difw

Release 0.0.31

Iñigo Martinez

Aug 09, 2023

CONTENTS

Fig. 1: Finite-dimensional spaces of simple, fast, and highly-expressive diffeomorphisms derived from parametric, continuously-defined, velocity fields in Numpy and Pytorch

CHAPTER

ONE

QUICK START

difw is a fast and open source library to compute fast and highly-expressive diffeomorphisms derived from parametric, continuously-defined, velocity fields in Numpy and Pytorch.

This documentation contains a *user-guide* (including *installation* procedure and *basic usage* of the library), an *API Reference*, as well as a detailed example. **difw** is released under the MIT License.

Finally, if you use **difw** in a scientific publication, we would appreciate *citations*.

1.1 Getting Started

The following code transforms a regular grid using a diffeomorphic curve parametrized with θ:

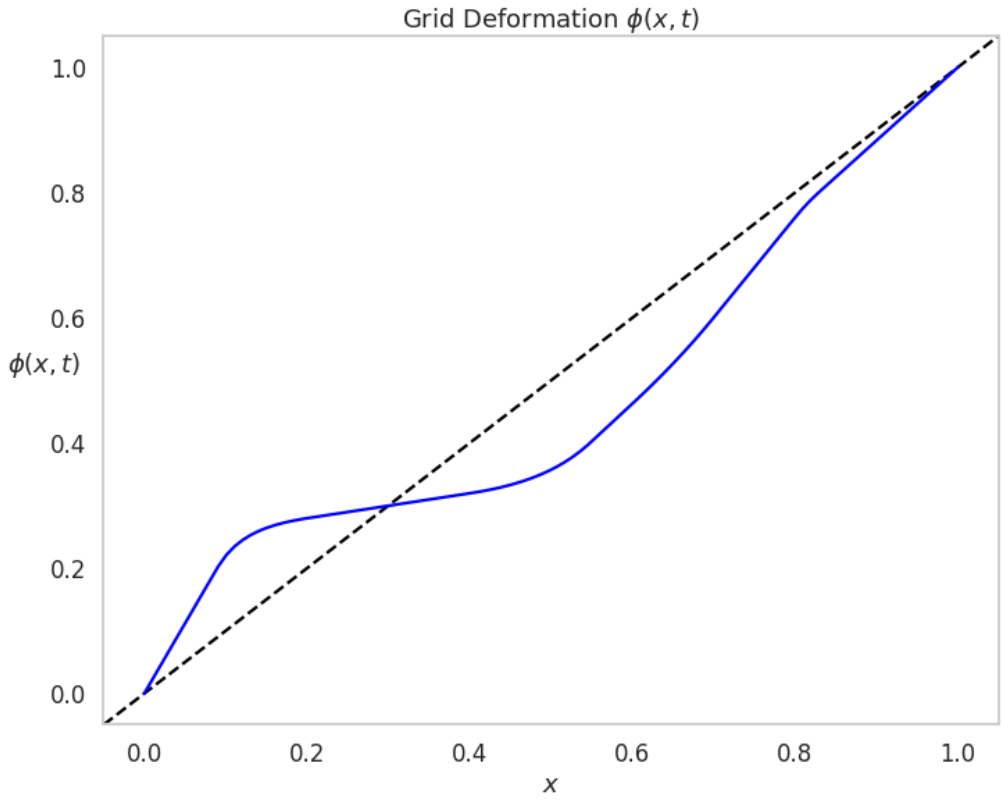
```
# Import difw library
import difw

# Transformation instance
T = difw.Cpab(tess_size=5, backend="numpy", device="cpu", zero_boundary=True, basis="qr")

# Generate grid
grid = T.uniform_meshgrid(100)

# Transformation parameters
theta = T.identity(epsilon=1)

# Transform grid
grid_t = T.transform_grid(grid, theta)
```

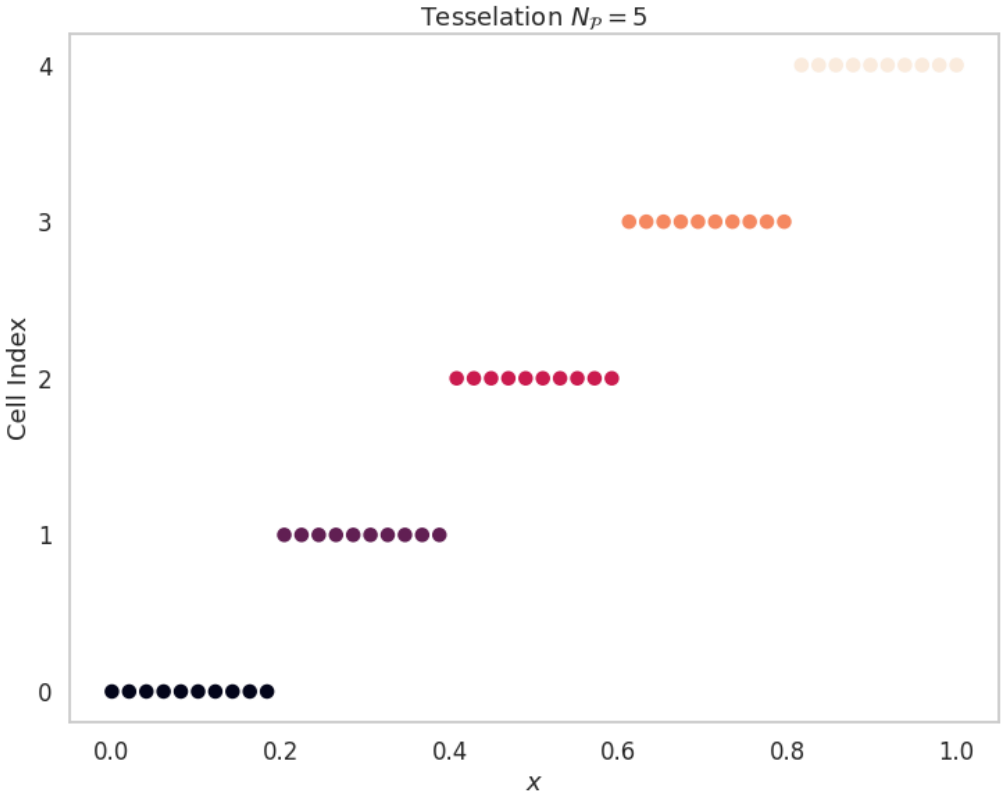
In this example, the tesselation is composed of 5 intervals, and the `zero_boundary` condition set to `True` constraints the velocity at the tesselation boundary (in this case, at `x=0` and `x=1`). The regular grid has 100 equally spaced points.

```
T.visualize_tesselation()
```

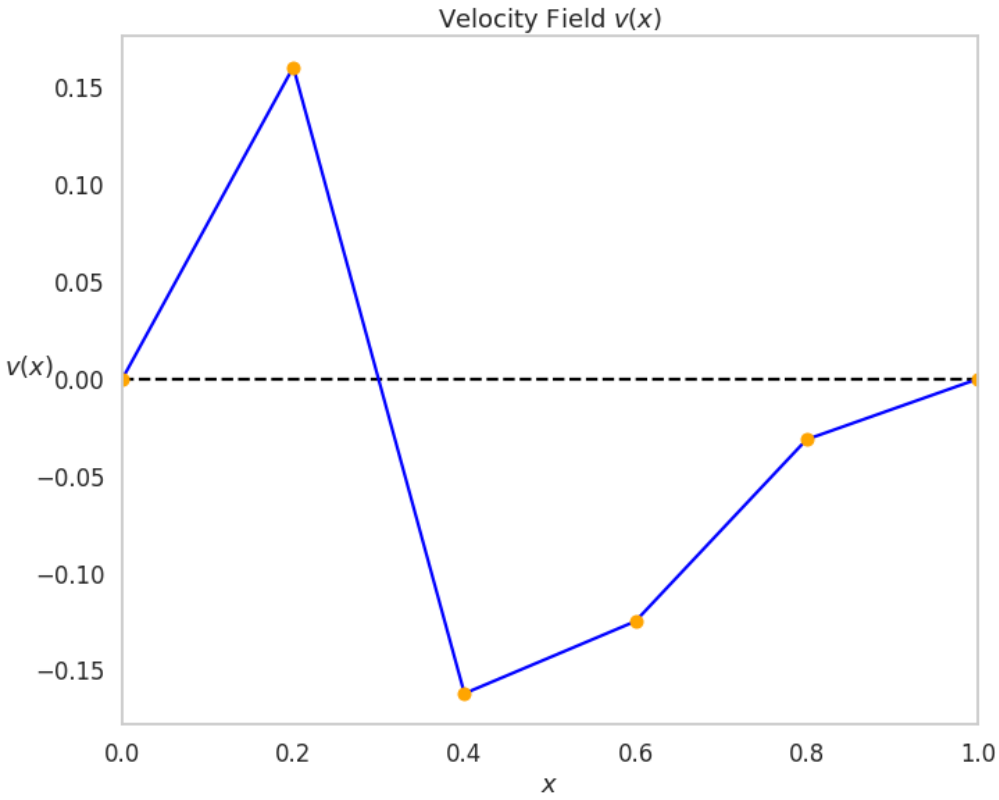
The velocity field is formed by a continuous piecewise affine function defined over 5 intervals. The parameters θ represent a basis of the null space for all continuous piecewise affine functions composed of 5 intervals. In this case, we have used the QR decomposition to build the basis. See the *API documentation* for more details about the transformation options.

