (self.n_cells, self.max_contacts),
    943                               None,
```

(continues on next page)

(continued from previous page)

```
   (...)
    949                                 x.data,
    950                                 self.Mx_dev.data).wait()
--> 951     self.program.calculate_BTBx(self.queue,
    952                                 (self.n_cells,),
    953                                 None,
    954                                 numpy.int32(self.max_contacts),
    955                                 self.cell_n_cts_dev.data,
    956                                 self.n_cell_tos_dev.data,
    957                                 self.cell_tos_dev.data,
    958                                 self.fr_ents_dev.data,
    959                                 self.to_ents_dev.data,
    960                                 self.Mx_dev.data,
    961                                 Ax.data).wait()
    962     # Tikhonov test
    963     #self.vaddkx(Ax, numpy.float32(0.01), Ax, x)
    964
    965     # Energy mimizing regularization
    966     self.program.calculate_Mx(self.queue,
    967                                 (self.n_cells,),
    968                                 None,
   (...)
    974                                 x.data,
    975                                 self.Mx_dev.data).wait()

File ~/miniconda3/envs/symbac/lib/python3.9/site-packages/pyopencl/__init__.py:901, in _
↪add_functionality.<locals>.kernel_call(self, queue, global_size, local_size, *args,␣
↪**kwargs)
    895 def kernel_call(self, queue, global_size, local_size, *args, **kwargs):
    896     # __call__ can't be overridden directly, so we need this
    897     # trampoline hack.
    898
    899     # Note: This is only used for the generic __call__, before
    900     # kernel_set_scalar_arg_dtypes is called.
--> 901     return self._enqueue(self, queue, global_size, local_size, *args, **kwargs)

File <pyopencl invoker for 'calculate_BTBx'>:8, in enqueue_knl_calculate_BTBx(self,␣
↪queue, global_size, local_size, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, global_
↪offset, g_times_l, allow_empty_ndrange, wait_for)

KeyboardInterrupt:
```

[ ]:
```
colonysim.get_simulation_dirs()
```

[ ]:
```
pickles = colonysim.get_simulation_pickles()
```

[ ]:
```
colonysim.get_max_scene_size()
```

[ ]:
```
colonysim.draw_simulation_OPL(n_jobs = -1, FL=True, density = 0.1, random_distribution =
↪"uniform", distribution_args = (0.9, 3))
```

[ ]:
```
from SyMBac.colony_renderer import ColonyRenderer
```

[ ]:
```
from SyMBac.PSF import PSF_generator
from SyMBac.renderer import convolve_rescale
from skimage.util import random_noise
from scipy.ndimage import gaussian_filter

from skimage.exposure import rescale_intensity
```

[ ]:
```
my_kernel = PSF_generator(
    radius = 50,
    wavelength = 0.75,
    NA = 1.45,
    n = 1.4,
    resize_amount = 3,
    pix_mic_conv = 0.065,
    apo_sigma = 8,
    mode="simple fluo",
    condenser = "Ph3",
    offset = 0.02
)
my_kernel.calculate_PSF()
my_kernel.plot_PSF()
```

[ ]:
```
my_renderer = ColonyRenderer(colonysim, my_kernel)
```

[ ]:
```
test_img = my_renderer.render_scene(-1)
```

[ ]:
```
mask = my_renderer.mask_loader(-1)
```

[ ]:
```
plt.imshow(test_img, cmap="Greys_r")
```

[ ]:
```
plt.imshow(np.roll(test_img, [0,], axis=[1]), cmap="Greys_r")
```

[ ]:
```
my_renderer.generate_random_samples(1000, 0.2, "training_data")
```

[ ]:
```
import random
random.choice([(0, mask.shape[0]), (1, mask.shape[1]), ([0,1], mask.shape)])
```

[ ]:
```
```

1.6 Convert SyMBac data to be compatible with Omnipose

[28]:
```
from glob import glob
import os
import random
from PIL import Image
from skimage.morphology import label
import numpy as np
import matplotlib.pyplot as plt
import tifffile
from skimage.util import img_as_ubyte
np.random.seed(4)
```

Load in the directories for the masks and convolutions (synthetic images)

[42]:
```
SyMBac_training_data_dir = "training_data_full"
savedir = "omnipose_SyMBac_TD"
```

[12]:
```
masks_dir = SyMBac_training_data_dir + "/masks/"
convs_dir = SyMBac_training_data_dir + "/convolutions/"
masks = sorted(glob(masks_dir+"/*"))
convs = sorted(glob(convs_dir+"/*"))
```

Grab the shape of one of the images

[14]:
```
img_shape = tifffile.imread(masks[0]).shape
print(img_shape)
```

```
(256, 46)
```

Omnipose trains well and more efficiently on tiles of images. We will generate training_data of size `tile_length` for this purpose. You will have generated many training samples using SyMBac, maybe >1000. Omnipose trains well on small datasets, and is also slow to train compared to DeLTA for the same number of images. Therefore we will sample (without replacement) indices from the existing training data, and concatengate adjacent frames.

[41]:
```
tile_length = 40
training_samples = 200
indices = random.sample(range(len(masks)-tile_length), training_samples)
label_required = False # If you did not use label=True in SyMBac, then you should set␣
↪this to true.
```

Generate the training data

[39]:
```
os.mkdir(savedir)
for i, x in enumerate(indices):
    x = indices[i]
    mask_tile = np.concatenate([tifffile.imread(mask) for mask in masks[x:x+tile_
↪length]], axis=1)
    if label_required:
        mask_tile = label(mask_tile)
    conv_tile = np.concatenate([tifffile.imread(conv) for conv in convs[x:x+tile_
↪length]], axis=1)
    conv_tile = conv_tile/conv_tile.max()
```

(continues on next page)

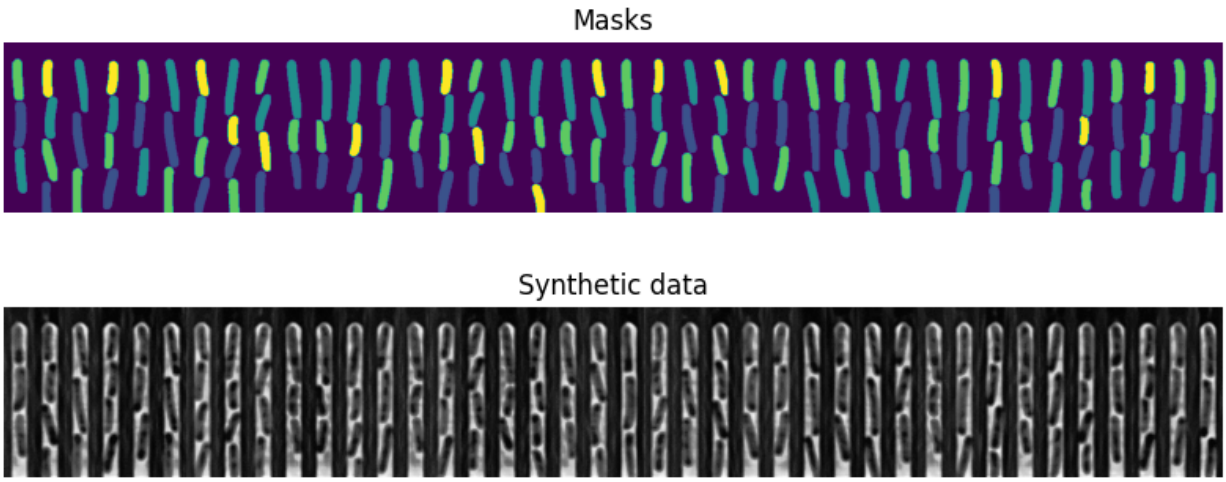
(continued from previous page)

