EMODnet Bathymetry

Improving interpolation techniques for production of EMODnet DTMs

Technical Report

Date: 15 December 2021
Prepared by: CORONIS

The European Marine Observation and Data Network (EMODnet) is financed by the European Union under Regulation (EU) 2021/1139 of the European Parliament and of the Council of 7 July 2021 establishing the European Maritime, Fisheries and Aquaculture Fund and its predecessor, Regulation (EU) No. 508/2014 of the European Parliament and of the Council of 15 May 2014 on the European Maritime and Fisheries Fund.

Table of Contents

1. Introduction:

During this first year of the project, Coronis focused on improving the python interpolation package, which is the core library for the interpolation methods to be implemented in Globe.

While the plan is to use it within Globe, the package can be used standalone, as well as a typical python library (to be added in other projects). The source code of the package is publicly available at:
https://github.com/coronis-computing/heightmap_interpolation

Also, a complete documentation, expanding the information in the present report, and including installation and usage information, was made available at:
https://emodnet-heightmap-interpolation.readthedocs.io/en/latest/

Moreover, with the aim of easing the adoption of the library, we also provide an already compiled docker image at DockerHub:
https://hub.docker.com/r/coroniscomputing/heightmap_interpolation

Several data providers and regional coordinators were kindly asked to provide relevant datasets requiring some kind of interpolation. The study on the density and distribution of the samples in those datasets put into relief that a single interpolation method may not fit the needs of all users. Consequently, we increased the number of interpolation algorithms in the Python package initially developed in HRSM2. Moreover, in order to ease usage, we unified all methods within the single command line entry point *interpolate_netcdf4.py*.

In the following sections, we start by introducing the interpolation problem, as well as the motivations for implementing several methods in the package. Then, we list all the interpolation methods implemented, along with their suitability depending on the input data and their pros/cons. Finally, we present the work to be done in the second year of the project.

2. Interpolation Methods

“Interpolation” is a broad term. In our case, it consists in obtaining elevation values at cells/points given a set of known reference elevation data at known locations. However, depending on the sampling/distribution of the input data, and where we want to interpolate it, there are several ways of dealing with this problem.

The typical literature for interpolation does not consider any specific distribution for the samples. In this sense, we find the **Scattered-data interpolators**. These methods work in two steps:

1. Take the known data points as reference to create an *interpolator*.
2. Apply the interpolator at whatever query point you desire. For interpolations on a grid, as in our case, the interpolation is queried at all the grid cells to be interpolated.

However, there are several cases in which the interpolation problem consists in filling "missing data", in the sense of having continuous and densely-sampled parts of the map that are missing and that we need to fill given the known data surrounding these parts. In these cases, the problem can be seen as "filling the holes in a coherent way". Obviously, the scattered data interpolators can be used for this purpose. However, there is a wide literature of methods trying to take advantage of the "filling" happening on a regular grid. In the computer vision literature, these are called **inpainting** methods. In this package we use inpainting approaches, usually devised for image processing, to tackle the interpolation problem on elevation grids. As mentioned above, these methods only work on the regular grids, but provide the advantage of providing **higher-degree** approximations **faster** than some similar approaches in the scattered area, and require **much less memory** to execute (the solver we implement just applies convolution operations on the input grid).

In the following sections, for each of the methods in the package, we will briefly describe their behaviour, provide the cases for which a given method is more suitable, and list their pros/cons. For a complete reference on how to call each method, and the list of parameters available to tune in each case, please refer to the documentation.