Taking into account the zero velocity constraints at the boundary, only 4 dimensions or degree of freedom are left to play with, and that indeed is the dimensionality of θ, a vector of 4 values.


Grid Deformation $\phi(x, t)$
1.0
0.8
0.6
$\phi(x, t)$
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
x


Tesselation $N_p = 5$
4
3
2
1
0
Cell Index
0.0
0.2
0.4
0.6
0.8
1.0
x

```
T.visualize_velocity(theta)
```



We can visualize the generated transformation based on the parameters θ:

```
T.visualize_deformgrid(theta)
```

In addition, for optimization tasks, it is useful to obtain the gradient of the transformation with respect to parameters θ. The gradient function can be obtained in closed-form solution. There are 4 different functions, one per dimension in θ:

```
T.visualize_gradient(theta)
```

1.2 Installation

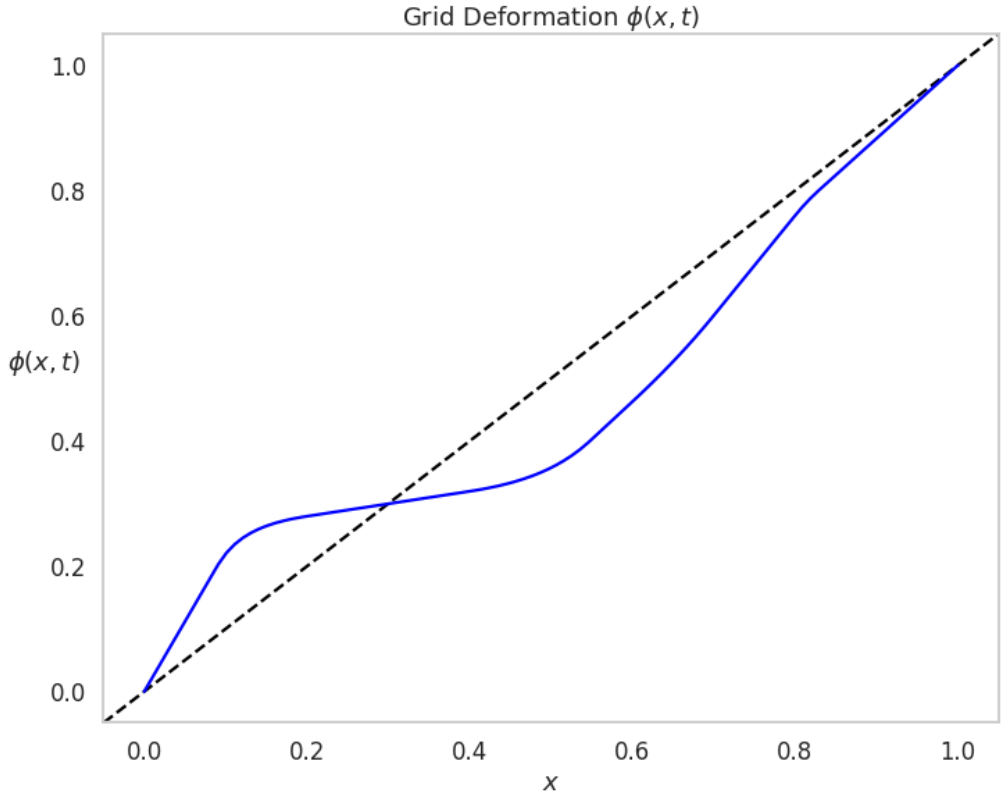
As the compiled **difw** package is hosted on the Python Package Index (PyPI) you can easily install it with `pip`. To install **difw**, run this command in your terminal of choice:

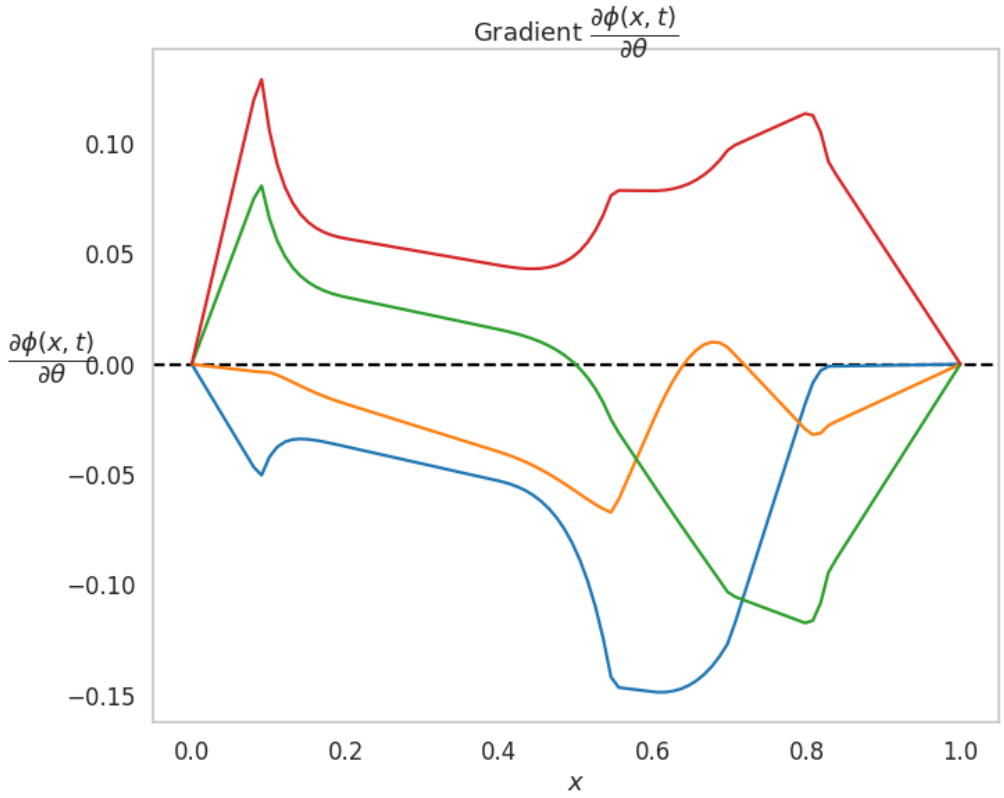
```
$ pip install difw
```

or, alternatively:

```
$ python -m pip install difw
```

If you want to get **difw**'s latest version, you can refer to the repository hosted at github:


Grid Deformation $\phi(x, t)$
1.0
0.8
0.6
$\phi(x, t)$
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
x


Gradient $\frac{\partial \phi(x, t)}{\partial \theta}$
0.10
0.05
$\frac{\partial \phi(x, t)}{\partial \theta}$
0.00
−0.05
−0.10
−0.15
0.0
0.2
0.4
0.6
0.8
1.0
x

```
python -m pip install https://github.com/imartinezl/difw/archive/master.zip
```

1.3 Environment Setup

1.3.1 Requirements

difw builds on `numpy`, `torch`, `scipy`, `ninja`, and `matplotlib` libraries.

