scikit-mpe

Release 0.2.4

Eugene Prilepin

May 23, 2021

CONTENTS

scikit-mpe is a package for extracting a minimal path in n-dimensional Euclidean space (on regular Cartesian grids) using the fast marching method.

The package can be used in various engineering and image processing tasks. For example, it can be used for extracting paths through tubular structures on 2-d and 3-d images, or shortest paths on terrain maps.

CHAPTER

ONE

INSTALLING

Python 3.6 or above is supported. You can install the package using pip:

```
pip install -U scikit-mpe
```

CHAPTER

TWO

A SIMPLE EXAMPLE

Here is the simple example: how to extract 2-d minimal path using some speed data.

```
from skmpe import mpe

# Somehow speed data is calculating
speed_data = get_speed_data()

# Extracting minimal path from the starting point to the ending point
path_info = mpe(speed_data, start_point=(10, 20), end_point=(120, 45))

# Getting the path data in numpy ndarray
path = path_info.path
```

CHAPTER

THREE

CONTENTS

3.1 Tutorial

3.1.1 Overview

scikit-mpe package allows you to extract N-dimensional minimal paths using existing speed data and starting/ending and optionally way points initial data.

Note: The package does not compute any speed data (a.k.a speed function). It is expected that the speed data was previously obtained/computed in some way.

The package can be useful for various engineering and image processing tasks. For example, the package can be used for extracting paths through tubular structures on 2-d and 3-d images, or shortest paths on a terrain map.

The package uses the fast marching method and ODE solver for extracting minimal paths.

Algorithm

The algorithm contains two main steps:

- First, the travel time is computing from the given ending point (zero contour) to every speed data point using the fast marching method.
- Second, the minimal path (travel time is minimizing) is extracting from the starting point to the ending point using ODE solver (Runge-Kutta for example) for solving the differential equation $x_t = -\nabla(t)/|\nabla(t)|$

If we have way points we need to perform these two steps for every interval between the starting point, the set of the way points and the ending point and concatenate the path pieces to the full path.

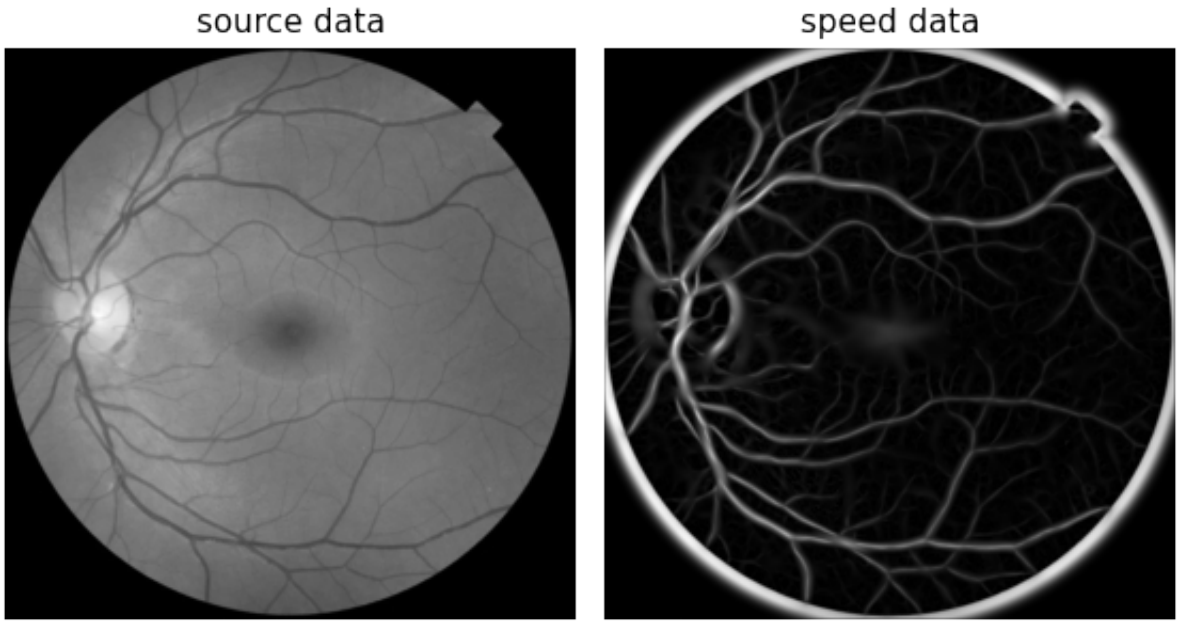
3.1.2 Quickstart

Let's look at a simple example of how the algorithm works.