In addition, in order to get a qualitative evaluation of the behaviour of each method, we will run them with default parameters on the following dataset:

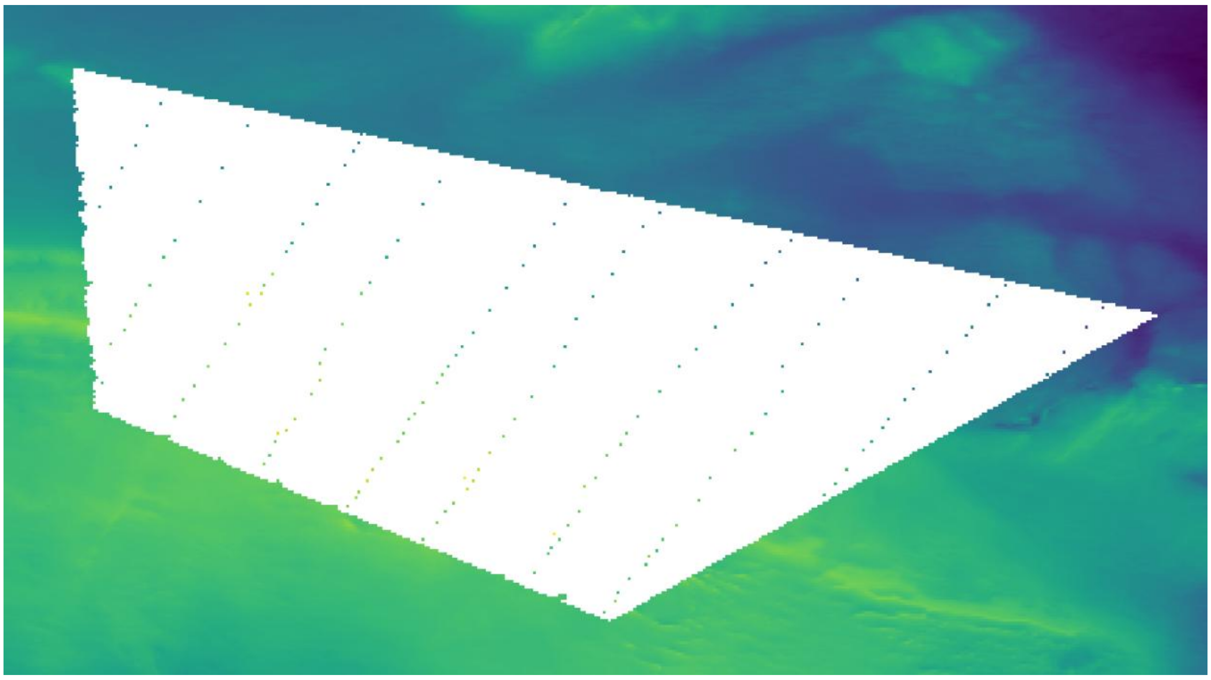

Figure 1. Example dataset. Colored areas and points represent the known reference elevation data, while the area to interpolate is shown in white. Data by courtesy of the Swedish Maritime Administration

Note that this dataset mixes both scattered and densely-sampled reference data.