1.3.2 Python 3

To find out which version of `python` you have, open a terminal window and try the following command:

```
$ python3 --version
Python 3.6.9
```

If you have `python3` on your machine, then this command should respond with a version number. If you do not have `python3` installed, follow these instructions.

1.3.3 Pip

`pip` is the reference Python package manager. It's used to install and update packages. In case `pip` is not installed in your OS, follow these procedure.

1.3.4 Virtual Environment

`venv` creates a "virtual" isolated Python installation and installs packages into that virtual installation. It is always recommended to use a virtual environment while developing Python applications. To create a virtual environment, go to your project's directory and run venv.

```
$ python3 -m venv env
```

Before you can start installing or using packages in your virtual environment you'll need to activate it.

```
$ source env/bin/activate
```

1.4 Source Code

difw is developed on GitHub, where the code is always available.

You can either clone the public repository:

```
$ git clone git://github.com/imartinezl/difw.git
```

Or, download the tarball:

```
$ curl -OL https://github.com/imartinezl/difw/tarball/main
# optionally, zipball is also available (for Windows users).
```

Once you have a copy of the source, you can embed it in your own Python package, or install it into your site-packages easily:

```
$ cd difw
$ python -m pip install .
```

CHAPTER

TWO

USER GUIDE

2.1 Introduction

The DIFW library allows to create transformations $\phi(x,t)$ based on the integration of a continuous piecewise affine velocity field $v(x)$. Let us bring some clarity to this sentence by including some definitions:

- The transformation $\phi(x,t)$ is created by the integration of a velocity field. For that, we need to solve a differential equation of the form:

$$\frac{\partial \phi(x,t)}{\partial t} = v(\phi(x))$$

The transformation $\phi(x,t)$ depend on two variables x (spatial dimension) and t (integration time).

- The velocity field $v(x)$ can be a function of any form and shape, but in this library we focus on an specific type of functions, which are continuous piecewise affine functions.
- Continous function: there are no discontinuities in the function domain
- Piecewise function: is a function that is defined by parts
- Affine: is a geometric transformation that consist on a linear transformation + a translation.

Thus, a continous, piecewise, and affine function is just a set of lines joined together. In summary, in this library integrate (efficiently) these functions to create diffeomorphic transformations $\phi(x,t)$ that are very useful for a lot of tasks in machine learning.

2.2 Loading libraries

First, we need to import the necessary Python libraries: `difw` library to compute the transformations, `matplotlib` for data visualization, `numpy` for array manipulation and `pytorch` for autodifferentiation and gradient descent optimization.

[1]:
```
import numpy as np
import torch
import matplotlib.pyplot as plt
import difw

plt.rcParams["figure.figsize"] = (10, 7)
```

2.3 Transformation parameters

In order to create a transformation $\phi(x, t)$, several options need to be specified. CPAB transformations are built by integrating a continuous piecewise affine velocity field $v(x)$. Such velocity field is defined onto a regular grid, or tesselation. In this example, we will set the number of intervals to 5 (`tess_size=5`).

The `backend` option let us choose between `numpy` backend and the `pytorch` backend, the preferred option for optimization tasks. These computations can be also executed on CPU or GPU `device` (for the `pytorch` backend).

The `zero_boundary` condition set to `True` constraints the velocity $v(x)$ at the tesselation boundary to 0, so $v(0) = 0$ and $v(1) = 0$.

The `basis` option let us choose between {`svd`, `sparse`, `rref`, `qr`}, and it represents the method to obtain the null space representation for continuous piecewise affine functions with `tess_size` intervals. In this case, we have used the QR decomposition to build the basis.

[2]:
```
tess_size = 5
backend = "numpy" # ["pytorch", "numpy"]
device = "cpu" # ["cpu", "gpu"]
zero_boundary = True # [True, False]
basis = "qr" # ["svd", "sparse", "rref", "qr"]

T = difw.Cpab(tess_size, backend, device, zero_boundary, basis)
```

2.4 Transformation example

Then, we need to create the one-dimensional grid that is going to be transformed. For that, we use the `uniform_meshgrid` method, and we set the number of equally spaced points in the grid to 100.

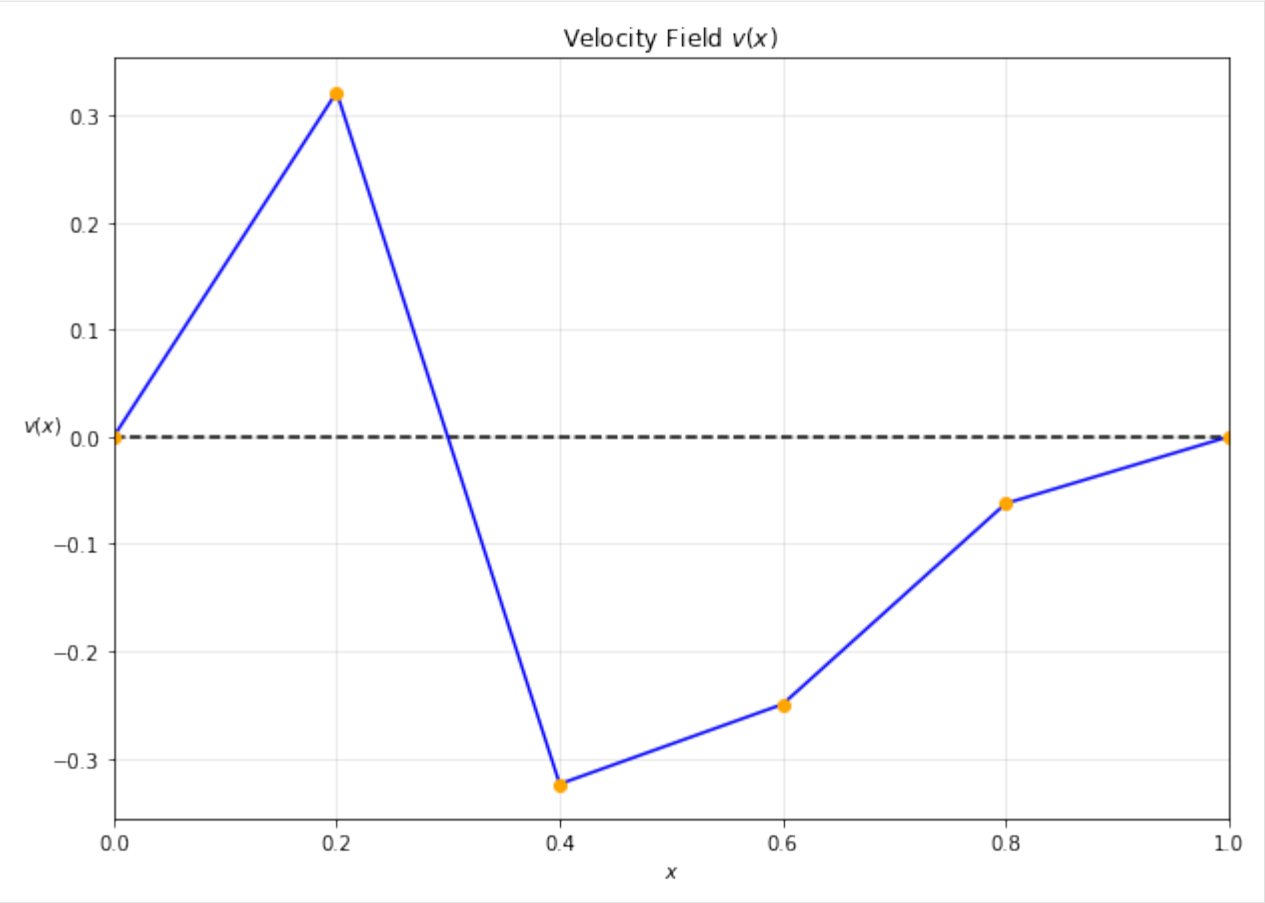
The velocity field $v(x)$ in CPAB transformations are parameterized by a vector θ. In this example, taking into account the zero velocity constraints at the boundary, only 4 dimensions or degree of freedom are left to play with, and that indeed is the dimensionality of θ, a vector of 4 values.