Note: We will use retina test image from scikit-image package as the test data for all examples.

First, we need a speed data (speed function). We can use one of the tubeness filters for computing speed data for our test data, sato filter for example.

```
from skimage.data import retina
from skimage.color import rgb2gray
from skimage.transform import rescale
from skimage.filters import sato

image_data = rescale(rgb2gray(retina()), 0.5)
speed_data = sato(image_data) + 0.05
speed_data[speed_data > 1.0] = 1.0

_, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(image_data, cmap='gray')
ax1.set_title('source data')
ax1.axis('off')
ax2.imshow(speed_data, cmap='gray')
ax2.set_title('speed data')
ax2.axis('off')
```



The speed data values must be in range [0.0, 1.0] and can be masked also.

where:

- 0.0 – zero speed (impassable)
- 1.0 – max speed
- masked – impassable

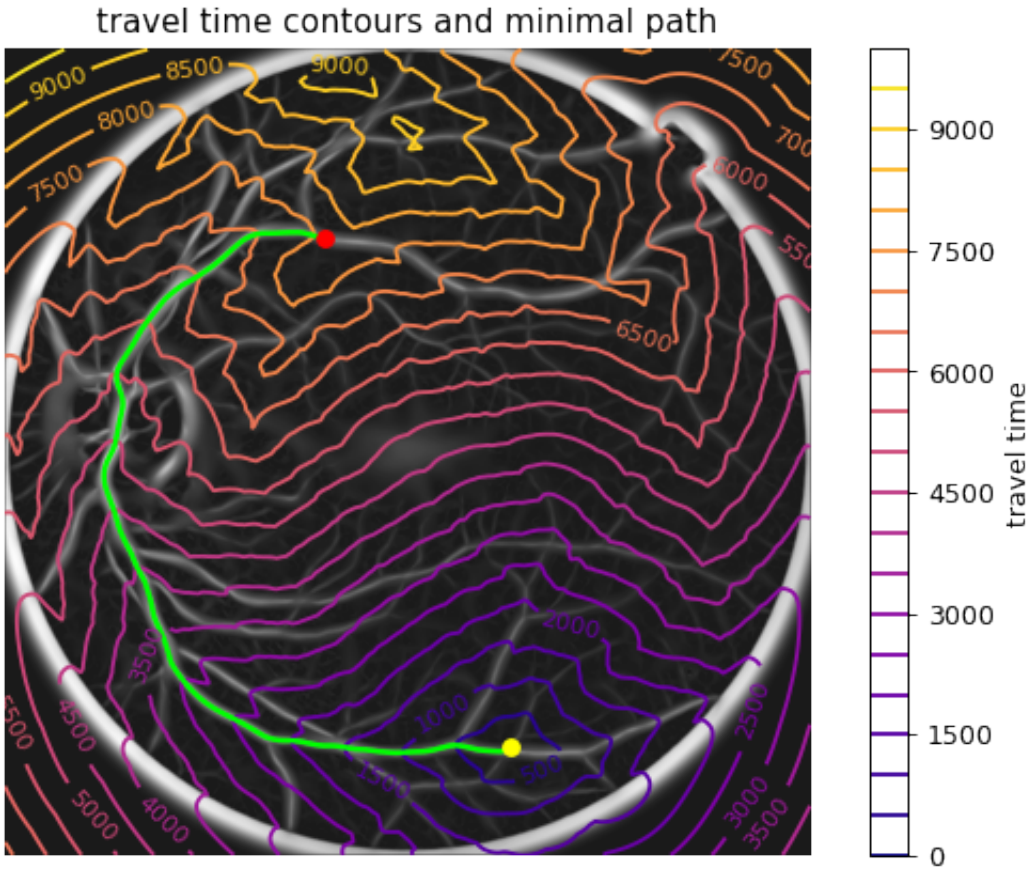
Second, let's try to extract the minimal path for some starting and ending points using **scikit-mpe** package and plot it. Also we can plot travel time contours.

```
from skmpe import mpe

# define starting and ending points
start_point = (165, 280)
end_point = (611, 442)
```

(continues on next page)