```
    conv_tile = img_as_ubyte(conv_tile)
    Image.fromarray(mask_tile.astype(np.ubyte)).save(f"{savedir}/train_{str(i).zfill(5)}_
↪masks.png")
    Image.fromarray(conv_tile).save(f"{savedir}/train_{str(i).zfill(5)}.png")
```

Visualise the last two training samples

[53]:
```
fig, axs = plt.subplots(2, figsize=(10,4))
axs[0].imshow(mask_tile)
axs[1].imshow(conv_tile, cmap="Greys_r")
for ax in axs:
    ax.axis("off")
axs[0].set_title("Masks")
axs[1].set_title("Synthetic data")
```

[53]:
```
Text(0.5, 1.0, 'Synthetic data')
```



1.7 Training Omnipose

You can then train omnipose by running the following command, which is the same one as given in the Omnipose GitHub repository.

```
python -m omnipose --train --use_gpu --dir SyMBac_training_data_dir --mask_filter "_masks
↪" --n_epochs 4000 --pretrained_model None --save_every 100 --save_each --learning_rate␣
↪0.1 --diameter 0 --batch_size 16
```

This will train for 4000 epochs and save a model every 100 epochs.

1.8 Segmenting mother machine data with Omnipose

This notebook is adapted from the Omnipose docs' basic tutorial which can be found here: https://omnipose.readthedocs.io/examples/mono_channel_bact.html

Here we simply load a single TIFF stack of single mother machine trenches, and tile the images for more efficient segmentation.

[1]:
```
import numpy as np
from cellpose import models, core
from cellpose import plot
import omnipose
from cellpose import models
from glob import glob
from natsort import natsorted

# This checks to see if you have set up your GPU properly.
# CPU performance is a lot slower, but not a problem if you
# are only processing a few images.
use_GPU = core.use_gpu()
print('>>> GPU activated? %d'%use_GPU)

import tifffile
from skimage.transform import rescale, resize, downscale_local_mean

# for plotting
import matplotlib as mpl
import matplotlib.pyplot as plt
```

```
2022-08-25 20:43:08,194 [INFO] ** TORCH CUDA version installed and working. **
>>> GPU activated? 1
```

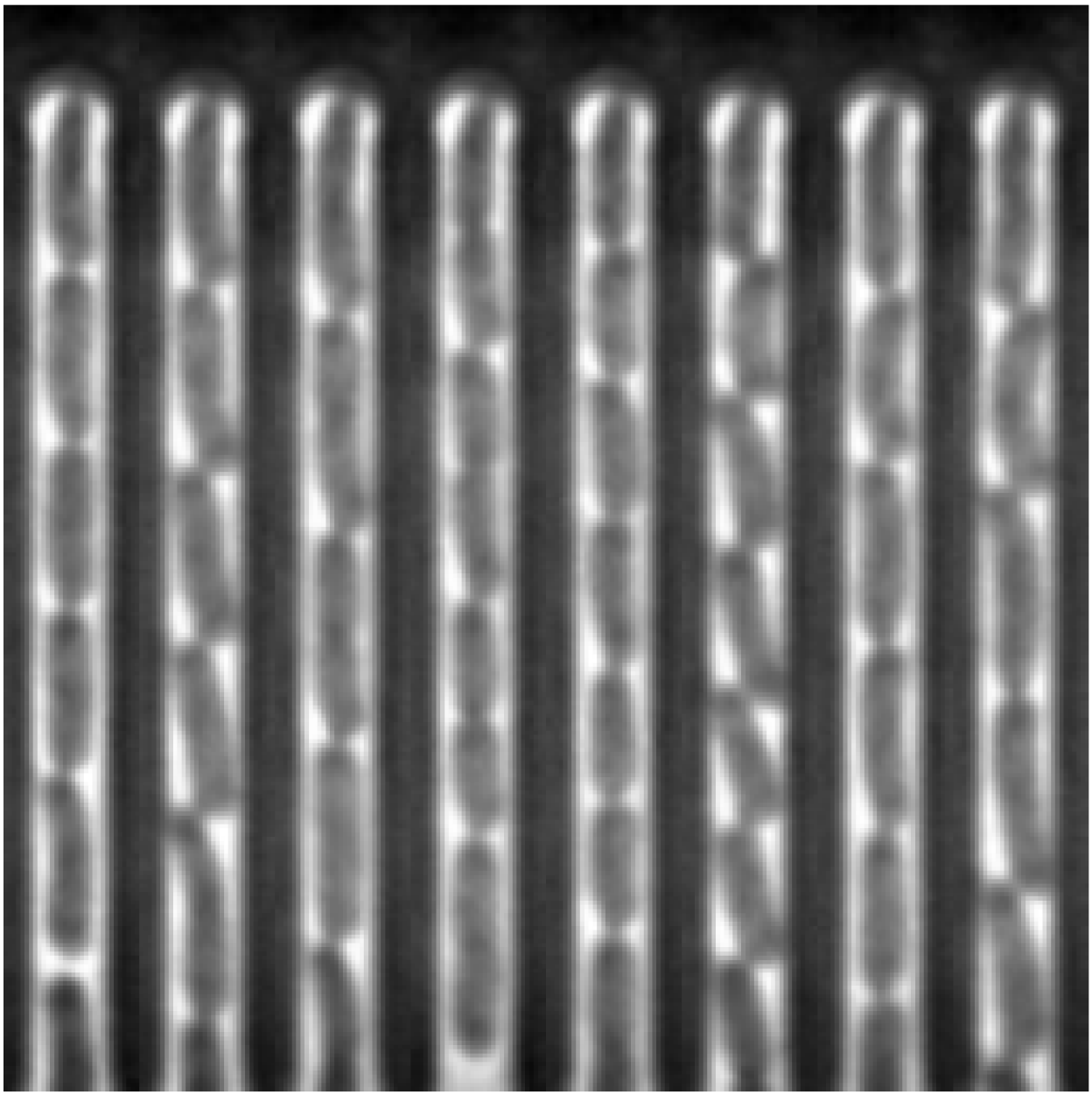
Load in the first 8 frames of the timeseries data

[2]:
```
real_data = tifffile.imread("real_data/trench_0.tif")[:8]
real_data = np.concatenate([data for data in real_data], axis=1)
```

Use omnipose's preprocessing step.

[3]:
```
from cellpose import io, transforms
from omnipose.utils import normalize99
imgs = [real_data]
nimg = len(imgs)
fig = plt.figure(figsize=[40]*2) # initialize figure
for k in range(len(imgs)):
    img = transforms.move_min_dim(imgs[k]) # move the channel dimension last
    if len(img.shape)>2:
        imgs[k] = np.mean(img,axis=-1) # or just turn into grayscale

