

# MF2022-Exercise3

March 22, 2022

## MathFound - EXERCISE 3 - MLS FOR CURVES, MESHES AND IMAGES

### General Rules

**Plagiarism note.** Copying code (either from other students or from external sources) is strictly prohibited! We will be using automatic anti-plagiarism tools, and any violation of this rule will lead to expulsion from the class.

Late submissions will not be accepted, except in case of serious illness or emergency. In that case please notify the assistants and provide a relevant medical certificate.

**Software.** All exercises of this course use Python.

**What to hand in.** Upload a .zip file of your solution in Moodle. The file must be called `MATHFOUND22-3-firstname-familyname.zip`. The .zip file MUST contain the following data inside:

- A jupyter notebook with the solution to the exercise and a description of what you implemented and comments/explanations on your results.

**OR**

- The solution in .py format and a `readme.pdf` file containing a description of what you've implemented and instructions for running it, as well as explanations/comments on your results.

**Grading** This homework is 8.3% of your final grade. Your submission will be graded according to the quality of the images produced by your program, the conformance of your program to the expected behaviour of the assignment, and your understanding of the underlying techniques used in the assignment. The submitted code must produce exactly the same images included in your submission.

### Goal of this exercise

In this exercise you will apply what you learned about the Moving Least Squares (MLS) method to curves, meshes and images. You will read and implement parts of research papers that used MLS.

## 1 Image deformation using MLS

The first task of this assignment is to write code which deforms images based on moving least squares as presented in the paper “Image Deformation using Moving Least Squares” by Schaefer et al. (provided with the exercise). In this paper, the authors propose a method to smoothly deform

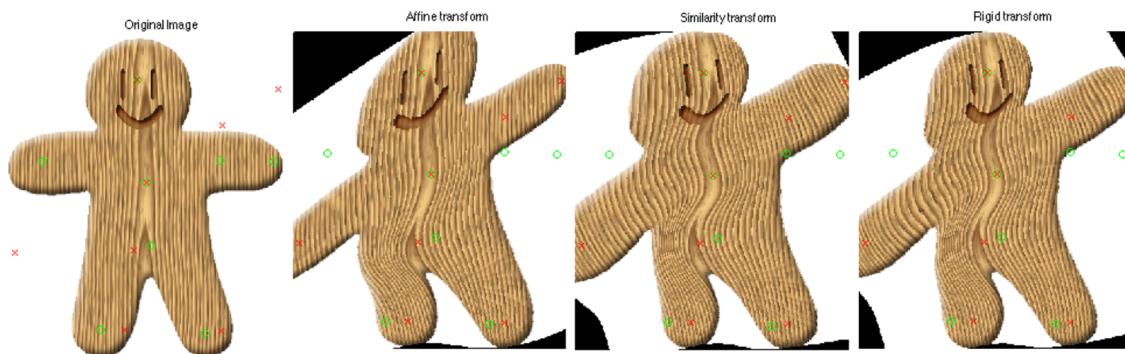
images using different kinds of transformations and allow the user to manipulate sets of points and segments to specify the deformations. For this exercise you will only focus on image deformations using sets of points.

Write a program which takes as input an image, some input and output control points and performs the deformation. The algorithm should follow the description from the paper. The three kinds of transformation, i.e. affine, similarity and rigid transformations should be supported by your program. Please note that we don't ask for an interactive application so that you don't need to worry about making the code fast (precomputation of some terms, approximation of the image by a grid), but you can do it if you wish.

Some pointers to help with your implementation:

- Read section 2 of the paper to understand the types of transformations you need to apply.
- Implement backwards warping to avoid artifacts (estimate transformation from the output points to the input points).
- You can use `scipy.ndimage.map_coordinates` to implement the warping process
- To optionally implement the image approximation by a grid for a significant speedup, you can use `numpy.meshgrid` and `scipy.interpolate.griddata`

Your result should resemble the following, where the green and red points are the input and output control points respectively.



### Required output

- Code that deforms images according to user input using all three methods described above (affine, similarity and rigid).
- Representative image results of the gingerbread man (provided with the exercise) and at least one more image of your choice.

### Selecting the control points interactively (optional)

If you want to create an interactive version of your image deformation code, you can look into using `ipywidgets` which allows you to set a callback function when the user clicks on the displayed image (with `matplotlib`). Any other solution is also fine (for example if you want to make a standalone `.py` for that part :).