(continued from previous page)

```
path_info = mpe(speed_data, start_point, end_point)

# get computed travel time for given ending point and extracted path
travel_time = path_info.pieces[0].travel_time
path = path_info.path

nrows, ncols = speed_data.shape
xx, yy = np.meshgrid(np.arange(ncols), np.arange(nrows))

fig, ax = plt.subplots(1, 1)
ax.imshow(speed_data, cmap='gray', alpha=0.9)
ax.plot(path[:,1], path[:,0], '-', color=[0, 1, 0], linewidth=2)
ax.plot(start_point[1], start_point[0], 'or')
ax.plot(end_point[1], end_point[0], 'o', color=[1, 1, 0])
tt_c = ax.contour(xx, yy, travel_time, 20, cmap='plasma', linewidths=1.5)
ax.clabel(tt_c, inline=1, fontsize=9, fmt='%d')
ax.set_title('travel time contours and minimal path')
ax.axis('off')
cb = fig.colorbar(tt_c)
cb.ax.set_ylabel('travel time')
```



3.1.3 Advanced Usage

Initial Data

The initial data is storing and validating in *InitialInfo* class which inherited from Pydantic BaseModel. The class checks speed data and points dimensions, boundaries and values.

Therefore, we cannot set an invalid data:

```
import numpy as np
from skmpe import InitialInfo

speed_data = np.zeros((100, 200))
start_point = (10, 300)  # out of bounds
end_point = (50, 60)

init_data = InitialInfo(
    speed_data=speed_data,
    start_point=start_point,
    end_point=end_point,
)
```

The code above is raising an exception:

```
Traceback (most recent call last):
  ...
    raise validation_error
pydantic.error_wrappers.ValidationError: 1 validation error for InitialInfo
start_point
  'start_point' (10, 300) coordinate 1 is out of 'speed_data' bounds [0, 200).␣
↪(type=value_error)
```

We can use *InitialInfo* explicity in *mpe()* function:

```
from skmpe import InitialInfo, mpe

init_data = InitialInfo(...)
result = mpe(init_data)
```

Also in most cases we can use the second *mpe()* function signature without using *InitialInfo* explicity:

```
from skmpe import mpe

...

result = mpe(speed_data, start_point, end_point)
```

Parameters

The algorithm parameters are storing and validating in *Parameters* class. We can use this class directly, or we can use *parameters()* context manager for manage parameters.

Also *default_parameters()* function returns the instance with default parameters:

```
>>> from skmpe import default_parameters
>>> print(default_parameters())

travel_time_spacing=1.0
travel_time_order=<TravelTimeOrder.first: 1>
travel_time_cache=False
ode_solver_method=<OdeSolverMethod.RK45: 'RK45'>
integrate_time_bound=10000.0
integrate_min_step=0.0
integrate_max_step=4.0
dist_tol=0.001
max_small_dist_steps=100
```

Important Parameters

The following parameters may be important in some cases:

- **travel_time_order** – the order of the fast-marching computation method. 2 is more accurate, but it is slower. By default it is 1. Use *TravelTimeOrder* enum for this parameter
- **travel_time_cache** – if we set way points we can use cached travel time. For example if we set one way point we can compute travel time once for this way point as source point. By default it is False.
- **ode_solver_method** – we can use some ODE methods for extracting path. Some methods may be work faster or more accurate on some speed data. Use *OdeSolverMethod* enum for this parameter. By default it is Runge-Kutta 4/5 (*RK45*)
- **integrate_time_bound** – if we want to extract a long path we need to set a greater value for time bound. By default it is 10000
- **integrate_min_step**, **integrate_max_step** – these options can be used to control of ODE solver steps. For example, lower value of `integrate_max_step` leads to lower the performance, but higher the accuracy.
- **dist_tol** – distance tolerance between steps for control path evolution. By default it is 0.001
- **max_small_dist_steps** – the maximum number of small distance steps while path evolution. Too small steps will be ignore N-times by this parameter.

Using Parameters

We can set the custom parameter values by *Parameters* class or *parameters()* context manager.

Using class:

```
from skmpe import Parameters, mpe

my_parameters = Parameters(travel_time_cache=True, travel_time_order=1)
result = mpe(..., parameters=my_parameters)
```

Using context manager:

```
from skmpe import parameters, mpe

with parameters(travel_time_cache=True, travel_time_order=1):
    # the custom parameters will be used automatically
    result = mpe(...)
```

Results

The whole extracted path results are storing in *PathInfoResult* class (named tuple). The instance of this class is returning from *mpe()* function. The pieces of the path (in the case with way points) are storing in *PathInfo* class.

PathInfoResult object contains:

- **path** – the whole extracted path in numpy array MxN where M is the number of points and N is dimension
- **pieces** – the list of extracted path pieces between start/end or way points in *PathInfo* instances. If we do not use way points, **pieces** list will be contain one piece.