    imgs[k] = normalize99(imgs[k])
    plt.subplot(1,len(imgs),k+1)
    plt.imshow(imgs[k],cmap='gray')
    plt.axis('off')
```



Load in the last model in the model directory in the training data directory.

[4]:
```
model_list = natsorted(glob("omnipose_training_data_large/models/*"))
model_name = model_list[-1]
print(model_name)
use_gpu = use_GPU# = False
model = models.CellposeModel(gpu=use_gpu, pretrained_model=model_name, omni=True,␣
↪concatenation=True)
```

```
omnipose_training_data_large/models/cellpose_residual_on_style_on_concatenation_off_omni_
↪omnipose_training_data_large_2022_08_17_19_22_08.661299_epoch_3999
```

(continues on next page)

(continued from previous page)

```
2022-08-25 20:43:08,897 [INFO] ** TORCH CUDA version installed and working. **
2022-08-25 20:43:08,897 [INFO] >>>> using GPU
```

Segment the image

[5]:
```
chans = [0,0] #this means segment based on first channel, no second channel

n = [0] # make a list of integers to select which images you want to segment
n = range(nimg) # or just segment them all

# define parameters
mask_threshold = -1
verbose = 0 # turn on if you want to see more output
use_gpu = use_GPU #defined above
transparency = True # transparency in flow output
rescale=None # give this a number if you need to upscale or downscale your images
omni = True # we can turn off Omnipose mask reconstruction, not advised
flow_threshold = 0. # default is .4, but only needed if there are spurious masks to␣
↪clean up; slows down output
resample = True #whether or not to run dynamics on rescaled grid or original grid
masks, flows, styles = model.eval([imgs[i] for i in n],channels=chans,rescale=rescale,
↪mask_threshold=mask_threshold,transparency=transparency,
                                  flow_threshold=flow_threshold,omni=omni,
↪resample=resample,verbose=verbose)
```

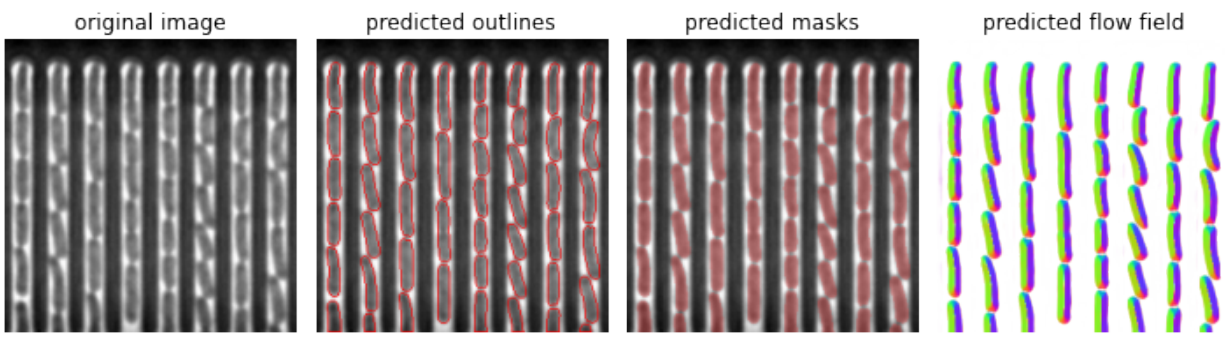
Visualise the segmentation with omnipose's visualisation.

[6]:
```
for idx,i in enumerate(n):

    maski = masks[idx]
    flowi = flows[idx][0]

    fig = plt.figure(figsize=(10,12))
    plot.show_segmentation(fig, omnipose.utils.normalize99(imgs[i]), maski, flowi,␣
↪channels=chans, omni=True, bg_color=0)

    plt.tight_layout()
    plt.savefig(f"omnipose_diagnostics/{str(idx).zfill(4)}.png", dpi = 120)
    plt.show()
```



1.9 SyMBac.cell

class SyMBac.cell.**Cell**(*length*, *width*, *resolution*, *position*, *angle*, *space*, *dt*, *growth_rate_constant*, *max_length*, *max_length_mean*, *max_length_var*, *width_var*, *width_mean*, *parent=None*, *daughter=None*, *lysis_p=0*, *pinching_sep=0*)

Cells are the agents in the simulation. This class allows for instantiating *Cell* object.

Note: Typically the user will not need to call this class, as it will be handled by *SyMBac.cell_simulation()*, specifically all cell setup happens when instantiating a simulation using *SyMBac.simulation.Simulation()*

__init__(*length*, *width*, *resolution*, *position*, *angle*, *space*, *dt*, *growth_rate_constant*, *max_length*, *max_length_mean*, *max_length_var*, *width_var*, *width_mean*, *parent=None*, *daughter=None*, *lysis_p=0*, *pinching_sep=0*)

Initialising a cell

For info about the Pymunk objects, see the API reference. http://www.pymunk.org/en/latest/pymunk.html Cell class has been tested and works with pymunk version 6.0.0

Parameters

- **length** (*float*) – Cell's length
- **width** (*float*) – Cell's width
- **resolution** (*int*) – Number of points defining cell's geometry
- **position** ((*float*, *float*)) – x,y coords of cell centroid
- **angle** (*float*) – rotation in radians of cell (counterclockwise)
- **space** (*pymunk.space.Space*) – The pymunk space of the cell
- **dt** (*float*) – Timestep the cell experiences every iteration
- **growth_rate_constant** (*float*) – The cell grows by a function of dt*growth_rate_constant depending on its growth model
- **max_length** (*float*) – The maximum length a cell reaches before dividing
- **max_length_mean** (*float*) – should be the same as max_length for reasons unless doing advanced simulations
- **max_length_var** (*float*) – The variance defining a normal distribution around max_length
- **width_var** (*float*) – The variance defining a normal distribution around width
- **width_mean** (*float*) – For reasons should be set equal to width unless using advanced features
- **body** (*pymunk.body.Body*) – The cell's pymunk body object
- **shape** (*pymunk.shapes.Poly*) – The cell's pymunk body object
- **ID** (*int*) – A unique identifier for each cell. At the moment just a number from 0 to 100_000_000 and cross fingers that we get no collisions.

create_pm_cell()

Creates a pymunk (pm) cell object, and places it into the pymunk space given when initialising the cell. If the cell is dividing, then two cells will be created. Typically this function is called for every cell, in every timestep to update the entire simulation.

Note: The return type of this function is dependent on the value returned by *SyMBac.cell.Cell.is_dividing()*. This is not good, and will be changed in a future version.

Returns

If *SyMBac.cell.Cell.is_dividing()* returns *True*, then a dictionary of values for the daughter cell is returned. A daughter can then be created. E.g:

```
>>> daughter_details = cell.create_pm_cell()
>>> daughter = Cell(**daughter_details)
```

If *SyMBac.cell.Cell.is_dividing()* returns *False*, then only a tuple containing (pymunk.body, pymunk.shape) will be returned.

Return type dict or (pymunk.body, pymunk.shape)

`get_angle()`

Gets the angle of the cell's pymunk body.

Returns **angle** – The cell's angle in radians.

Return type float

`get_centroid()`

Calculates the centroid of the cell from the vertices.

Returns **centroid** – The cell's centroid in coordinates relative to the pymunk space which the cell exists in.

Return type float

`get_vertex_list()`

Calculates the vertex list (a set of x,y coordinates) which parameterise the outline of the cell

Returns **vertices** – A list of vertices, each in a tuple, where the order is *(x, y)*. The coordinates are relative to the pymunk space in which the cell exists.

Return type list(tuple(float, float))

`is_dividing()`

Checks whether a cell is dividing by comparing its current length to its max length (defined when instnatiated).

Returns **output** – *True* if `self.length > self.max_length`, else *False*.

Return type bool

`update_length()`

A method, typically called every timepoint to update the length of the cell according to `self.length = self.length + self.growth_rate_constant*self.dt*self.length`.

Contains additional logic to control the amount of cell pinching happening according to the difference between the maximum length and the current length.

Return type None

`update_parent(parent)`

Parameters **parent** (*SyMBac.cell.Cell*) – The SyMBac cell object to assign as the parent to the current cell.

Return type None

update_position()

A method, typically called every timepoint to keep synchronised the cell position (`self.position` and `self.angle`) with the position of the cell's corresponding body in the pymunk space (`self.body.position` and `self.body.angle`).

Return type None

1.10 SyMBac.PSF

class SyMBac.PSF.**Camera**(*baseline*, *sensitivity*, *dark_noise*)

Class for instantiating Camera objects.

Example:

```
>>> my_camera = Camera(baseline=100, sensitivity=2.9, dark_noise=8)
>>> my_camera.render_dark_image()
```

__init__(*baseline*, *sensitivity*, *dark_noise*)

Parameters

- **baseline** (*int*) – The baseline intensity of the camera.
- **sensitivity** (*float*) – The camera sensitivity.
- **dark_noise** – The camera dark noise

render_dark_image(*size*, *plot=True*)

Render a sample synthetic dark image from the camera

Parameters

- **size** (*tuple(int, int)*) – Size of the dark image.
- **plot** (*bool*) – Whether or not to plot the image.

Returns Dark image sample.

Return type np.ndarray

class SyMBac.PSF.**PSF_generator**(*radius*, *wavelength*, *NA*, *n*, *apo_sigma*, *mode*, *condenser=None*, *z_height=None*, *resize_amount=None*, *pix_mic_conv=None*, *scale=None*, *offset=0*, *pz=0*, *working_distance=None*)

Instantiate a PSF generator, allows you to create phase contrast or fluorescence PSFs.

Example:

```
>>> #Creating a phase contrast PSF
>>> my_kernel = PSF_generator(
        radius = 50,
        wavelength = 0.75,
        NA = 1.2,
        n = 1.3,
        resize_amount = 3,
        pix_mic_conv = 0.065,
        apo_sigma = 10,
```

(continues on next page)

(continued from previous page)

```
        mode="phase contrast",
        condenser = "Ph3"
    )
>>> my_kernel.calculate_PSF()
>>> my_kernel.plot_PSF()
```

__init__(*radius*, *wavelength*, *NA*, *n*, *apo_sigma*, *mode*, *condenser=None*, *z_height=None*, *resize_amount=None*, *pix_mic_conv=None*, *scale=None*, *offset=0*, *pz=0*, *working_distance=None*)

Parameters

- **radius** (*int*) – Radius of the PSF.
- **wavelength** (*float*) – Wavelength of imaging light in micron.
- **NA** (*float*) – Numerical aperture of the objective lens.
- **n** (*float*) – Refractive index of the imaging medium.
- **apo_sigma** (*float*) – Gaussian apodisation sigma for phase contrast PSF (will be ignored for fluorescence PSFs).
- **mode** (*str*) – Either `phase contrast`, `simple fluo`, or *3d fluo* `.
- **condenser** (*str*) – Either `Ph1`, `Ph2`, `Ph3`, `Ph4`, or `PhF` (will be ignored for fluorescence PSFs).
- **z_height** (*int*) – The Z-size of a 3D fluorescence PSF. Will be ignored for `mode=phase contrast` or `simple fluo`.
- **resize_amount** (*int*) – Upscaling factor, typically chosen to be 3.
- **pix_mic_conv** (*float*) – Micron per pixel conversion factor. E.g approx 0.1 for 60x on some cameras.
- **scale** (*float*) – If not provided will be calculated as `self.pix_mic_conv / self.resize_amount`.
- **offset** (*float*) – A constant offset to add to the PSF, increases accuracy of long range effects, especially useful for colony simulations.``.

static gaussian_2D(*size*,)

Returns a 2D gaussian (numpy array) of size (pixels x pixels) and gaussian radius ()

static get_condensers()

Returns a dictionary of common phase contrast condenser dimensions, where the numbers are W, R, diameter (in mm)

static get_fluorescence_kernel(*wavelength*, *NA*, *n*, *radius*, *scale*, *offset=0*)

Returns a 2D numpy array which is an airy-disk approximation of the fluorescence point spread function

Parameters

- **Lambda** (*float*) – Wavelength of imaging light (micron)
- **NA** (*float*) – Numerical aperture of the objective lens
- **n** (*float*) – Refractive index of the imaging medium (~1 for air, ~1.4-1.5 for oil)
- **radius** (*int*) – The radius of the PSF to be rendered in pixels
- **scale** (*float*) – The pixel size of the image to be rendered (micron/pix)

- **offset** (*float*) – A constant offset to add to the PSF, increases accuracy of long range effects, especially useful for colony simulations.

Return type 2-D numpy array representing the fluorescence contrast PSF

static get_phase_contrast_kernel(*R*, *W*, *radius*, *scale*, *NA*, *n*, *sigma*, *wavelength*, *offset=0*)

Returns a 2D numpy array which is the phase contrast kernel based on microscope parameters

Parameters

- **R** (*float*) – The radius of the phase contrast condenser (in mm)
- **W** (*float*) – The width of the phase contrast condenser opening (in mm)
- **radius** (*int*) – The radius of the PSF to be rendered in pixels
- **scale** (*float*) – The pixel size of the image to be rendered (micron/pix)
- **NA** (*float*) – Numerical aperture of the objective lens
- **n** (*float*) – Refractive index of the imaging medium (~1 for air, ~1.4-1.5 for oil)
- **sigma** (*float*) – radius of a 2D gaussian of the same size as the PSF (in pixels) which is multiplied by the PSF to simulate apodisation of the PSF
- (*float*) – The mean wavelength of the imaging light (in micron)
- **offset** (*float*) – A constant offset to add to the PSF, increases accuracy of long range effects, especially useful for colony simulations.

Return type 2-D numpy array representing the phase contrast PSF

static somb(*x*)

Returns the sombrero function of a 2D numpy array, defined as

$$somb(x) = \frac{2J_1(\pi x)}{\pi x}$$

1.11 SyMBac.renderer

class SyMBac.renderer.**Renderer**(*simulation*, *PSF*, *real_image*, *camera=None*, *additional_real_images=None*)

Instantiates a renderer, which given a simulation, PSF, real image, and optionally a camera, generates the synthetic data

Example:

```
>>> from SyMBac.renderer import Renderer
>>> my_renderer = Renderer(my_simulation, my_kernel, real_image, my_camera)
>>> my_renderer.select_intensity_napari()
>>> my_renderer.optimise_synth_image(manual_update=False)
>>> my_renderer.generate_training_data(
        sample_amount=0.2,
        randomise_hist_match=True,
        randomise_noise_match=True,
        burn_in=40,
        n_samples = 500,
        save_dir="/tmp/test/",
        in_series=False
    )
```

__init__(*simulation*, *PSF*, *real_image*, *camera=None*, *additional_real_images=None*)

Parameters

- **simulation** (SyMBac.simulation.Simulation) – The SyMBac simulation.
- **PSF** (*SyMBac.psf.PSF_generator*) – The PSF to be applied to the synthetic data.
- **real_image** (*np.ndarray*) – A real image sample
- **camera** (SyMBac.PSF.Camera) – (optional) The simulation camera object to be applied to the synthetic data
- **additional_real_images** (*List*) – List of additional images which will be randomly used to fourier match during the rendering process.

generate_PC_OPL(*scene*, *mask*, *media_multiplier*, *cell_multiplier*, *device_multiplier*, *y_border_expansion_coefficient*, *x_border_expansion_coefficient*, *defocus*)

Takes a scene drawing, adds the trenches and colours all parts of the image to generate a first-order phase contrast image, uncorrupted (unconvolved) by the phase contrat optics. Also has a fluorescence parameter to quickly switch to fluorescence if you want.

Parameters

- **main_segments** (*list*) – A list of the trench segments, used for drawing the trench
- **offset** (*int*) – The same offset from the draw_scene function. Used to know the cell offset.
- **scene** (*2D numpy array*) – A scene image
- **mask** (*2D numpy array*) – The mask for the scene
- **media_multiplier** (*float*) – Intensity multiplier for media (the area between cells which isn't the device)
- **cell_multiplier** (*float*) – Intensity multiplier for cell
- **device_multiplier** (*float*) – Intensity multiplier for device
- **y_border_expansion_coefficient** (*int*) – Another offset-like argument. Multiplies the size of the image on each side by this value. 3 is a good starting value because you want the image to be relatively larger than the PSF which you are convolving over it.
- **x_border_expansion_coefficient** (*int*) – Another offset-like argument. Multiplies the size of the image on each side by this value. 3 is a good starting value because you want the image to be relatively larger than the PSF which you are convolving over it.
- **fluorescence** (*bool*) – If true converts image to a fluorescence (hides the trench and swaps to the fluorescence PSF).
- **defocus** (*float*) – Simulated optical defocus by convolving the kernel with a 2D gaussian of radius defocus.

Returns

- **expanded_scene** (*2D numpy array*) – A large (expanded on x and y axis) image of cells in a trench, but unconvolved. (The raw PC image before convolution)
- **expanded_scene_no_cells** (*2D numpy array*) – Same as expanded_scene, except with the cells removed (this is necessary for later intensity tuning)
- **expanded_mask** (*2D numpy array*) – The masks for the expanded scene

generate_test_comparison(*media_multiplier=75*, *cell_multiplier=1.7*, *device_multiplier=29*, *sigma=8.85*, *scene_no=- 1*, *match_fourier=False*, *match_histogram=True*, *match_noise=False*, *debug_plot=False*, *noise_var=0.001*, *defocus=3.0*, *halo_top_intensity=1*, *halo_bottom_intensity=1*, *halo_start=0*, *halo_end=1*, *random_real_image=None*)

Takes all the parameters we've defined and calculated, and uses them to finally generate a synthetic image.

Parameters

- **media_multiplier** (*float*) – Intensity multiplier for media (the area between cells which isn't the device)
- **cell_multiplier** (*float*) – Intensity multiplier for cell
- **device_multiplier** (*float*) – Intensity multiplier for device
- **sigma** (*float*) – Radius of a gaussian which simulates PSF apodisation
- **scene_no** (*int in range(len(cell_timeseries_properties))*) – The index of which scene to render
- **scale** (*float*) – The micron/pixel value of the image
- **match_fourier** (*bool*) – If true, use sfmatch to match the rotational fourier spectrum of the synthetic image to a real image sample
- **match_histogram** (*bool*) – If true, match the intensity histogram of a synthetic image to a real image
- **offset** (*int*) – The same offset value from draw_scene
- **debug_plot** (*bool*) – True if you want to see a quick preview of the rendered synthetic image
- **noise_var** (*float*) – The variance for the simulated camera noise (gaussian)
- **kernel** (SyMBac.PSF.PSF_generator) – A kernel object from SyMBac.PSF.PSF_generator
- **resize_amount** (*int*) – The upscaling factor to render the image by. E.g a resize_amount of 3 will interally render the image at 3x resolution before convolving and then downsampling the image. Values >2 are recommended.
- **real_image** (*2D numpy array*) – A sample real image from the experiment you are trying to replicate
- **image_params** (*tuple*) – A tuple of parameters which describe the intensities and variances of the real image, in this order: (real_media_mean, real_cell_mean, real_device_mean, real_means, real_media_var, real_cell_var, real_device_var, real_vars).
- **error_params** (*tuple*) – A tuple of parameters which characterises the error between the intensities in the real image and the synthetic image, in this order: (mean_error,media_error,cell_error,device_error,mean_var_error,media_var_error, cell_var_error,device_var_error). I have given an example of their calculation in the example notebooks.
- **fluorescence** (*bool*) – If true converts image to a fluorescence (hides the trench and swaps to the fluorescence PSF).
- **defocus** (*float*) – Simulated optical defocus by convolving the kernel with a 2D gaussian of radius defocus.

- **halo_top_intensity** (*float*) – Simulated "halo" caused by the microfluidic device. This sets the starting muliplier of a linear ramp which is applied down the length of the image in the direction of the trench. ,
- **halo_bottom_intensity** (*float*) – Simulated "halo" caused by the microfluidic device. This sets the ending multiplier of a lienar ramp which is applied down the length of the image. E.g, if `image` has shape `(y, x)`, then this results in `image = image * np.linspace(halo_lower_int,halo_upper_int, image.shape[0])[:, None]`.

Returns

- **noisy_img** (*2D numpy array*) – The final simulated microscope image
- **expanded_mask_resized_reshaped** (*2D numpy array*) – The final image's accompanying masks

generate_training_data(*sample_amount*, *randomise_hist_match*, *randomise_noise_match*, *burn_in*, *n_samples*, *save_dir*, *in_series=False*, *seed=False*, *n_jobs=1*, *dtype=<class 'numpy.uint8'>*)

Generates the training data from a Jupyter interactive output of generate_test_comparison

Parameters

- **sample_amount** (*float*) – The percentage sampling variance (drawn from a uniform distribution) to vary intensities by. For example, a sample_amount of 0.05 will randomly sample +/- 5% above and below the chosen intensity for cells, media and device. Can be used to create a little bit of variance in the final training data.
- **randomise_hist_match** (*bool*) – If true, histogram matching is randomly turned on and off each time a training sample is generated
- **randomise_noise_match** (*bool*) – If true, noise matching is randomly turned on and off each time a training sample is generated
- **burn_in** (*int*) – Number of frames to wait before generating training data. Can be used to ignore the start of the simulation where the trench only has 1 cell in it.
- **n_samples** (*int*) – The number of training images to generate
- **save_dir** (*str*) – The save directory of the training data
- **in_series** (*bool*) – Whether the images should be randomly sampled, or rendered in the order that the simulation was run in.
- **seed** (*float*) – Optional arg, if specified then the numpy random seed will be set for the rendering, allows reproducible rendering results.

optimise_synth_image(*manual_update*)

Parameters **manual_update** (*bool*) – Whether to turn on manual updating. This is recommended if you have no/a slow GPU. Will display a button to allow manual updating of the image optimiser

Returns ipywidget object for optimisation of synthetic data

1.12 SyMBac.simulation

class SyMBac.simulation.**Simulation**(*trench_length*, *trench_width*, *cell_max_length*, *max_length_var*, *cell_width*, *width_var*, *lysis_p*, *sim_length*, *pix_mic_conv*, *gravity*, *phys_iters*, *resize_amount*, *save_dir*)

Class for instantiating Simulation objects. These are the basic objects used to run all SyMBac simulations. This class is used to parameterise simulations, run them, draw optical path length images, and then visualise them.

Example:

```
>>> from SyMBac.simulation import Simulation
>>> my_simulation = Simulation(
        trench_length=15,
        trench_width=1.3,
        cell_max_length=6.65, #6, long cells # 1.65 short cells
        cell_width= 1, #1 long cells # 0.95 short cells
        sim_length = 100,
        pix_mic_conv = 0.065,
        gravity=0,
        phys_iters=15,
        max_length_var = 0.,
        width_var = 0.,
        lysis_p = 0.,
        save_dir="/tmp/",
        resize_amount = 3
    )
>>> my_simulation.run_simulation(show_window=False)
>>> my_simulation.draw_simulation_OPL(do_transformation=True, label_masks=True)
>>> my_simulation.visualise_in_napari()
```

__init__(*trench_length*, *trench_width*, *cell_max_length*, *max_length_var*, *cell_width*, *width_var*, *lysis_p*, *sim_length*, *pix_mic_conv*, *gravity*, *phys_iters*, *resize_amount*, *save_dir*)

Initialising a Simulation object

Parameters

- **trench_length** (*float*) – Length of a mother machine trench (micron)
- **trench_width** (*float*) – Width of a mother machine trench (micron)
- **cell_max_length** (*float*) – Maximum length a cell can reach before dividing (micron)
- **cell_width** (*float*) – the average cell width in the simulation (micron)
- **pix_mic_conv** (*float*) – The micron/pixel size of the image
- **gravity** (*float*) – Pressure forcing cells into the trench. Typically left at zero, but can be varied if cells start to fall into each other or if the simulation behaves strangely.
- **phys_iters** (*int*) – Number of physics iterations per simulation frame. Increase to resolve collisions if cells are falling into one another, but decrease if cells begin to repel one another too much (too high a value causes cells to bounce off each other very hard). 20 is a good starting point
- **max_length_var** (*float*) – Variance of the maximum cell length
- **width_var** (*float*) – Variance of the maximum cell width
- **save_dir** (*str*) – Location to save simulation output

- **lysis_p** (*float*) – probability of cell lysis
- **sim_length** (*int*) – number of frames to simulate (where each one is dt). Start with 200-1000, and grow your simulation from there.
- **resize_amount** (*int*) – This is the "upscaling" factor for the simulation and the entire image generation process. Must be kept constant across the image generation pipeline. Starting value of 3 recommended.

run_simulation(*show_window=True*, *streamlit_mode=False*)

Run the simulation

Parameters **show_window** (*bool*) – Whether to show the pyglet window while running the simulation. Typically would be *false* if running SyMBac headless.

visualise_in_napari()

Opens a napari window allowing you to visualise the simulation, with both masks, OPL images, interactively. :return:

1.13 SyMBac.cell_geometry

SyMBac.cell_geometry.**centroid**(*vertices*)

Return the centroid of a list of vertices

Parameters **vertices** (*list(tuple)*) – A list of tuples containing x,y coordinates.

SyMBac.cell_geometry.**get_vertices**(*cell_length*, *cell_width*, *angle*, *resolution*)

Generates coordinates for a cell centered aronud (0,0)

Parameters

- **cell_length** (*float*) – The length of the STRAIGHT part of the cell's wall. Total length is cell_length + cell_width because the poles are models as semi-circles.
- **cell_width** (*float*) – Total thickness of the cell, defines the poles too.
- **angle** (*float*) – Angle in radians to rotate the cell by (counter-clockwise)
- **resolution** (*int*) – Number of points defining the cell wall geometry

Return type list of lists containing cell x and y coords

Examples

Create a cell of length 10+4 rotated by 1 radian with a resolution of 20:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> verts = get_vertices(10,4,1,20)
>>> verts_y = [y[0] for y in verts]
>>> verts_x = [x[1] for x in verts]
>>> plt.plot(verts_x,verts_y)
```

SyMBac.cell_geometry.**rotate**(*origin*, *point*, *angle*)

Rotate a point counterclockwise by a given angle around a given origin.

The angle should be given in radians.