3. Scattered data Interpolators

Nearest Neighbors

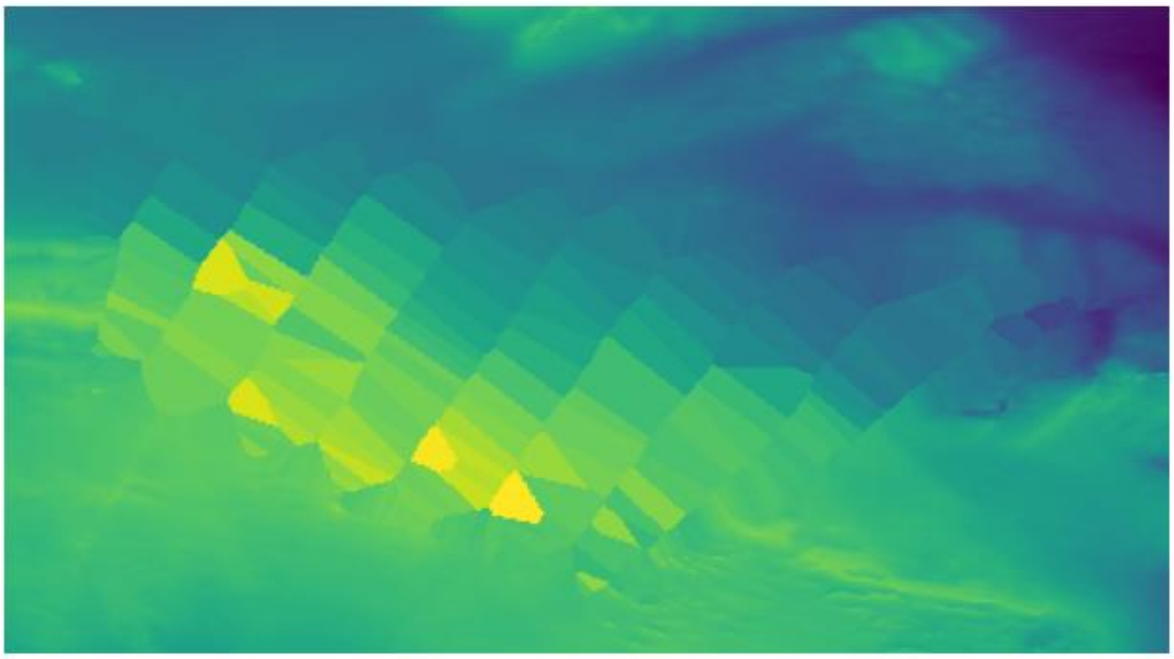

Figure 2. Example dataset interpolated using the Nearest Neighbors interpolant (*nearest* option in interpolate_netcdf4.py).

Each cell to interpolate gets its value from the nearest reference cell.

Suitable for

- Quick initialization of the interpolation using PDE inpainters (see sections below).
- Quick large-area interpolation.

Advantages

- Fastest interpolator.
- As opposed to the other two fast scattered data interpolation methods (*linear* and *cubic*), it can interpolate outside of the convex hull of the reference data.

Disadvantages

- Results look *blocky*, as many points get the same elevation value.

Linear

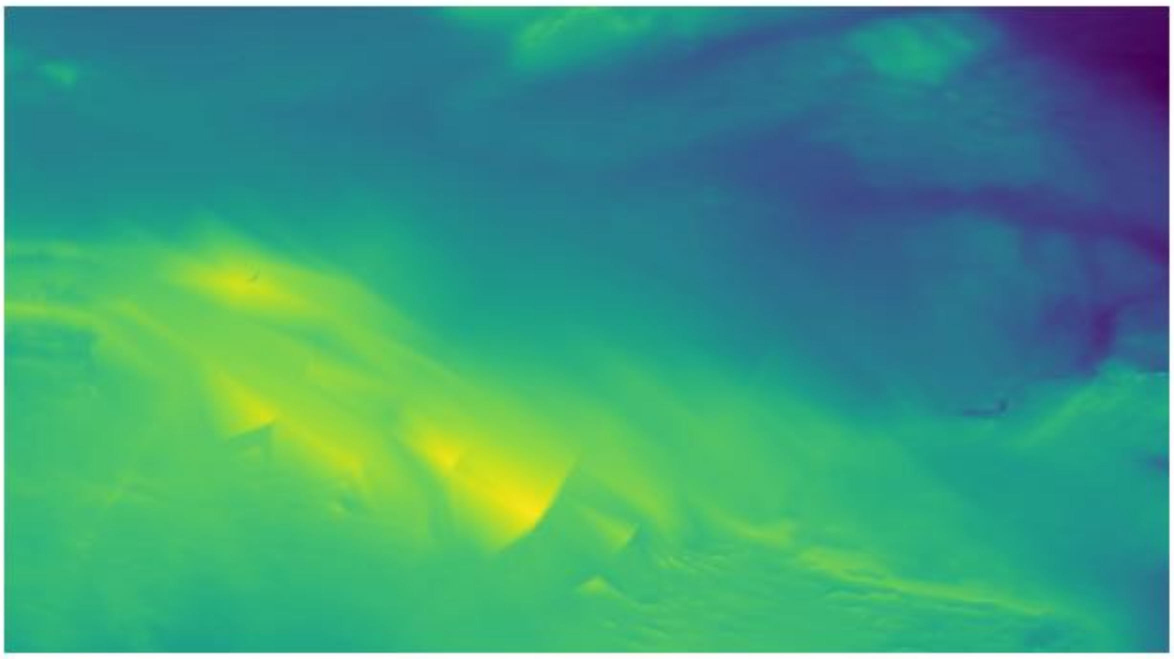

Figure 3. Example dataset interpolated using the Linear interpolant (*linear* option in interpolate_netcdf4.py).