PathInfo object contains:

- **path** – the extracted path piece in numpy array MxN where M is the number of points and N is dimension
- **start_point** – the starting point
- **end_point** – the ending point
- **travel_time** – the computed travel time data for given speed data
- **extraction_result** – the raw extraction result in *PathExtractionResult* instance. This data is returning from *MinimalPathExtractor* class (low-level API). The data contains additional info about extracted path and info about extracting process. This data may be useful for debugging.
- **reversed** – The flag indicates that the path piece is reversed. This is relevant when using `travel_time_cache == True` parameter.

Low-level API

If you need full control over the extracting process, you can use *MinimalPathExtractor* class. The class is used inside *mpe()* function for extracting the pieces of path.

Here is a simple example of usage:

```
from skmpe import MinimalPathExtractor, Parameters

speed_data = ...
start_point = ...
end_point = ...
parameters = Parameters(...)  # optional, also we can use 'parameters' context manager

# Create the instance and compute travel time data
mpe = MinimalPathExtractor(speed_data, end_point, parameters)

# get computed travel time data
travel_time = mpe.travel_time

# get phi (zero contour for given end_point)
phi = mpe.phi
```

(continues on next page)

(continued from previous page)

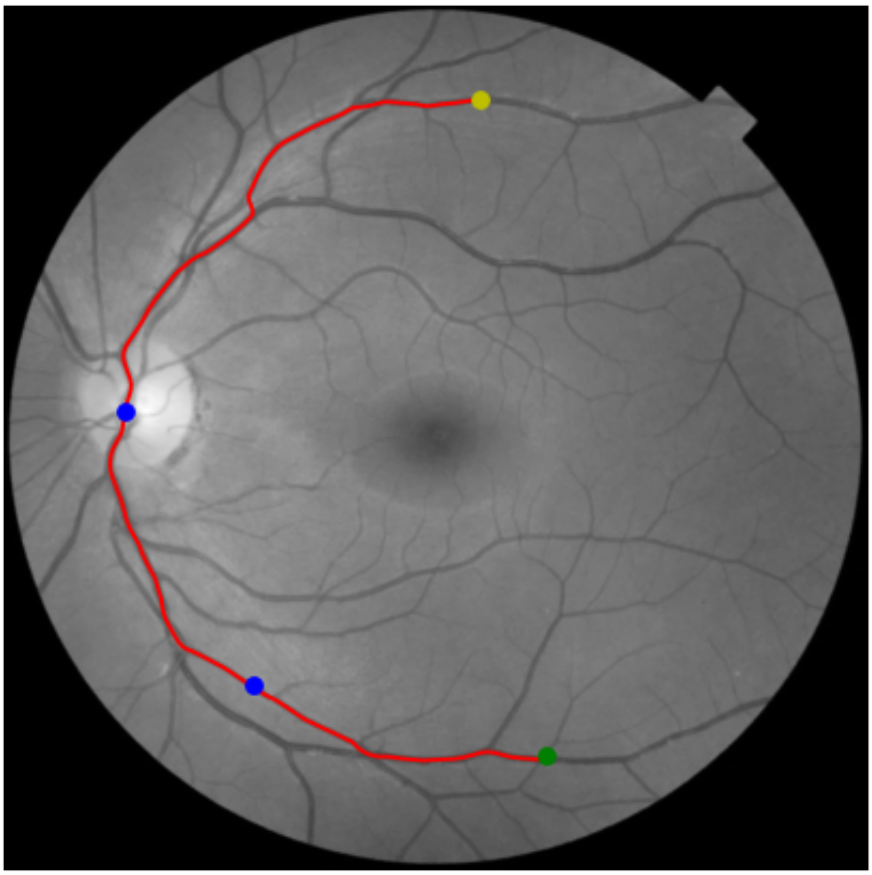
```
# extract path from start_point to end_point
# it returns PathExtractionResult instance
extracting_result = mpe(start_point)

# the list of path points
path_points = extracting_result.path_points

# the list of integrate time values in every path point
path_integrate_times = extracting_result.path_integrate_times

# the list of travel time values in every path point
path_travel_times = extracting_result.path_travel_times

# the number of ODE solver steps
step_count = extracting_result.step_count

# the number of right hand function evaluations in ODE solver
func_eval_count = extracting_result.func_eval_count
```