Finally, we can pass the `grid` and the `theta` parameters to the `transform_grid` method and compute the transformed grid `grid_t` $\phi(x)$.

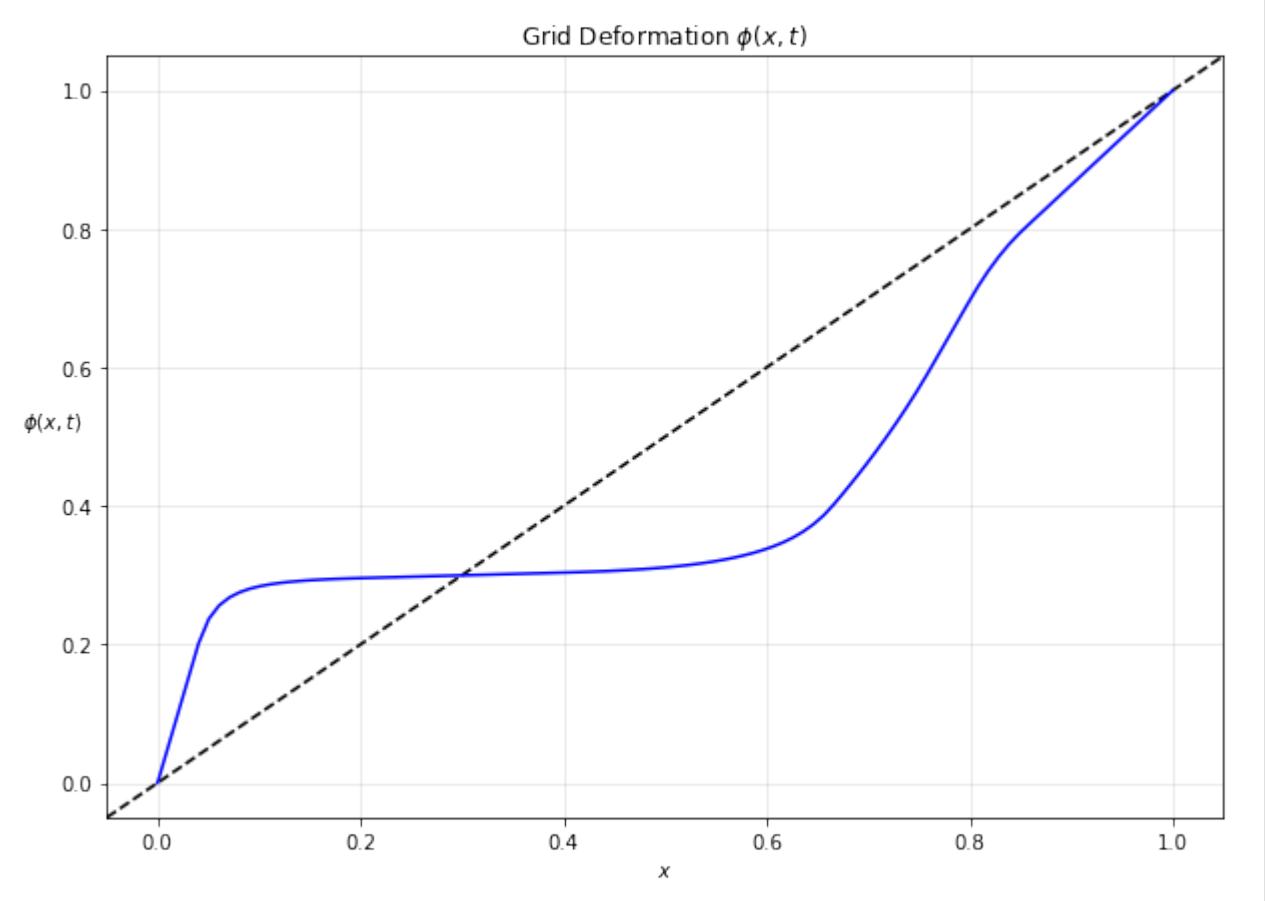
[3]:
```
outsize = 100
grid = T.uniform_meshgrid(outsize)

batch_size = 1
theta = T.identity(batch_size, epsilon=2)

grid_t = T.transform_grid(grid, theta)
```

We can use the methods `visualize_velocity` and `visualize_deformgrid` to plot the velocity field $v(x)$ and the transformed grid $\phi(x, t)$ respectively.

[4]:
```
T.visualize_velocity(theta);
```



[5]:
```
T.visualize_deformgrid(theta);
```



The dotted black line represents the identity tranformation $\phi(x, t) = x$.

2.5 Integration details

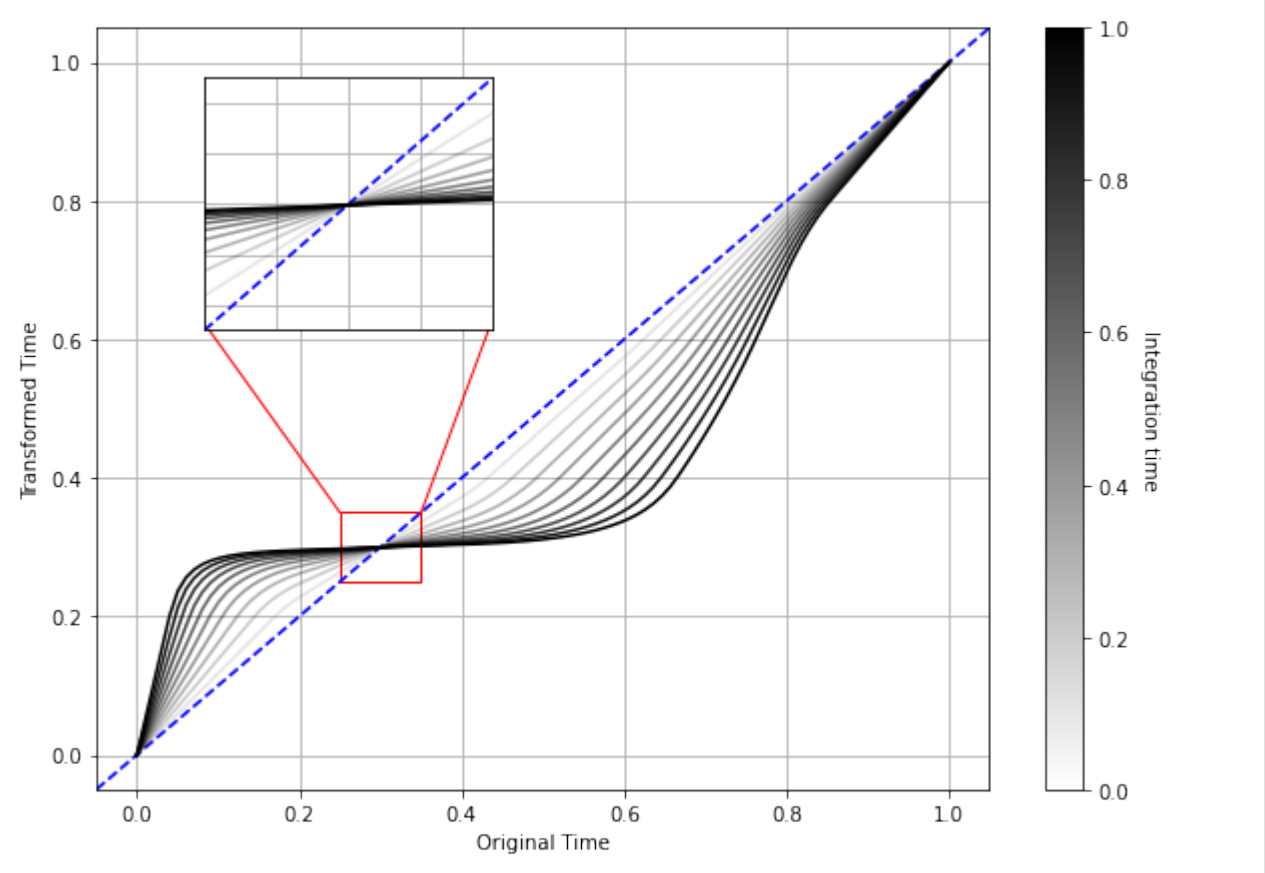
By default, the velocity field is integrated up to $t == 1$. The following figure shows the how the transformed grid changes along the integration time t.