Computes a linear interpolant by creating a 2D Delaunay triangulation using the reference data points. Upon a given query point, it searches in which of the triangle in the XY plane it falls, and computes a barycentric interpolation of the elevation using the reference values at the vertices of the triangle.

Suitable for

- Quick large-area interpolation.

Advantages

- Fast classical interpolation method applicable to large areas.

Disadvantages

- May produce artifacts if samples' density vary rapidly, or if the scattered samples are not uniformly distributed over the inpainting area (see figure above).
- Does not "extrapolate" in query locations outside of the convex hull of the reference data.

Cubic

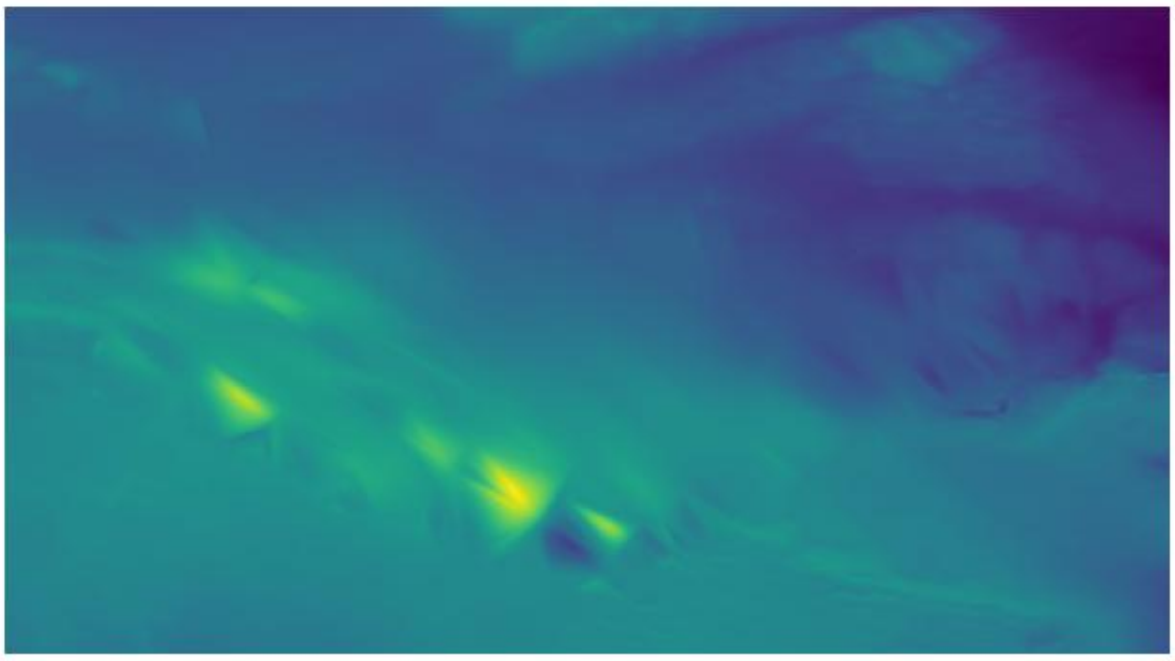

Figure 4. Example dataset interpolated using the Cubic interpolant (*cubic* option in interpolate_netcdf4.py).

As in the *linear* method, it creates a 2D Delaunay triangulation using the reference data points and query points are interpolated within the triangle where they fall in the XY plane. However, as opposed to using a linear barycentric interpolation within the triangle, it uses a piecewise cubic interpolating Bezier polynomial.

Suitable for

- Quick large-area interpolation.

Advantages

- Provides a smoother interpolation than the *linear* method at a similar computational cost.

Disadvantages

- May produce artifacts if samples' density vary rapidly, or if the scattered samples are not uniformly distributed over the inpainting area (see figure above).
- Does not "extrapolate" in query locations outside of the convex hull of the reference data.