3.2 Examples

3.2.1 Retina Vessels

Extracting the minimal path through the retina vessels with additional way points.

```
from skimage.data import retina
from skimage.color import rgb2gray
from skimage.transform import rescale
from skimage.filters import sato

from skmpe import mpe

image = rescale(rgb2gray(retina()), 0.5)
speed_image = sato(image)

start_point = (76, 388)
end_point = (611, 442)
way_points = [(330, 98), (554, 203)]

path_info = mpe(speed_image, start_point, end_point, way_points)

px, py = path_info.path[:, 1], path_info.path[:, 0]

plt.imshow(image, cmap='gray')
plt.plot(px, py, '-r')
plt.plot(*start_point[::-1], 'oy')
plt.plot(*end_point[::-1], 'og')
for p in way_points:
    plt.plot(*p[::-1], 'ob')
plt.axis('off')
```



3.2.2 Bricks

Extracting the shortest paths through “bricks” image.

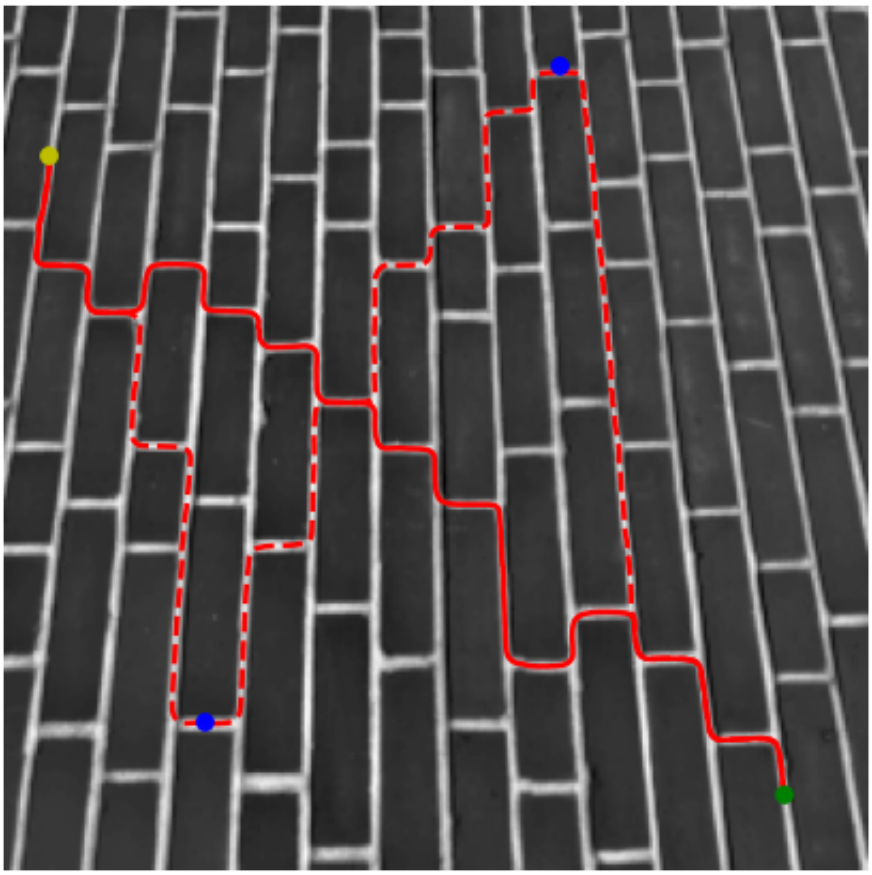
```
from skimage.data import brick
from skimage.transform import rescale
from skimage.exposure import rescale_intensity, adjust_sigmoid

from skmpe import parameters, mpe

image = rescale(brick(), 0.5)
speed_image = rescale_intensity(
    adjust_sigmoid(image, cutoff=0.5, gain=10).astype(np.float_), out_range=(0., 1.))

start_point = (44, 13)
end_point = (233, 230)
way_points = [(211, 59), (17, 164)]

with parameters(integrate_max_step=1.0):
    path_info1 = mpe(speed_image, start_point, end_point)
    path_info2 = mpe(speed_image, start_point, end_point, way_points)

px1, py1 = path_info1.path[:, 1], path_info1.path[:, 0]
px2, py2 = path_info2.path[:, 1], path_info2.path[:, 0]

plt.imshow(image, cmap='gray')
plt.plot(px1, py1, '-r', linewidth=2)
plt.plot(px2, py2, '--r', linewidth=2)

plt.plot(*start_point[::-1], 'oy')
plt.plot(*end_point[::-1], 'og')
for p in way_points:
    plt.plot(*p[::-1], 'ob')
plt.axis('off')
```

3.2.3 Maze

Finding the path in the maze.