SyMBac.cell_geometry.**wall**(*thickness*, *start*, *end*, *t_or_b*, *resolution*)

Generates the straight part of the cell wall's coordinates (all but the poles)

Parameters

- **thickness** (*float*) – The distance from the top cell wall to the bottom cell wall
- **start** (*float*) – The start coordinate of the cell wall
- **end** (*float*) – The end coordinate of the cell wall
- **t_or_b** (*int*) – 0 for top wall 1 for bottom wall
- **resolution** (*int*) – Number of points defining the cell wall geometry

Returns return[0] is the wall's x coordinates return[0] is the wall's y coordiantes

Return type tuple(Numpy Array, Numpy Array)

Examples

Create two cell walls of length 10, 3 apart

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> top_wall = wall(3,0,10,0,20)
>>> bottom_wall = top_wall = wall(3,0,10,1,20)
>>> plt.plot(walls[0], walls[1])
>>> plt.plot(walls[0], walls[1])
>>> plt.show()
```

1.14 SyMBac.cell_simulation

SyMBac.cell_simulation.**create_space**()

Creates a pymunk space

Return pymunk.Space space A pymunk space

SyMBac.cell_simulation.**run_simulation**(*trench_length*, *trench_width*, *cell_max_length*, *cell_width*, *sim_length*, *pix_mic_conv*, *gravity*, *phys_iters*, *max_length_var*, *width_var*, *save_dir*, *lysis_p=0*, *show_window=True*, *streamlit_mode=False*)

Runs the rigid body simulation of bacterial growth based on a variety of parameters. Opens up a Pyglet window to display the animation in real-time. If the simulation looks bad to your eye, restart the kernel and rerun the simulation. There is currently a bug where if you try to rerun the simulation in the same kernel, it will be extremely slow.

Parameters

- **trench_length** (*float*) – Length of a mother machine trench (micron)
- **trench_width** (*float*) – Width of a mother machine trench (micron)
- **cell_max_length** (*float*) – Maximum length a cell can reach before dividing (micron)
- **cell_width** (*float*) – the average cell width in the simulation (micron)
- **pix_mic_conv** (*float*) – The micron/pixel size of the image

- **gravity** (*float*) – Pressure forcing cells into the trench. Typically left at zero, but can be varied if cells start to fall into each other or if the simulation behaves strangely.
- **phys_iters** (*int*) – Number of physics iterations per simulation frame. Increase to resolve collisions if cells are falling into one another, but decrease if cells begin to repel one another too much (too high a value causes cells to bounce off each other very hard). 20 is a good starting point
- **max_length_var** (*float*) – Variance of the maximum cell length
- **width_var** (*float*) – Variance of the maximum cell width
- **save_dir** (*str*) – Location to save simulation output
- **lysis_p** (*float*) – probability of cell lysis

Returns

- **cell_timeseries** (*lists*) – A list of parameters for each cell, such as length, width, position, angle, etc. All used in the drawing of the scene later
- **space** (*a pymunk space object*) – Contains the rigid body physics objects which are the cells.

SyMBac.cell_simulation.**step_and_update**(*dt*, *cells*, *space*, *phys_iters*, *ylim*, *cell_timeseries*, *x*, *sim_length*, *save_dir*)

Evolves the simulation forward

Parameters

- **dt** (*float*) – The simulation timestep
- **cells** (*list(SyMBac.cell.Cell)*) – A list of all cells in the current timestep
- **space** (*pymunk.Space*) – The simulations's pymunk space.
- **phys_iters** (*int*) – The number of physics iteration in each timestep
- **ylim** (*int*) – The y coordinate threshold beyond which to delete cells
- **cell_timeseries** (*list*) – A list to store the cell's properties each time the simulation steps forward
- **list** (*int*) – A list with a single value to store the simulation's progress.
- **sim_length** (*int*) – The number of timesteps to run.
- **save_dir** (*str*) – The directory to save the simulation information.

Returns **cells**

Return type list(*SyMBac.cell.Cell*)

SyMBac.cell_simulation.**update_cell_lengths**(*cells*)

Iterates through all cells in the simulation and updates their length according to their growth law.

Parameters **cells** (*list(SyMBac.cell.Cell)*) – A list of all cells in the current timepoint of the simulation.

SyMBac.cell_simulation.**update_cell_parents**(*cells*, *new_cells*)

Takes two lists of cells, one in the previous frame, and one in the frame after division, and updates the parents of each cell

Parameters

- **cells** (*list(SyMBac.cell.Cell)*) –
- **new_cells** (*list(SyMBac.cell.Cell)*) –

SyMBac.cell_simulation.**update_cell_positions**(*cells*)

Iterates through all cells in the simulation and updates their positions, keeping the cell object's position synchronised with its corresponding pymunk shape and body inside the pymunk space.

Parameters **cells** (*list(SyMBac.cell.Cell)*) – A list of all cells in the current timepoint of the simulation.

SyMBac.cell_simulation.**update_pm_cells**(*cells*)

Iterates through all cells in the simulation and updates their pymunk body and shape objects. Contains logic to check for cell division, and create daughters if necessary.

Parameters **cells** (*list(SyMBac.cell.Cell)*) – A list of all cells in the current timepoint of the simulation.

SyMBac.cell_simulation.**wipe_space**(*space*)

Deletes all cells in the simulation pymunk space.

Parameters **space** (*pymunk.Space*) –

1.15 SyMBac.drawing

SyMBac.drawing.**OPL_to_FL**(*cell*, *density*)

Parameters

- **cell** (*np.ndarray*) – A 2D numpy array consisting of a rasterised cell
- **density** (*float*) – Number of fluorescent molecules per volume element to sample in the cell

Returns A cell with fluorescent reporters sampled in it

Rtypes np.ndarray

SyMBac.drawing.**draw_scene**(*cell_properties*, *do_transformation*, *space_size*, *offset*, *label_masks*, *pinching=True*)

Draws a raw scene (no trench) of cells, and returns accompanying masks for training data.

Parameters

- **properties** (*cell*) – A list of cell properties for that frame
- **do_transformation** (*bool*) – True if you want cells to be bent, false and cells remain straight as in the simulation
- **space_size** (*tuple*) – The xy size of the numpy array in which the space is rendered. If too small then cells will not fit. recommend using the *SyMBac.drawing.get_space_size()* function to find the correct space size for your simulation
- **offset** (*int*) – A necessary parameter which offsets the drawing a number of pixels from the left hand side of the image. 30 is a good number, but if the cells are very thick, then might need increasing.
- **label_masks** (*bool*) – If true returns cell masks which are labelled (good for instance segmentation). If false returns binary masks only. I recommend leaving this as True, because you can always binarise the masks later if you want.
- **pinching** (*bool*) – Whether or not to simulate cell pinching during division