Radial Basis Functions

This method was developed in the previous phase of EMODnet Bathymetry (HRSM2), and included in the package as part of the new command-line interface.

A Radial Basis Function (RBF) is a function whose value depends only on the distance between the input and some fixed point. The basic idea of a RBF interpolator is to construct an interpolant of the data using a summation of several RBF centered at the input reference data points. The formal definition is the following:

$$s(x) = p(x) + \sum_{i=1}^{N} \lambda_i \phi(|x - x_i|)$$

Where $\phi(|x - x_i|)$ is a given radial basis function centered at a known/reference data point x_i, $p(x)$ is a polynomial of small degree evaluated at point x, and λ_i is a scalar weight.

Thus, basically, we have a polynomial (1st term) capturing the main trend of the data, and the summation of weighted RBFs (2nd term). Therefore, the unknowns of this interpolant are mainly the few terms of the polynomial $p(x)$ and the λ_i weight of each RBF. These unknowns can be solved using a linear system of equations. In matrix form, this corresponds to:

$$A = \begin{pmatrix} A & P \\ P^T & 0 \end{pmatrix} \begin{pmatrix} \lambda \\ c \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix}$$

Where:

- $A_{i,j} = \phi(|x_i - x_j|)$
- $P_{i,j} = p_j(x_i)$ are the coefficients of the polynomial.
- f are known elevation values at x_i.

While solving this system of equations is conceptually simple, it is important to notice that the matrix A is a square matrix with side length equal to the number of input data points. Therefore, this formulation becomes prohibitively complex for large datasets, as the amount of memory and computational resources required for solving and/or evaluating the interpolant is too large. This is the reason why there is no figure showing the result in this section: even for a small dataset as the one we are using, **it is not feasible to compute the interpolant in a reasonable amount of time and resources**.

However, it has the nice feature of allowing some “tuning” of the properties of the interpolating surface via the RBF type that we choose. You can check the complete list of RFB types available in the documentation.

Suitable for

- Best approximation quality for the interpolant.

- Small datasets. They can be small in the number of input reference points, and large in the number of query points (huge scattered data).

Advantages

- Allows tuning the properties of the interpolating surface by changing the RBF type and parameters.

Disadvantages

- Depending on the input data and the selected RBF type, the resulting interpolant surface may **overshoot** the input data (minimum and maximum elevation values may be outside the range of the input data).