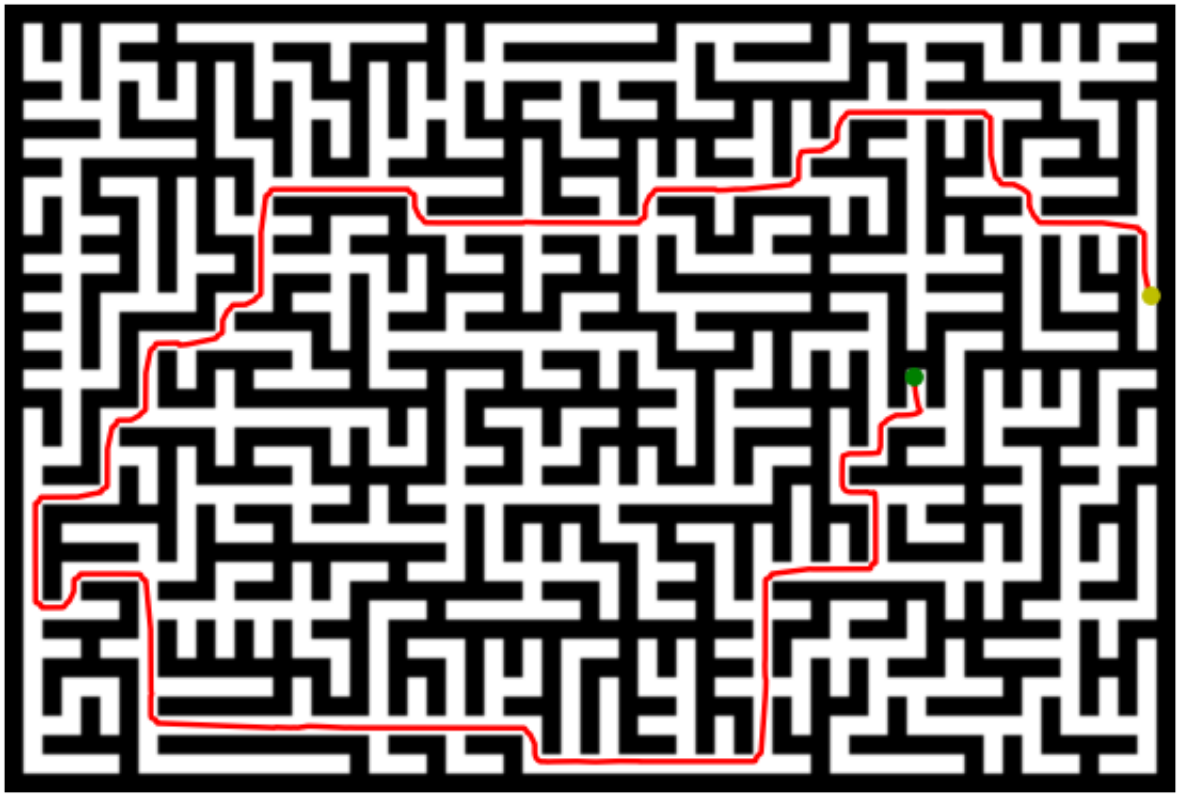
```
from skimage.io import imread
from skimage.exposure import rescale_intensity

from skmpe import parameters, mpe, OdeSolverMethod

image = imread('_static/maze.png', as_gray=True).astype(np.float_)
speed_image = rescale_intensity(image, out_range=(0.005, 1.0))

start_point = (60, 238)
end_point = (77, 189)

with parameters(ode_solver_method=OdeSolverMethod.LSODA, integrate_max_step=1.0):
    path_info = mpe(speed_image, start_point, end_point)

path = path_info.path

plt.imshow(image, cmap='gray')
```

(continues on next page)



(continued from previous page)

```
plt.plot(path[:, 1], path[:, 0], '-r', linewidth=2)

plt.plot(*start_point[::-1], 'oy')
plt.plot(*end_point[::-1], 'og')

plt.axis('off')
```



3.3 API Reference

3.3.1 API Summary

InitialInfo	Initial info data model
PathInfo	The named tuple with info about extracted path or piece of path
PathInfoResult	The named tuple with path info result
TravelTimeOrder	The enumeration of travel time computation orders
OdeSolverMethod	The enumeration of supported ODE solver methods
Parameters	MPE algorithm parameters model
parameters	Context manager for using specified parameters
default_parameters	Returns the default parameters
MPEError	Base exception class for all MPE errors