Returns

- **space, space_masks** (*2D numpy array, 2D numpy array*)
- **space** (*2D numpy array*) – Not to be confused with the pyglet object calledspace in some other functions. Simply a 2D numpy array with an image of cells from the input frame properties
- **space_masks** (*2D numy array*) – The masks (labelled or bool) for that scene.

SyMBac.drawing.**find_farthest_vertices**(*vertex_list*)

Given a list of vertices, find the pair of vertices which are farthest from each other

Parameters **vertex_list** (*list(tuple(float, float))*) – List of pairs of vertices [(x,y), (x,y), ...]

Returns The two vertices maximally far apart

Return type array(tuple(float, float))

SyMBac.drawing.**gen_cell_props_for_draw**(*cell_timeseries_lists*, *ID_props*)

Parameters

- **cell_timeseries_lists** (*list*) – A list (single frame) from cell_timeseries, the output from run_simulation. E.g: cell_timeseries[x]
- **ID_props** (*list*) – A list of properties for each cell in that frame, the output of generate_curve_props()

Returns **cell_properties** – The final property list used to actually draw a scene of cells. The input to draw_scene

Return type list

SyMBac.drawing.**generate_curve_props**(*cell_timeseries*)

Generates individual cell curvature properties. 3 parameters for each cell, which are passed to a cosine function to modulate the cell's curvature.

Parameters **cell_timeseries** (*list(cell_properties)*) – The output of *SyMBac.simulation.Simulation.run_simulation()*

Returns **outupt**

Return type A numpy array of unique curvature properties for each cell in the simulation

SyMBac.drawing.**get_centroid**(*vertices*)

Return the centroid of a list of vertices

Parameters **vertices** (*list(tuple(float, float))*) – List of tuple of vertices where each tuple is (x, y)

Returns Centroid of the vertices.

Return type tuple(float, float)

SyMBac.drawing.**get_distance**(*vertex1*, *vertex2*)

Get euclidian distance between two sets of vertices.

Parameters

- **vertex1** (*tuple(float, float)*) – Vertex 1
- **vertex2** (*tuple(float, float)*) – Vertex 2

Returns Absolute distance between two points

Return type float

SyMBac.drawing.**get_midpoint**(*vertex1*, *vertex2*)

Get the midpoint between two vertices.

Parameters

- **vertex1** (*tuple(float, float)*) – Vertex 1
- **vertex2** (*tuple(float, float)*) – Vertex 2

Returns Midpoint between vertex 1 and 2

Return type tuple(float, float)

SyMBac.drawing.**get_space_size**(*cell_timeseries_properties*)

Parameters **cell_timeseries_properties** – A list of cell properties over time. Generated from SyMBac.simulation.Simulation.draw_simulation_OPL()

Returns Iterates through the simulation timeseries properties, finds the extreme cell positions and retrieves the required image size to fit all cells into.

Return type tuple(float, float)

SyMBac.drawing.**make_images_same_shape**(*real_image*, *synthetic_image*, *rescale_int=True*)

Makes a synthetic image the same shape as the real image

SyMBac.drawing.**midpoint_intercept**(*vertex1*, *vertex2*)

Get the y-intercept of the line connecting two vertices

Parameters

- **vertex1** (*tuple(float, float)*) – Vertex 1
- **vertex2** (*tuple(float, float)*) – Vertex 2

Returns Y indercept of line between vertex 1 and 2

Return type float

SyMBac.drawing.**place_cell**(*length*, *width*, *angle*, *position*, *space*)

Creates a cell and places it in the pymunk space

Parameters

- **length** (*float*) – length of the cell
- **width** (*float*) – width of the cell
- **angle** (*float*) – rotation of the cell in radians counterclockwise
- **position** (*tuple*) – x,y coordinates of the cell centroid
- **space** (*pymunk.space.Space*) – Pymunk space to place the cell in

Return type nothing, updates space

SyMBac.drawing.**raster_cell**(*length*, *width*, *separation*, *pinching=True*, *FL=False*)

Produces a rasterised image of a cell with the intensiity of each pixel corresponding to the optical path length (thickness) of the cell at that point.

Parameters

- **length** (*int*) – Cell length in pixels
- **width** (*int*) – Cell width in pixels
- **separation** (*int*) – An int between (0, *width*) controlling how much pinching is happening.

- **pinching** (*bool*) – Controls whether pinching is happening

Returns **cell** – A numpy array which contains an OPL image of the cell. Can be converted to a mask by just taking `cell > 0`.

Return type np.array

SyMBac.drawing.**vertices_slope**(*vertex1*, *vertex2*)

Get the slope between two vertices

Parameters

- **vertex1** (*tuple(float, float)*) – Vertex 1
- **vertex2** (*tuple(float, float)*) – Vertex 2

Returns Slope between vertex 1 and 2

Return type float

1.16 SyMBac.misc

SyMBac.misc.**get_sample_images**()

Return a dict of sample mother machine images.

Returns A dict with sample images, current keys are: "E. coli 100x", "E. coli 100x stationary", "E. coli DeLTA"

Return type dict

SyMBac.misc.**resize_mask**(*mask*, *resize_shape*, *ret_label*)

Resize masks while maintaining their connectivity and values

Parameters

- **mask** (*np.ndarray*) – Input mask
- **resize_shape** (*tuple(int, int)*) – Shape to resize the mask to
- **ret_label** (*bool*) – Whether to return labeled or bool masks

Returns Resized mask

Return type np.ndarray

SyMBac.misc.**unet_weight_map**(*y*, *wc=None*, *w0=10*, *sigma=5*)

Generate weight maps as specified in the U-Net paper for boolean mask.

Parameters

- **mask** (*Numpy array*) – 2D array of shape (image_height, image_width) representing binary mask of objects.
- **wc** (*dict*) – Dictionary of weight classes.
- **w0** (*int*) – Border weight parameter.
- **sigma** (*int*) – Border width parameter.

Returns Training weights. A 2D array of shape (image_height, image_width).

Return type Numpy array

References

Taken from the original U-net paper[1]

1.17 SyMBac.pySHINE

This module is a set of selected Python translation of functions from the SHINE toolbox, which was originally written in MATLAB. Please see the references in the function documentation.

SyMBac.pySHINE.**lumMatch**(*images*, *mask=None*, *lum=None*)

Match the luminosity of a stack of images. For documentation see [1]_

References

SyMBac.pySHINE.**rescale_shine**(*images*, *option=1*)

Rescale the intensity of a stack of images. For documentation see [1]_

References

SyMBac.pySHINE.**sfMatch**(*images*, *rescaling=0*, *tarmag=None*)

Match the rotational fourier spectrum of a stack of images. For documentation see [1]_

References

[1] Ronneberger, O., Fischer, P., Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. In: Navab, N., Hornegger, J., Wells, W., Frangi, A. (eds) Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. MICCAI 2015. Lecture Notes in Computer Science(), vol 9351. Springer, Cham. https://doi.org/10.1007/978-3-319-24574-4_28

PYTHON MODULE INDEX

S

INDEX

Symbols

C

D

F

G

I

L

M

O

P

R

S

U

V

W