Partition of Unity Radial Basis Functions

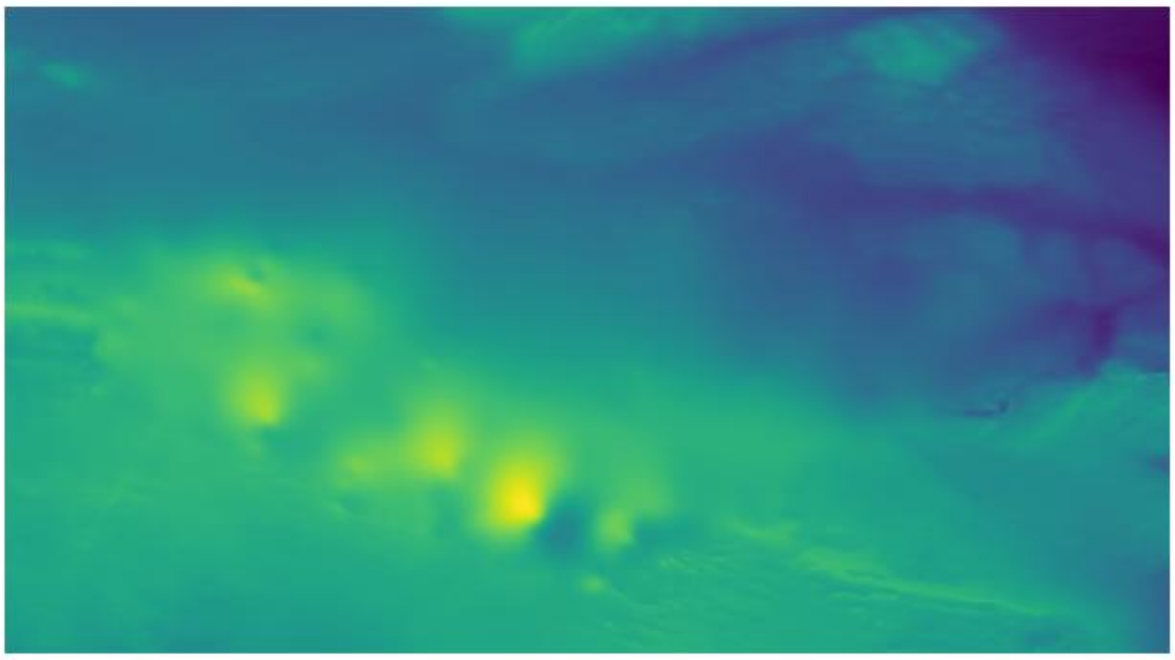

Figure 5. Example dataset interpolated using the Partition of Unity Radial Basis Functions interpolant (*purbf* option in interpolate_netcdf4.py).

This method was also developed in EMODnet Bathymetry HRSM2. Based on the low applicability of the original definition of the RBF interpolant, the Partition of Unity Radial Basis Functions (*purbf*) is an attempt to lower as much as possible the memory and computational requirements of the RBF interpolator.

The Partition of Unity Method (PUM) divides the global domain into smaller overlapping subdomains. In each of these subdomains, a RBF interpolant is computed using the formulation presented in *Radial Basis Functions*. Then, when evaluating a query location, the contributions of several neighboring RBF interpolations are *blended* together in order to get the final value.

More precisely, we enforce a quadtree decomposition. In the following figure we can see an example of this decomposition:

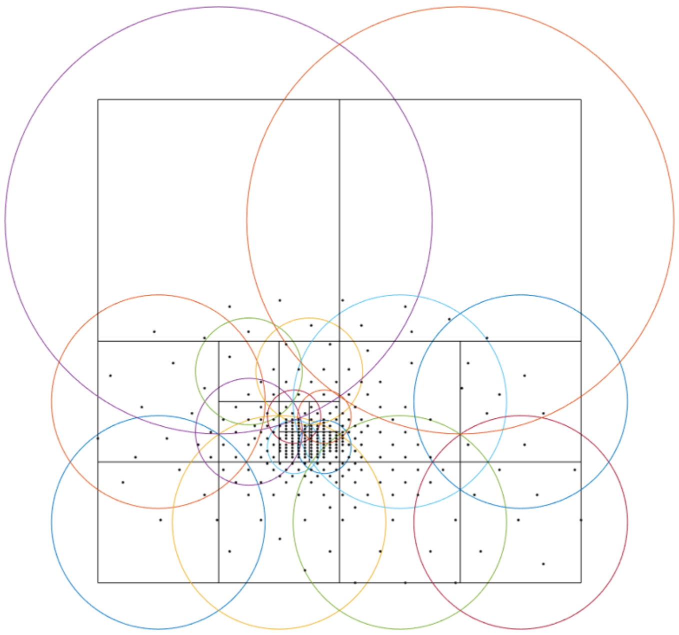

Figure 6. An example of the decomposition in *purbf* method. Reference data points are marked as black dots, the quadtree decomposition is shown using squares, and the domain of each local RBF corresponding to each square is shown with a colored circle.

Each cell in the quadtree defines a local RBF interpolant and its area of influence. Note how the different areas overlap between them (a condition for continuity) and how the area of influence of each local interpolant adapts to the complexity of the data (larger regions in more sparse areas, and smaller regions in denser ones). Finally, since the extent of local RBF is limited, we also ensure that at least one local interpolant covers all the data within the possible query space (i.e., it covers the extent of the input grid).

The PU interpolant preserves the local approximation order for the global fit. Therefore, large RBF interpolants can be computed by solving small interpolation problems and then combining them together with the global PU.

Suitable for

- Datasets for which the basic RBF interpolator required too much memory and computational resources.