Continued on next page

Table 1 – continued from previous page

ComputeTravelTimeError	The exception occurs when computing travel time has failed
PathExtractionError	Base exception class for all extracting path errors
EndPointNotReachedError	The exception occurs when the ending point is not reached
PathExtractionResult	The named tuple with info about extracted path
MinimalPathExtractor	Minimal path extractor
mpe	Extracts a minimal path by start/end and optionally way points

3.3.2 Data and Models

class skmpe.**InitialInfo**

Initial info data model

speed_data
Speed data in numpy ndarray

start_point
The starting point

end_point
The ending point

way_points
The tuple of way points

all_points(*self*) → List[Sequence[int]]
Returns all initial points

point_intervals(*self*) → List[Tuple[Sequence[int], Sequence[int]]]
Returns the list of the tuples of initial point intervals

class skmpe.**PathInfo**

Bases: tuple

The named tuple with info about extracted path or piece of path

path
The path in numpy ndarray

start_point
The starting point

end_point
The ending point

travel_time
The travel time numpy ndarray

extraction_result
The path extraction result in *PathExtractionResult* that is returned by *MinimalPathExtractor*

reversed
The flag is true if the extracted path is reversed

class skmpe.**PathInfoResult**
Bases: tuple

The named tuple with path info result

path
Path data in numpy array

pieces
The tuple of *PathInfo* for every path piece

3.3.3 Parameters

class skmpe.**TravelTimeOrder**
Bases: enum.IntEnum

The enumeration of travel time computation orders

Orders:

- **first** – the first ordered travel time computation
- **second** – the second ordered travel time computation

first = 1

second = 2

class skmpe.**OdeSolverMethod**
Bases: str, enum.Enum

The enumeration of supported ODE solver methods

BDF = 'BDF'

DOP853 = 'DOP853'

LSODA = 'LSODA'

RK23 = 'RK23'

RK45 = 'RK45'

Radau = 'Radau'

class skmpe.**Parameters**
MPE algorithm parameters model

travel_time_spacing
The travel time computation spacing

default: 1.0

travel_time_order
The travel time computation order

default: TravelTimeOrder.first

travel_time_cache
Use or not travel time computation cache for extracting paths with way points

default: True

ode_solver_method
ODE solver method

default: 'RK45'

integrate_time_bound
Integration time bound

default: 10000

integrate_min_step
Integration minimum step

default: 0.0

integrate_max_step
Integration maximum step

default: 4.0

dist_tol
Distance tolerance for control path evolution

default: 1e-03

max_small_dist_steps
The max number of small distance steps while path evolution

default: 100

dist_tol = None

integrate_max_step = None

integrate_min_step = None

integrate_time_bound = None

max_small_dist_steps = None

ode_solver_method = None

travel_time_cache = None

travel_time_order = None

travel_time_spacing = None

skmpe.**parameters**(***kwargs*)

Context manager for using specified parameters

Parameters

kwargs [mapping] The parameters

Examples

```
>>> from skmpe import parameters
>>> with parameters(integrate_time_bound=200000) as params:
>>>     print(params.__repr__())

Parameters(
    travel_time_spacing=1.0,
    travel_time_order=<TravelTimeOrder.first: 1>,
    travel_time_cache=False,
    ode_solver_method=<OdeSolverMethod.RK45: 'RK45'>,
    integrate_time_bound=200000.0,
    integrate_min_step=0.0,
    integrate_max_step=4.0,
    dist_tol=0.001,
    max_small_dist_steps=100
)
```

```
from skmpe import parameters, mpe

...

with parameters(integrate_time_bound=200000):
    path_result = mpe(start_point, end_point)
```

skmpe.**default_parameters**() → skmpe.Parameters

Returns the default parameters

Returns

parameters [Parameters] Default parameters

Examples

```
>>> from skmpe import default_parameters
>>> print(default_parameters().__repr__())

Parameters(
    travel_time_spacing=1.0,
    travel_time_order=<TravelTimeOrder.first: 1>,
    travel_time_cache=False,
    ode_solver_method=<OdeSolverMethod.RK45: 'RK45'>,
    integrate_time_bound=10000.0,
    integrate_min_step=0.0,
    integrate_max_step=4.0,
    dist_tol=0.001,
    max_small_dist_steps=100
)
```