[6]:
```
grid = T.uniform_meshgrid(outsize)
theta = T.identity(batch_size, epsilon=2)

fig, ax = plt.subplots()
ax_zoom = fig.add_axes([0.2,0.58,0.2,0.25])

ax.axline((0,0),(1,1), color="blue", ls="dashed")
ax_zoom.axline((0,0),(1,1), color="blue", ls="dashed")

N = 11
for i in range(N):
    time = i / (N-1)
    grid_t = T.transform_grid(grid, theta, time=time)
    ax.plot(grid, grid_t.T, label=round(time, 2), color="black", alpha=time)
    ax_zoom.plot(grid, grid_t.T, label=round(time, 2), color="black", alpha=time)
```

(continues on next page)

(continued from previous page)

```
ax.grid()
ax.set_xlabel("Original Time")
ax.set_ylabel("Transformed Time")

sm = plt.cm.ScalarMappable(cmap="gray_r")
cbar = plt.colorbar(sm, ax=ax)
cbar.ax.get_yaxis().labelpad = 15
cbar.ax.set_ylabel('Integration time', rotation=270)

ax_zoom.grid()
ax_zoom.set_xlim(.25, .35)
ax_zoom.set_ylim(.25, .35)
ax_zoom.set_xticklabels([])
ax_zoom.set_yticklabels([])
ax_zoom.xaxis.set_ticks_position('none')
ax_zoom.yaxis.set_ticks_position('none')

from matplotlib.patches import Rectangle
import matplotlib.lines as lines
r = Rectangle((.25,.25), 0.1, 0.1, edgecolor="red", facecolor="none", lw=1)
ax.add_patch(r)

line = lines.Line2D([0.085,0.25], [0.62, 0.35], color="red", lw=1)
ax.add_line(line)
line = lines.Line2D([0.435,0.35], [0.62, 0.35], color="red", lw=1)
ax.add_line(line);
```



2.6 Scaling and squaring

The DIFW library allows to use the scaling and squaring method to approximate the velocity field integration. This method uses the following property of diffeomorphic transformations to accelerate the computation of the integral:

$$\phi(x, t+s) = \phi(x, t) \circ \phi(x, s)$$

Thus, computing the transformation ϕ at time $t+s$ is equivalent to composing the transformations at time t and s. In the scaling and squaring method, we impose $t = s$, so that we need to compute only one transformation and self-compose it:

$$\phi(x, 2t) = \phi(x, t) \circ \phi(x, t)$$

Repeating this procedure multiple times (N), we can efficienty approximate the integration:

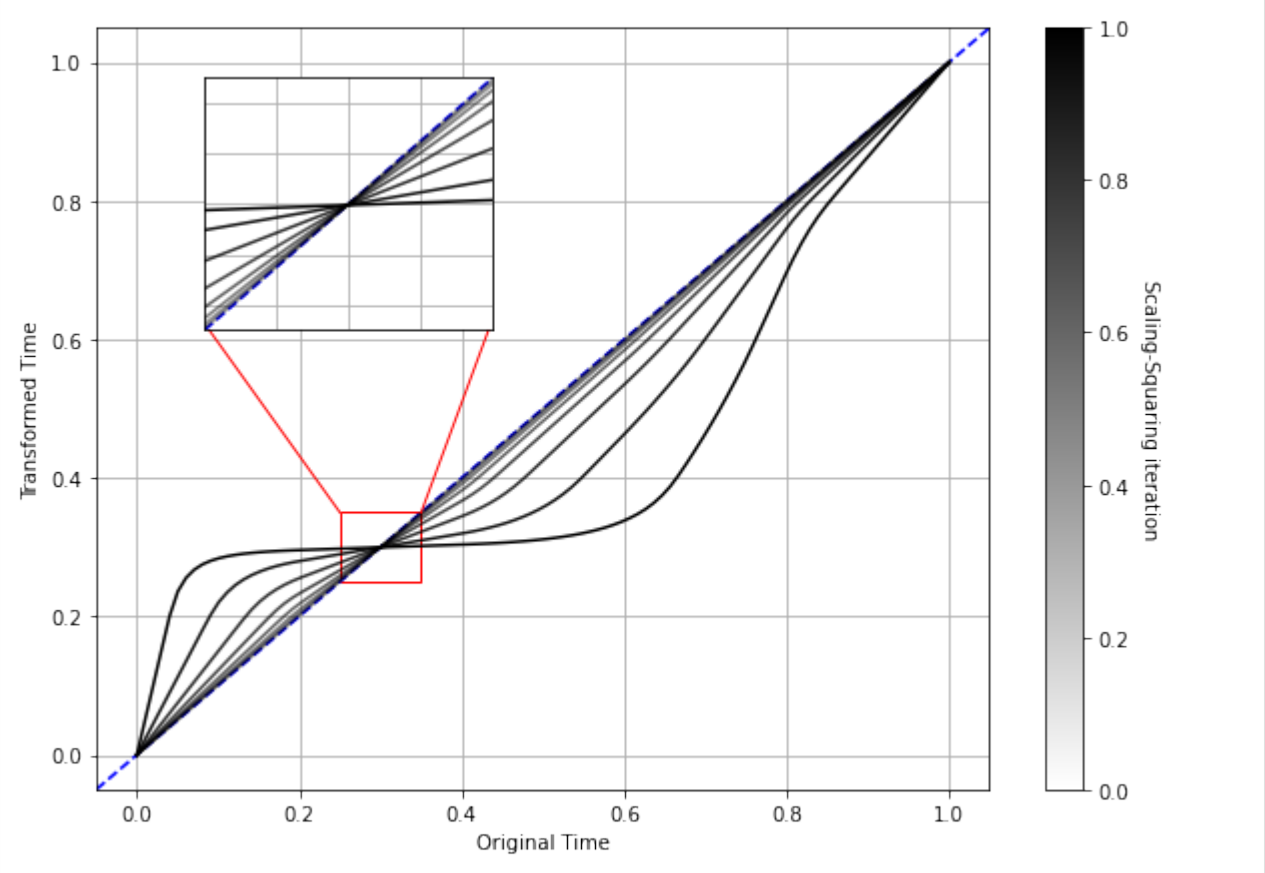
$$\phi(x, t^{2N}) = \phi(x, t) \underbrace{\circ \cdots \circ}_{N} \phi(x, t)$$

[7]:
```
grid = T.uniform_meshgrid(outsize)
theta = T.identity(batch_size, epsilon=2)

fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
ax_zoom = fig.add_axes([0.2,0.58,0.2,0.25])

ax.axline((0,0),(1,1), color="blue", ls="dashed")
ax_zoom.axline((0,0),(1,1), color="blue", ls="dashed")

N = 11
for i in range(N):
    alpha = i / (N-1)
    grid_t = T.transform_grid_ss(grid, theta / 2**N, N=i+1)
    ax.plot(grid, grid_t.T, label=round(time, 2), color="black", alpha=alpha)
    ax_zoom.plot(grid, grid_t.T, label=round(time, 2), color="black", alpha=alpha)

ax.grid()
ax.set_xlabel("Original Time")
ax.set_ylabel("Transformed Time")

sm = plt.cm.ScalarMappable(cmap="gray_r")
cbar = plt.colorbar(sm, ax=ax)
cbar.ax.get_yaxis().labelpad = 15
cbar.ax.set_ylabel('Scaling-Squaring iteration', rotation=270)

ax_zoom.grid()
ax_zoom.set_xlim(.25, .35)
ax_zoom.set_ylim(.25, .35)
ax_zoom.set_xticklabels([])
ax_zoom.set_yticklabels([])
ax_zoom.xaxis.set_ticks_position('none')
ax_zoom.yaxis.set_ticks_position('none')

from matplotlib.patches import Rectangle
import matplotlib.lines as lines
r = Rectangle((.25,.25), 0.1, 0.1, edgecolor="red", facecolor="none", lw=1)
ax.add_patch(r)

line = lines.Line2D([0.085,0.25], [0.62, 0.35], color="red", lw=1)
ax.add_line(line)
line = lines.Line2D([0.435,0.35], [0.62, 0.35], color="red", lw=1)
ax.add_line(line);
```