Advantages

- Tunnable output: as in the RBF interpolator, changing the base RBF will change the shape/properties of the output interpolated surface.
- Preferrable in cases where the number of reference data points is far smaller than the number of points to interpolate.

Disadvantages

- While compared to the pure RBF, reduction in computational requirements is huge, it may not be sufficient for processing large datasets (i.e., it will still be slower to compute than other options in this package).

4. PDE-based Inpainting Interpolators

Our heightmaps are bivariate functions of the form $u(x, y) = z$, where x/y are the coordinates in a plane and z the corresponding elevation value.

A simple way of defining the interpolant is to define the properties that the "interpolating surface" $f(u)$ must be satisfied at interpolated areas using Partial Differential Equations (PDEs).

Once defined a given PDE, we can solve it using finite differences in a gradient-descent manner, where:

$$f(u)_{t+1} = u_t - \phi * \nabla(f(u_t))$$

Being the subindex t the iteration index, $\nabla(f(u_t))$ the PDE or the *gradient* that we need to follow,ϕ the size of the update step at each iteration. Given a properly small ϕ, we can iterate the equation above to *steady state* (i.e., no change) in order to solve for the functional.

Using discretized differential stencils, we can work directly on the input cell grid, and evolve the previous equation using just convolutions. We implement all the methods in this section using the same PDE solver. However, we explain in the next section some of the speed-up tricks that we use to accelerate the classical gradient descent optimization.

Speed-Up Tricks

The convergence speed of the gradient descent optimization on the inpainted area is highly dependent on the initial values. It is not the same as trying to evolve the solution using the optimization starting from a very vague solution (e.g. all unknown initial values are zero) rather than starting from initial values closer to the solution. In this direction, we provide two ways to better initialize the problem in Initializer and Multi-Grid Solver below.

Initializer

The initializers available are:

- *zeros*: init unknown values with zeros. This is the worst initializer, just kept here for comparison purposes with the rest.
- *mean*: init unknown values with a constant equal to the mean of the reference elevation values.
- *nearest*: use the nearest interpolant to initialize unknown values.
- *linear*: use the linear interpolant to initialize unknown values. Since this interpolant is just defined over the convex hull of the input data, data outside it will get a constant value equal to the mean of the reference elevation values.
- *cubic*: use the cubic interpolant to initialize unknown values. Same as in *linear*, it will get the mean value of reference value outside the convex hull of the reference data points.
- *harmonic*: uses the harmonic inpainter to fill the missing data. Note that, while being the fastest of the inpainter methods, this involves solving another gradient descent optimization, so depending on the complexity of the data it may be very slow.

Multi-Grid Solver

By setting the proper parameters, the interpolate_netcdf4.py function will use a Multi-Grid Solver (MGS). Basically, instead of solving the optimization problem at the full resolution grid directly, it will do it in a multi-resolution way.

The MGS starts building a pyramid of different levels of resolution from the original grid, where each level of the pyramid contains a halved resolution version of the previous one:

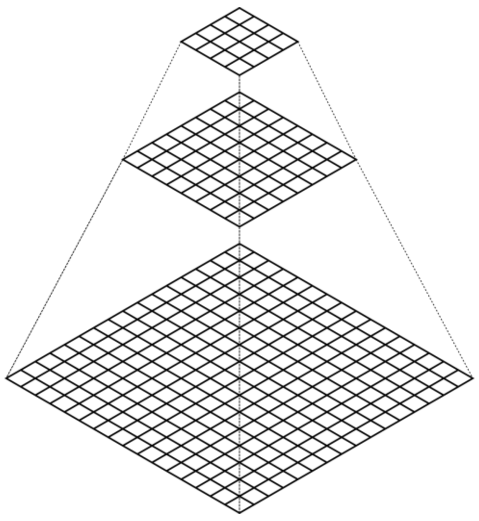

Figure 7. Schematic of the multi-resolution pyramid created by the Multi-Grid Solver. The original grid (bottom of the pyramid) is halved in resolution recursively to get lower resolution versions of the problem. Then, starting from the top of the pyramid, the inpainting problem

is solved in a lower resolution version, and upscaled and propagated to the next (higher resolution) level of the pyramid as initial guess.