3.3.4 Exceptions

class skmpe.**MPEError**
Bases: Exception

Base exception class for all MPE errors

class skmpe.**ComputeTravelTimeError**
Bases: *skmpe.MPEError*

The exception occurs when computing travel time has failed

class skmpe.**PathExtractionError**(**args, travel_time: numpy.ndarray, start_point: Sequence[int], end_point: Sequence[int]*)
Bases: *skmpe.MPEError*

Base exception class for all extracting path errors

property end_point
Ending point

property start_point
Starting point

property travel_time
Computed travel time data

class skmpe.**EndPointNotReachedError**(**args, travel_time: numpy.ndarray, start_point: Sequence[int], end_point: Sequence[int], extracted_points: List[Sequence[float]], last_distance: float, reason: str*)
Bases: *skmpe.PathExtractionError*

The exception occurs when the ending point is not reached

property end_point
Ending point

property extracted_points
The list of extracted path points

property last_distance
The last distance to the ending point from the last path point

property reason
The reason of extracting path termination

property start_point
Starting point

property travel_time
Computed travel time data

3.3.5 Path Extraction

class skmpe.**PathExtractionResult**
Bases: tuple

The named tuple with info about extracted path

Notes

The instance of the class is returned from *MinimalPathExtractor.__call__()*.

path_points
The extracted path points in the list

path_integrate_times
The list of integrate times for every path point

path_travel_times
The list of travel time values for every path point

step_count
The number of integration steps

func_eval_count
The number of evaluations of the right hand function

class skmpe.**MinimalPathExtractor**(*speed_data: numpy.ndarray, end_point: Sequence[int], parameters: Optional[skmpe.Parameters] = None*)
Minimal path extractor

Minimal path extractor based on the fast marching method and ODE solver.

Parameters

speed_data [np.ndarray] The speed data (n-d numpy array)

end_point [Sequence[int]] The ending point (a.k.a. "source point")

parameters [class:*Parameters*] The parameters

Raises

ComputeTravelTimeError [Computing travel time has failed]

Examples

```
from skmpe import MinimalPathExtractor

# some function for computing speed data
speed_data_2d = compute_speed_data_2d()

mpe = MinimalPathExtractor(speed_data_2d, end_point=(10, 25))
path = mpe((123, 34))
```

__call__(*self, start_point: Sequence[int]*) → skmpe.PathExtractionResult
Extract path from start point to source point (ending point)

Parameters

start_point [Sequence[int]] The starting point

Returns

path_extraction_result [*PathExtractionResult*] The path extraction result

Raises

PathExtractionError [Extracting path has failed]

EndPointNotReachedError [The extracted path is not reached the ending point]

property parameters
Returns the parameters

property phi
Returns the computed phi (zero contour) for given source point

property travel_time
Returns the computed travel time for given speed data

skmpe.**mpe**(**args*, ***kwargs*) → skmpe.PathInfoResult
Extracts a minimal path by start/end and optionally way points

The function is high level API for extracting paths.

Parameters

init_info [*InitialInfo*] (sign 1) The initial info

start_point [Sequence[int]] (sign 2) The starting point

end_point [Sequence[int]] (sign 2) The ending point

way_points [Sequence[Sequence[int]]] (sign 2) The way points

parameters [*Parameters*] The parameters

Returns

path_info [*PathInfoResult*] Extracted path info

See also:

InitialInfo, ***Parameters***, ***MinimalPathExtractor***

Notes

There are two signatures of *mpe* function.

Use *InitialInfo* for init data:

```
mpe(init_info: InitialInfo, *,
    parameters: Optional[Parameters] = None) -> ResultPathInfo
```

Set init data directly:

```
mpe(speed_data: np.ndarray, *,
    start_point: Sequence[int],
    end_point: Sequence[int],
    way_points: Sequence[Sequence[int]] = (),
    parameters: Optional[Parameters] = None) -> ResultPathInfo
```

3.4 Changelog

3.4.1 v0.2.4 (23.05.2021)

- Update scikit-fmm dependency version #9

3.4.2 v0.2.3 (14.04.2021)

- Fixed #1

3.4.3 v0.2.2 (28.05.2020)

- Update documentation: add maze example and low-level api section

3.4.4 v0.2.1 (27.05.2020)

- Fix building docs on ReadTheDocs

3.4.5 v0.2.0 (27.05.2020)

- Refactoring the package with changing some low-level API
- Add documentation

3.4.6 v0.1.1

- Fix links
- Update readme

3.4.7 v0.1.0

- Initial release

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

D

E

F

I

L

M

O

P

R

S

T

W