2.7 Data transformation

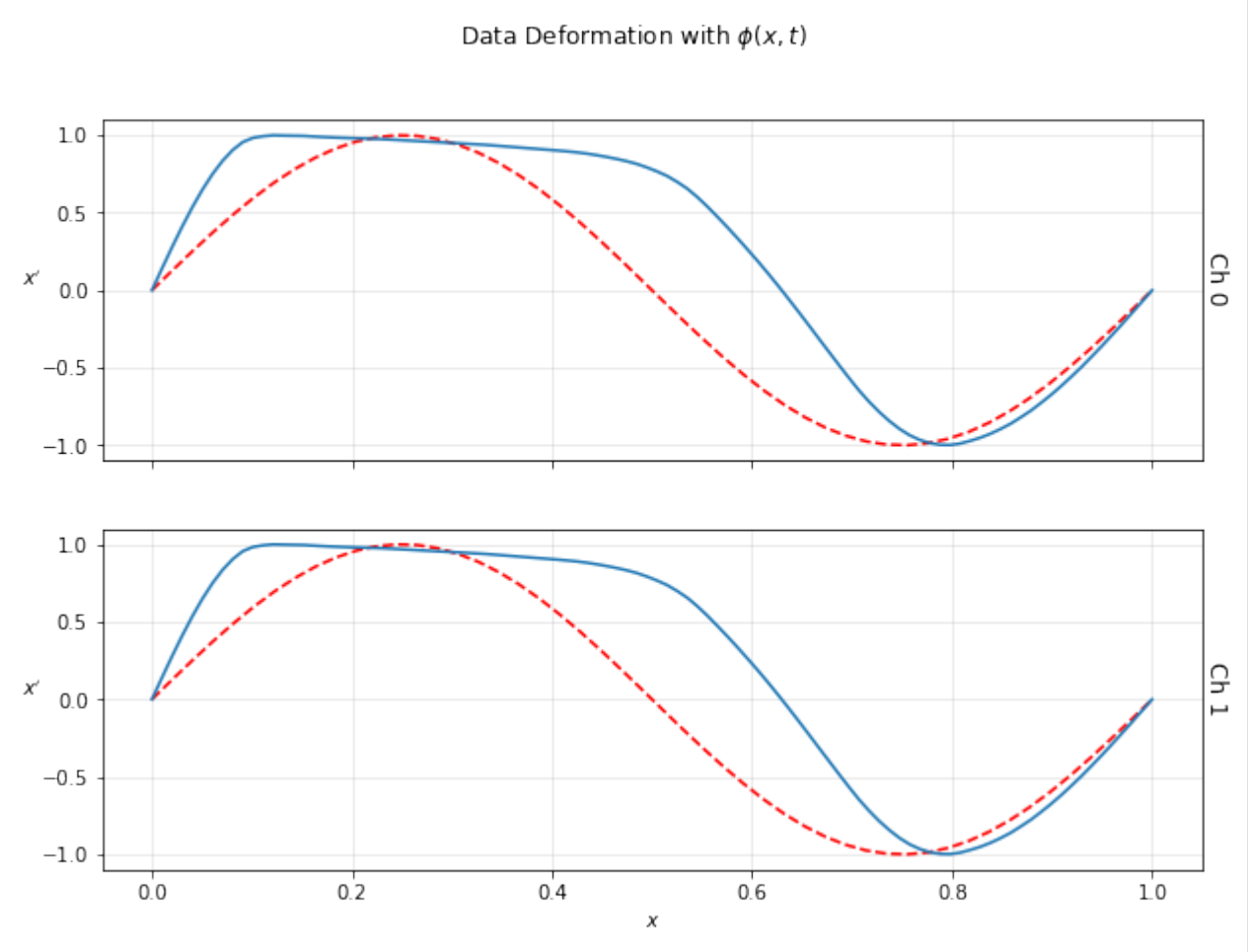
The time series data must have a shape (batch, length, channels). In this example, we have created a sinusoidal dataset of one batch, 50 points in length, and 2 channels. Then, to transform time series data, we can use the `transform_data` method and pass as arguments:

- data: n-dimensional array of shape (batch, length, channels)
- theta: transformation parameters
- outsize: length of the transformed data, with final shape (batch, outsize, channels)

[8]:
```
batch_size = 1
length = 50
channels = 2
outsize = 100

# Generation
m = np.ones((batch_size, channels))
x = np.linspace(m*0, m*2*np.pi, length, axis=1)
data = np.sin(x)

theta = T.identity(batch_size, epsilon=1)
data_t = T.transform_data(data, theta, outsize)
```

And we can visualize this data transformation with the `visualize_deformdata` method. The red curves represent the original data and the blue ones are the transformed data after applying the transformation.

[9]:
```
T.visualize_deformdata(data, theta);
```



CHAPTER

THREE

API REFERENCE

This part of the documentation covers all the interfaces of `difw`. For parts where `difw` depends on external libraries, we document the most important right here and provide links to the canonical documentation.

3.1 Core

class Cpab(*tess_size: int*, *backend: str = 'numpy'*, *device: str = 'cpu'*, *zero_boundary: bool = True*, *basis: str = 'rref'*)

Continous piecewise-affine based transformations. Package main class.

Parameters

- **tess_size** (*int*) – tesselation size, with the number of vertices.
- **backend** (*str, optional*) – computational backend to use. Choose between "*numpy*", or "*pytorch*". Defaults to "*numpy*".
- **device** (*str, optional*) – device to use. Choose between "*cpu*" or "*gpu*". For the numpy backend only the "*cpu*" option is valid. Defaults to "*cpu*".
- **zero_boundary** (*bool, optional*) – determines if the velocity at the boundary is zero
- **basis** (*str, optional*) – constrain basis to use. Choose between "*rref*", "*svd*", or "*sparse*". Defaults to "*rref*".

params: Parameters

tess: *Tessellation*

backend_name: str

backend: module

uniform_meshgrid(*n_points: int*)

Generates a uniformly separated, one-dimensional meshgrid

Parameters
n_points (*int*) – number of points

Returns
vector of points

Return type
numpy.ndarray or torch.Tensor

covariance_cpa(*length_scale: float = 0.1*, *output_variance: float = 1*)

Generates the covariance matrix for a continuous piecewise-affine space. This function uses the squared exponential kernel with parameters λ_s: *length_scale* and λ_v: *output_variance*

Parameters

- **length_scale** (*float,>0*) – how fast the covariance declines between the cells
- **output_variance** (*float,>0*) – overall variance from the mean

Returns
[d, d] matrix.

Return type
numpy.ndarray or torch.Tensor

sample_transformation(*n_sample: int*, *mean=None*, *cov=None*)

Generates samples of a random transformation vector. By default the method samples from a standard Gaussian distribution.

Parameters

- **n_sample** (*int*) – number of transformations to sample
- **mean** (*numpy.ndarray or torch.Tensor*) – [d,] vector, mean of multivariate gaussian
- **cov** (*numpy.ndarray or torch.Tensor*) – [d,d] matrix, covariance of multivariate gaussian

Returns
[n_sample, d] matrix, where each row is a independent sample from a multivariate gaussian

Return type
numpy.ndarray or torch.Tensor

sample_transformation_with_prior(*n_sample: int*, *mean=None*, *length_scale: float = 0.1*, *output_variance: float = 1*)

Generates samples of smooth transitions. Covariance matrix for the continuous piecewise-affine space. This function uses the squared exponential kernel with parameters λ_s: *length_scale* and λ_v: *output_variance*

Parameters

- **n_sample** (*int*) – number of transformations to sample
- **mean** (*numpy.ndarray or torch.Tensor*) – [d,] vector, mean of multivariate gaussian
- **length_scale** (*float,>0*) – how fast the covariance declines between the cells
- **output_variance** (*float,>0*) – overall variance from the mean

Returns
[d, d] matrix.