## Solution

```
[1]: # write your solution here
```

## 2 Curve and Surface Reconstruction Using MLS

As described in the lectures, moving least squares (MLS) can be used to approximate a function given samples from that function at sample points. In this exercise, we will utilize this method to reconstruct curves and smooth meshes. We provide a few example datasets.

### 2.1 Implicit MLS Surfaces

Given some samples points and attributes (i.e. normals, colors), we seek to find a function  $f(\mathbf{x}) : \mathbb{R}^2 \text{ or } \mathbb{R}^3 \rightarrow \mathbb{R}$  such that the curve or surface is given by  $\mathcal{S} = \{\mathbf{x} | f(\mathbf{x}) = 0, \nabla f(\mathbf{x}) \neq \mathbf{0}\}$

### 2.2 Curves derivation

MLS based surfaces can be computed by considering different degrees for the local approximations and constraints, as described in the lectures. The first part of the exercise is to derive on paper one of the simplest definitions for MLS based surfaces. For this task, we assume that the locally fit polynomial consists of a single constant term,  $f(\mathbf{x}) = c_0$ ,

$$\operatorname{argmin}_{c_0} \sum_i \phi_i(\mathbf{x}) (c_0 - f_i(\mathbf{x}))^2$$

Using the following local functional approximation

$$f_i(\mathbf{x}) = \mathbf{n}_i^T (\mathbf{x} - \mathbf{x}_i)$$

**derive the closed-form expression for the resulting implicit function  $f(\mathbf{x})$ .**

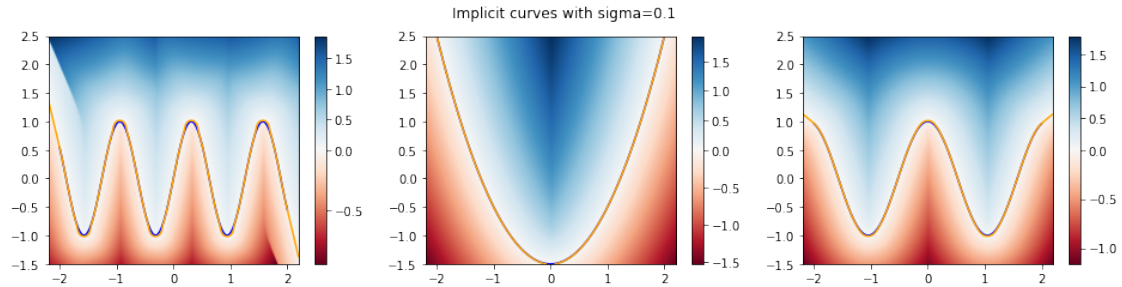
Hint: it is a weighted least squares problem for the scalar  $c_0$ , and hence reduces to a simple normalized average.

## Solution

*write your answer here*

### 2.3 Curves plotting

Evaluate the derived  $f(\mathbf{x})$  for the given example point sets with 2D points and normals, on a regular grid. For the definition of  $\phi$ , use a Gaussian of a chosen  $\sigma$  such that  $\phi(r) = e^{-r^2/\sigma^2}$ . Initialize a matrix representing the grid. Then, for each grid center, evaluate the function  $f$  and store the value at the entry of the matrix corresponding to that grid. Finally, plot this matrix as a heatmap or similar, overlaid with the sample points in the same image (as shown below). Optionally visualize the zero level set of the matrix (in orange).



Generate a few images with different  $\sigma$  values to illustrate how it affects the approximation and discuss your results. You can see the reconstructed curve by observing the values of  $f$  close to zero. How is the value of  $\sigma$  influencing the results ?