Return type
numpy.ndarray or torch.Tensor

identity(*n_sample: int = 1*, *epsilon: float = 0*)

Generates the parameters for the identity transformation (vector of zeros)

Parameters

- **n_sample** (*int*) – number of transformations to sample
- **epsilon** (*float, >0*) – number to add to the identity transformation for stability

Returns
[n_sample, d] matrix, where each row is a sample

Return type
numpy.ndarray or torch.Tensor

transform_grid(*grid*, *theta*, *method=None*, *time=1.0*)

Integrates a grid using the parametrization in theta.

Parameters

- **grid** (*numpy.ndarray or torch.Tensor*) – [n_points] vector or [n_batch, n_points] tensor i.e. either a single grid for all theta values, or a grid for each theta value
- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix
- **method** (*str*) – integration method, one of "*closed_form*", "*numeric*".
- **time** (*float*) – integration time. Defaults to 1.0

Returns
[n_batch, n_points] tensor, with the transformed grid. The slice transformed_grid[i] corresponds to the grid being transformed by theta[i]

Return type
numpy.ndarray or torch.Tensor

transform_grid_ss(*grid*, *theta*, *method=None*, *time=1.0*, *N=0*)

Integrates a grid using the scaling-squaring method and the parametrization in theta.

Parameters

- **grid** (*numpy.ndarray or torch.Tensor*) – [n_points] vector or [n_batch, n_points] tensor i.e. either a single grid for all theta values, or a grid for each theta value
- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix
- **method** (*str*) – integration method, one of "*closed_form*", "*numeric*".
- **time** (*float*) – integration time. Defaults to 1.0
- **N** (*int*) – number of scaling iterations. Defaults to 0

Returns
[n_batch, n_points] tensor, with the transformed grid. The slice transformed_grid[i] corresponds to the grid being transformed by theta[i]

Return type
numpy.ndarray or torch.Tensor

gradient_grid(*grid*, *theta*, *method=None*, *time=1.0*)

Integrates and returns the gradient of the transformation w.r.t. the parametrization in theta.

Parameters

- **grid** (*numpy.ndarray or torch.Tensor*) – [n_points] vector or [n_batch, n_points] tensor i.e. either a single grid for all theta values, or a grid for each theta value
- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix

Returns

tuple containing

- **transformed_grid** (numpy.ndarray or torch.Tensor): [n_batch, n_points] tensor, with the transformed grid. The slice transformed_grid[i] corresponds to the grid being transformed by theta[i]
- **gradient_grid** (numpy.ndarray or torch.Tensor): [n_batch, n_points, d] tensor, with the gradient grid. The slice gradient_grid[i, j] corresponds to the gradient of grid being transformed by theta[i] and with respect to the parameter theta[j]

Return type
tuple

`gradient_space`(*grid*, *theta*, *method=None*, *time=1.0*)

Integrates and returns the gradient of the transformation w.r.t. the parametrization in theta.

Parameters

- **grid** (*numpy.ndarray or torch.Tensor*) – [n_points] vector or [n_batch, n_points] tensor i.e. either a single grid for all theta values, or a grid for each theta value
- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix

Returns

tuple containing

- **transformed_grid** (numpy.ndarray or torch.Tensor): [n_batch, n_points] tensor, with the transformed grid. The slice transformed_grid[i] corresponds to the grid being transformed by theta[i]
- **gradient_grid** (numpy.ndarray or torch.Tensor): [n_batch, n_points, d] tensor, with the gradient grid. The slice gradient_grid[i, j] corresponds to the gradient of grid being transformed by theta[i] and with respect to the parameter theta[j]

Return type
tuple

`interpolate`(*data*, *grid*, *outsize: int*)

Linear interpolation method

Parameters

- **data** (*numpy.ndarray or torch.Tensor*) – [n_batch, n_points, n_channels] tensor with input data.
- **grid** (*numpy.ndarray or torch.Tensor*) – [n_batch, n_points] tensor with grid points that are used to interpolate the data
- **outsize** – number of points in the output

Returns
[n_batch, outsize, n_channels] tensor with the interpolated data

Return type
interpolated(numpy.ndarray or torch.Tensor)

`transform_data`(*data*, *theta*, *outsize: int*, *method=None*, *time=1.0*)

Combines the transform_grid and interpolate methods

Parameters

- **data** (*numpy.ndarray or torch.Tensor*) – [n_batch, n_points, n_channels] tensor with input data.
- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix

- **outsize** – number of points that is transformed and interpolated
- **method** (*str*) – integration method, one of "*closed_form*", "*numeric*".
- **time** (*float*) – integration time. Defaults to 1.0

Returns
[n_batch, outsize, n_channels] tensor, transformed and interpolated data

Return type
numpy.ndarray or torch.Tensor

transform_data_ss(*data*, *theta*, *outsize*, *method=None*, *time=1.0*, *N=0*)

Combines the transform_grid with scaling-squaring and interpolate methods

Parameters

- **data** (*numpy.ndarray or torch.Tensor*) – [n_batch, n_points, n_channels] tensor with input data
- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix
- **outsize** – number of points that is transformed (with scaling and squaring) and interpolated
- **method** (*str*) – integration method, one of "*closed_form*", "*numeric*"
- **time** (*float*) – integration time. Defaults to 1.0
- **N** (*int*) – number of scaling iterations. Defaults to 0

Returns
[n_batch, outsize, n_channels] tensor, transformed and interpolated data

Return type
numpy.ndarray or torch.Tensor

calc_velocity(*grid*, *theta*)

Calculates the velocity for each point in grid based on the theta parametrization

Parameters

- **grid** (*numpy.ndarray or torch.Tensor*) – [n_batch, n_points] tensor with grid points that are used to interpolate the data
- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix

Returns
[n_points] vector with velocity values

Return type
numpy.ndarray or torch.Tensor

visualize_velocity(*theta*, *n_points: Optional[int] = None*, *fig: Optional[Figure] = None*)

Visualizes the vectorfield for a specific parametrization vector theta

Parameters

- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix
- **n_points** (*int*) – number of points to plot
- **fig** (*matplotlib.figure.Figure*) – matplotlib figure handle

Returns
handle to lines plot

Return type
matplotlib.axes.Axes

visualize_deformgrid(*theta*, *method=None*, *time: float = 1.0*, *n_points: int = 100*, *fig: Optional[Figure] = None*)

Visualizes the deformation for a specific parametrization vector theta

Parameters

- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix
- **method** (*str*) – integration method, one of "*closed_form*", "*numeric*"
- **time** (*float*) – integration time. Defaults to 1.0
- **n_points** (*int*) – number of points to plot
- **fig** (*matplotlib.figure.Figure*) – matplotlib figure handle

Returns
handle to lines plot

Return type
matplotlib.axes.Axes

visualize_tesselation(*n_points: int = 50*, *fig: Optional[Figure] = None*)

Visualizes the tesselation for a specific parametrization vector theta

Parameters

- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix
- **n_points** (*int*) – number of points to plot
- **fig** (*matplotlib.figure.Figure*) – matplotlib figure handle

Returns
handle to tesselation plot

Return type
matplotlib.axes.Axes

visualize_gradient(*theta*, *method=None*, *time: float = 1.0*, *n_points: int = 100*, *fig: Optional[Figure] = None*)

Visualizes the gradient for a specific parametrization vector theta

Parameters

- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix
- **method** (*str*) – integration method, one of "*closed_form*", "*numeric*"
- **time** (*float*) – integration time. Defaults to 1.0
- **n_points** (*int*) – number of points to plot
- **fig** (*matplotlib.figure.Figure*) – matplotlib figure handle

Returns
handle to gradient plot

Return type
matplotlib.axes.Axes

visualize_deformdata(*data*, *theta*, *method=None*, *outsize: int = 100*, *target=None*, *fig: Optional[Figure] = None*)

Visualizes the transformed data for a specific parametrization vector theta

Parameters

- **theta** (*numpy.ndarray or torch.Tensor*) – [n_batch, d] matrix
- **method** (*str*) – integration method, one of "*closed_form*", "*numeric*"
- **outsize** (*int*) – number of points that is transformed and interpolated
- **target** (*numpy.ndarray or torch.Tensor*) – [n_batch, n_points, n_channels] tensor with target data.
- **time** (*float*) – integration time. Defaults to 1.0
- **n_points** (*int*) – number of points to plot
- **fig** (*matplotlib.figure.Figure*) – matplotlib figure handle

Returns
handle to gradient plot

Return type
matplotlib.axes.Axes

3.2 Tessellation

class Tessellation(*nc*, *xmin=0*, *xmax=1*, *zero_boundary=True*, *basis='rref'*)

Regular grid tessellation

cell_centers()

constrain_matrix()

zero_boundary_constrains()

generate_basis_rref()

basis_svd()

basis_qr()

qr_null(*A*, *tol=None*)

basis_rref_zb()

basis_rref_backup()

basis_rref()

basis_rref_zb_new()

basis_rref_new()

generate_basis_sparse()

basis_sparse()

basis_sparse_zb()

plot_basis()

3.3 Backend

3.3.1 Numpy

3.3.1.1 Functions

assert_version()

to(*x*, *dtype=<class 'numpy.float32'>*, *device=None*)

tonumpy(*x*)

check_device(*x*, *device_name*)

backend_type()

sample_transformation(*d*, *n_sample=1*, *mean=None*, *cov=None*, *device='cpu'*)

identity(*d*, *n_sample=1*, *epsilon=0*, *device='cpu'*)

uniform_meshgrid(*xmin*, *xmax*, *n_points*, *device='cpu'*)

calc_velocity(*grid*, *theta*, *params*)

exp(**args*, ***kwargs*)

linspace(**args*, ***kwargs*)

meshgrid(**args*, ***kwargs*)

matmul(**args*, ***kwargs*)

max(**args*, ***kwargs*)

ones(**args*, ***kwargs*)

pdist(*c*)

transformer(*grid*, *theta*, *params*, *method=None*, *time=1.0*)

gradient(*grid*, *theta*, *params*, *method=None*, *time=1.0*)

gradient_space(*grid*, *theta*, *params*, *method=None*, *time=1.0*)

interpolate_grid(*transformed_grid*, *params*)

3.3.1.2 Transformer

batch_effect(*x*, *theta*)

get_affine(*x*, *theta*, *params*)

precompute_affine(*x*, *theta*, *params*)

right_boundary(*c*, *params*)

left_boundary(*c*, *params*)

get_cell(*x*, *params*)

get_velocity(*x*, *theta*, *params*)

get_psi(*x*, *t*, *theta*, *params*)

get_hit_time(*x*, *theta*, *params*)

get_phi_numeric(*x*, *t*, *theta*, *params*)

integrate_numeric(*x*, *theta*, *params*, *time=1.0*)

integrate_closed_form(*x*, *theta*, *params*, *time=1.0*)

integrate_closed_form_trace(*x*, *theta*, *params*, *time=1.0*)

derivative_numeric(*x*, *theta*, *params*, *time=1.0*, *h=0.001*)

derivative_closed_form(*x*, *theta*, *params*, *time=1.0*)

derivative_psi_theta(*x*, *t*, *theta*, *k*, *params*)

derivative_phi_time(*x*, *t*, *theta*, *k*, *params*)

derivative_thit_theta(*x*, *theta*, *k*, *params*)

derivative_space_numeric(*x*, *theta*, *params*, *time=1.0*, *h=0.001*)

derivative_space_closed_form(*x*, *theta*, *params*, *time=1.0*)

derivative_thit_x(*x*, *t*, *theta*, *params*)

derivative_psi_x(*x*, *t*, *theta*, *params*)

derivative_psi_t(*x*, *t*, *theta*, *params*)

3.3.1.3 Interpolation

interpolate(*data*, *grid*, *outsize*)

interpolate_grid(*data*)

3.3.2 Pytorch

3.3.2.1 Functions

assert_version()

to(*x*, *dtype=torch.float32*, *device=None*)

tonumpy(*x*)

check_device(*x*, *device_name*)

backend_type()

sample_transformation(*d*, *n_sample=1*, *mean=None*, *cov=None*, *device='cpu'*)

identity(*d*, *n_sample=1*, *epsilon=0*, *device='cpu'*)

uniform_meshgrid(*xmin*, *xmax*, *n_points*, *device='cpu'*)

exp(**args*, ***kwargs*)

linspace(**args*, ***kwargs*)

meshgrid(**args*, ***kwargs*)

matmul(**args*, ***kwargs*)

max(**args*, ***kwargs*)

ones(**args*, ***kwargs*)

pdist(*c*)

3.3.2.2 Transformer

cmpf(*x*, *y*)

cmpf0(*x*)

batch_effect(*x*, *theta*)

get_affine(*x*, *theta*, *params*)

precompute_affine(*x*, *theta*, *params*)

right_boundary(*c*, *params*)

left_boundary(*c*, *params*)

get_cell(*x*, *params*)

get_velocity(*x*, *theta*, *params*)

calc_velocity(*grid*, *theta*, *params*)

get_psi(*x*, *t*, *theta*, *params*)

get_hit_time(*x*, *theta*, *params*)

get_phi_numeric(*x*, *t*, *theta*, *params*)

integrate_numeric(*x*, *theta*, *params*, *time=1.0*)

integrate_closed_form(*x*, *theta*, *params*, *time=1.0*)

derivative_numeric(*x*, *theta*, *params*, *time=1.0*, *h=0.001*)

derivative_closed_form(*x*, *theta*, *params*, *time=1.0*)

derivative_psi_theta(*x*, *t*, *theta*, *k*, *params*)

derivative_phi_time(*x*, *t*, *theta*, *k*, *params*)

derivative_thit_theta(*x*, *theta*, *k*, *params*)

integrate_closed_form_trace(*x*, *theta*, *params*, *time=1.0*)

derivative_closed_form_trace(*result*, *x*, *theta*, *params*)

derivative_numeric_trace(*phi_1*, *x*, *theta*, *params*, *time=1.0*, *h=0.001*)

derivative_space_numeric(*x*, *theta*, *params*, *time=1.0*, *h=0.001*)

derivative_space_closed_form(*x*, *theta*, *params*, *time=1.0*)

derivative_thit_x(*x*, *t*, *theta*, *params*)

derivative_psi_x(*x*, *t*, *theta*, *params*)

derivative_psi_t(*x*, *t*, *theta*, *params*)

3.3.2.3 Interpolation

interpolate(*data*, *grid*, *outsize*)

interpolate_grid(*data*)

Cpab(tess_size[, backend, device, ...])	Continous piecewise-affine based transformations.
Tessellation(nc[, xmin, xmax, ...])	Regular grid tessellation

CHAPTER

FOUR

AUTHORS

This package is being developed and maintained by Iñigo Martinez at the department of Data Intelligence for Energy & Industrial Processes at Vicomtech.

4.1 Lead Development Team

- Iñigo Martinez (imartinez@vicomtech.org)
- Igor Garcia Olaizola (iolaizola@vicomtech.org)

CHAPTER

FIVE

CITING DIFW

This software is released under the MIT License.

difw was accepted as a short paper and published in the Proceedings of the 39 th International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

Note that if you use this code and/or the results of running it to support any form of publication we request to include the following citation:

Bibtex entry:

```
@article{https://doi.org/10.48550/arxiv.2206.08107,
  doi = {10.48550/ARXIV.2206.08107},
  url = {https://arxiv.org/abs/2206.08107},
  author = {Martinez, Iñigo and Viles, Elisabeth and Olaizola, Igor G.},
  keywords = {Machine Learning (cs.LG), Artificial Intelligence (cs.AI), FOS:␣
↪Computer and information sciences, FOS: Computer and information sciences},
  title = {Closed-Form Diffeomorphic Transformations for Time Series Alignment}
↪,
  publisher = {arXiv},
  year = {2022},
  copyright = {arXiv.org perpetual, non-exclusive license}
}
```

CHAPTER

SIX

CHANGELOG

6.1 Version 0.0.1

- Alpha release of difw

CHAPTER

SEVEN

LICENSE

```
MIT License

Copyright (c) 2021 Iñigo Martínez

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

E

G

I