### Implementation hints

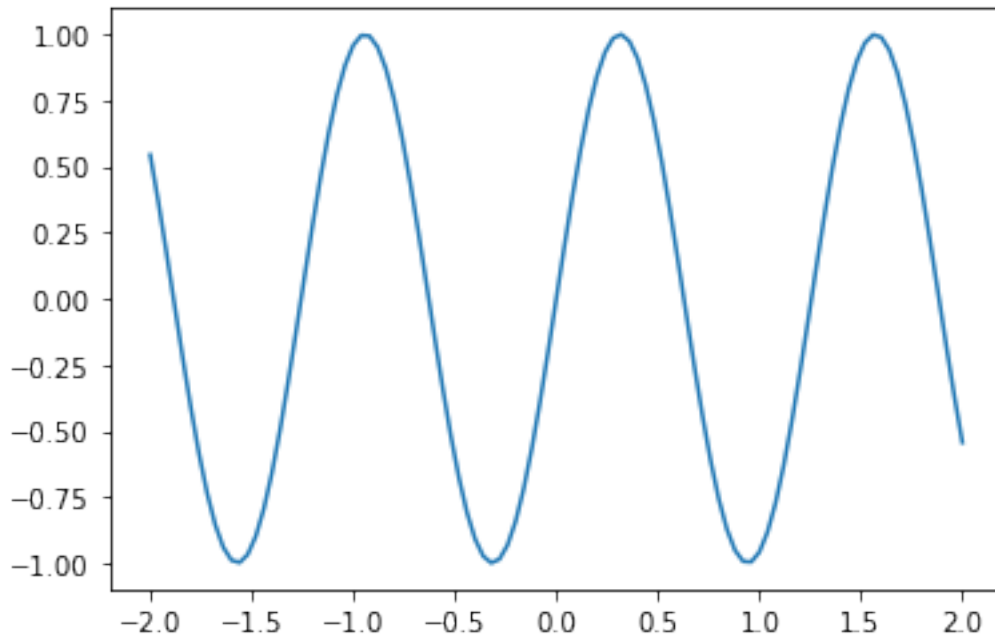
- You can use `imshow` from `matplotlib` to visualize the matrix
- If the plotted  $f(\mathbf{x})$  looks too smooth, try decreasing  $\sigma$ . Similarly, if you get discontinuous  $f(\mathbf{x})$ , try increasing  $\sigma$ .
- You can use `numpy` to load the data:

```
[2]: import matplotlib.pyplot as plt
import numpy as np

curve_data = np.load('data/curve_data1.npz')
points = curve_data['points']
normals = curve_data['normals']

plt.plot(points[:,0], points[:,1])

plt.show()
plt.close()
```



### Solution

```
[3]: # Curve plotting functions to help you get started, feel free to modify
import matplotlib.colors

def plot_curve(fig, ax, points, x, y, grid):
    extent = (np.min(x), np.max(x), np.min(y), np.max(y))
    norm = matplotlib.colors.TwoSlopeNorm(vcenter=0.0)

    mappable = ax.imshow(grid, origin="lower", extent=extent, cmap="RdBu",
        ↪ norm=norm)
    ax.plot(points[:,0], points[:,1], color="blue")
    ax.contour(x, y, grid, levels=[0], colors="orange")

    fig.colorbar(mappable, ax=ax)

[4]: NX, NY = 300, 200 # grid resolution
x_axis = np.linspace(-2.2, 2.2, NX)
y_axis = np.linspace(-1.5, 2.5, NY)

fig, axis = plt.subplots(1, 3, figsize=(16, 3.5))

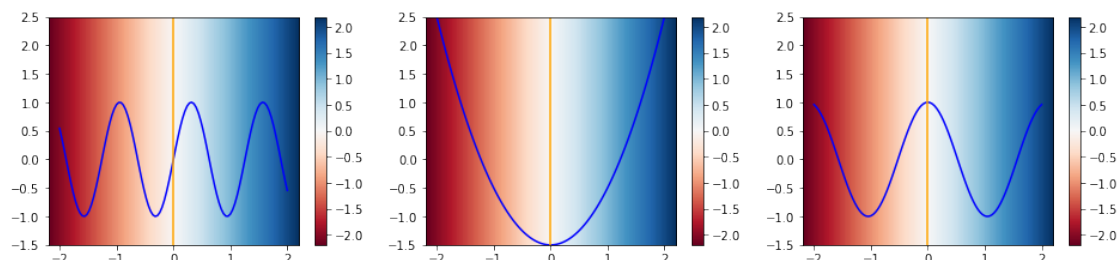
for curve_i, ax in enumerate(axis):
    curve_data = np.load(f'data/curve_data{curve_i + 1}.npz')
    points = curve_data['points']
```

```

normals = curve_data['normals']

grid = [x_axis] * NY
# TODO - fill the `grid` with  $f(x)$  values
plot_curve(fig, ax, points, x_axis, y_axis, grid)

```



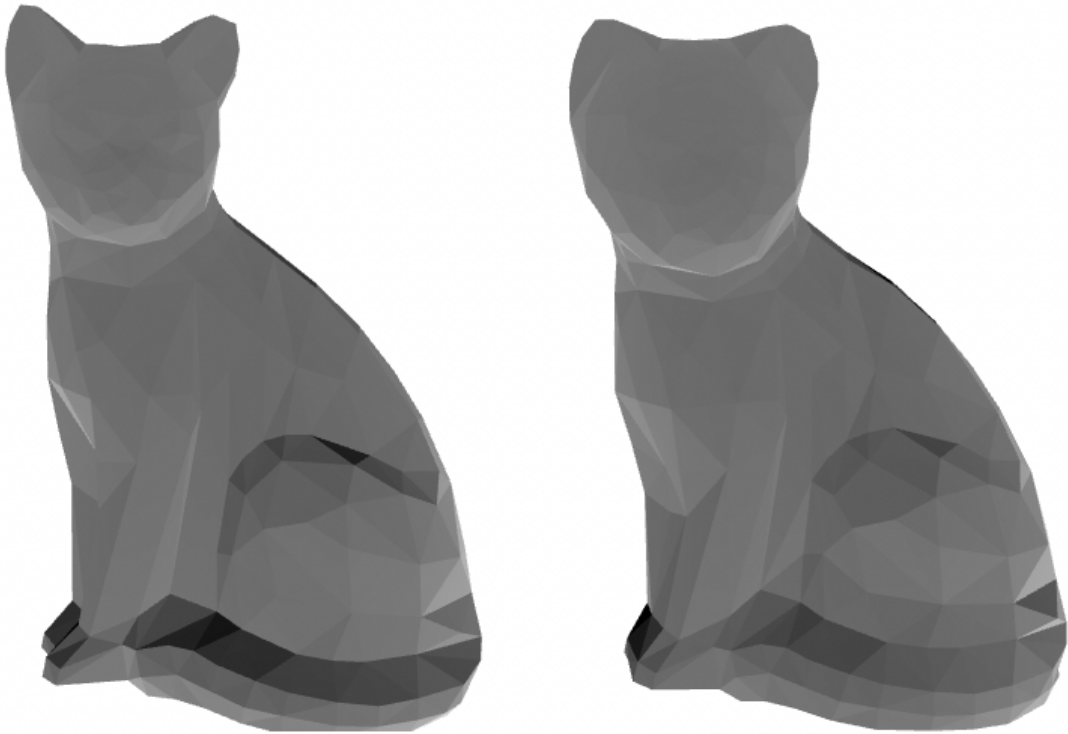
### Optional bonus exercise - numerically stable implementation

*Added to Exercise 3, maximum points cannot exceed 100%*

You will observe that with small  $\sigma$ , the function  $f$  becomes numerically unstable at some regions of the grid. Derive and implement numerically stable version of  $f(\mathbf{x})$  with the given  $\phi(r)$

## 2.4 Smoothing meshes

Load the vertices and normals of the provided meshes. These will serve as the sample points and normals. Then, project each vertex coordinate  $\mathbf{v}$  of the mesh by  $\mathbf{v} = \mathbf{v} - f(\mathbf{x})\nabla f(\mathbf{x})$ , where  $f(\mathbf{x})$  is the same as in the first task. This will give you new smoothed coordinates  $\mathbf{v}$  of the mesh vertices. The expression for  $\nabla f(\mathbf{x})$  can be derived on paper - use the chain rule, the result is a long formula :). The answer is not the same as the one in the attached paper (but part of it is similar). Apply this smoothing a few times, obtaining a result similar to this



To load, modify and display meshes, you can use the `trimesh` package (`pip install --user trimesh`). Here's a quick example to load and visualize the stanford bunny mesh

```
[5]: import trimesh
import numpy as np

[ ]: # To load a mesh
mesh = trimesh.load_mesh('data/bunny.off', process=False)

# To access the vertices, faces and normals
vertices = mesh.vertices
faces = mesh.faces
normals = mesh.vertex_normals

# ... update vertices

# Create a new mesh with the updated vertices
new_mesh = trimesh.Trimesh(vertices, faces)

new_mesh.show()
```

## Hints

- The expression for  $\nabla f(\mathbf{x})$  can be easily derived on paper. If you wish, you may confirm your expression by comparing it to the expression in Section 4.3 of the paper “Feature Preserving

Point Set Surfaces based on Non-Linear Kernel Regression” provided next to this notebook.

- The function  $f(\mathbf{x})$  (and  $\nabla f(\mathbf{x})$ ) is always computed by using the initial positions and normals of the vertices, even if you do multiple iterations of projections, only the query points  $\mathbf{x}$  change.
- As in the previous exercise, try changing  $\sigma$  to see its smoothing effect.
- You can use `mesh.scale` to scale  $\sigma$  proportionally to the mesh size

### Required outputs

- The derivation of  $\nabla f(\mathbf{x})$
- The mesh smoothing code
- Screenshots of results produced by your code for both meshes

### Solution - derivation

*write your answer here*

### Solution - smoothing code

```
[7]: # write your solution here
```