Then, starting from the coarser level, we solve the inpainting problem there, and use that solution to initialize the solver in the next (higher resolution) level of the pyramid.

Therefore, we use upscaled versions of the problem solved at coarser resolutions to initialize the inpainting problem at higher resolutions. In this way, the initial values of the optimization at each level of the pyramid are closer to the final solution, decreasing like this the number of iterations required for convergence.

Harmonic Inpainter

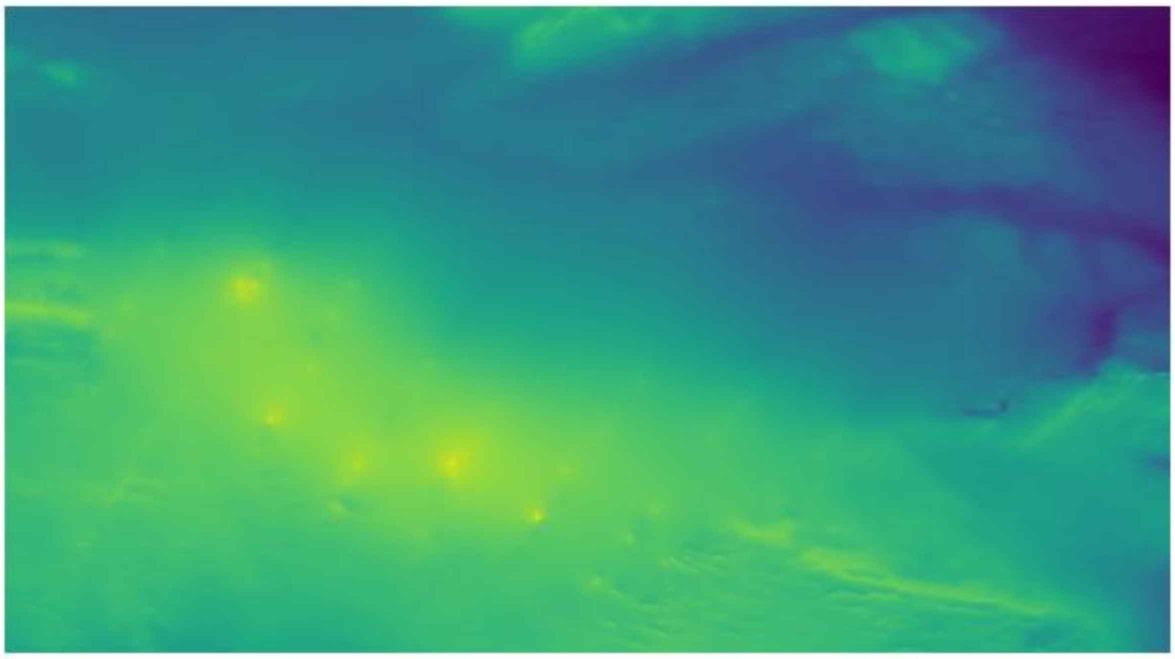

Figure 8. Example dataset interpolated using the Harmonic inpainter (*harmonic* option in interpolate_netcdf4.py).

An harmonic surface is a twice differentiable function satisfying the Laplace equation:

$$\nabla(f(u_t)) = \nabla^2 u_t = 0$$

This method has many analogies:

- It can be seen as an "isotropic diffusion" of the elevation values at the borders surrounding the missing data towards the area to interpolate.
- Its evolution follows the heat diffusion equation.
- It minimizes the Sobolev norm on the grid, constrained to the input reference data.
- The interpolated surface is a "minimum energy surface", and many times it is described as the "shape a film of soap would take if layed over the data points".

Suitable for

- Filling large gaps smoothly without overshooting the input data.

Advantages

- Fastest of the inpainting methods.
- It will never overshoot the data (minimum and maximum elevation values never below/over the reference ones).

Disadvantages

- Does not work well with sparsely sampled data: isolated data points will not contribute much to the interpolation.

Total Variation (TV) Inpainter

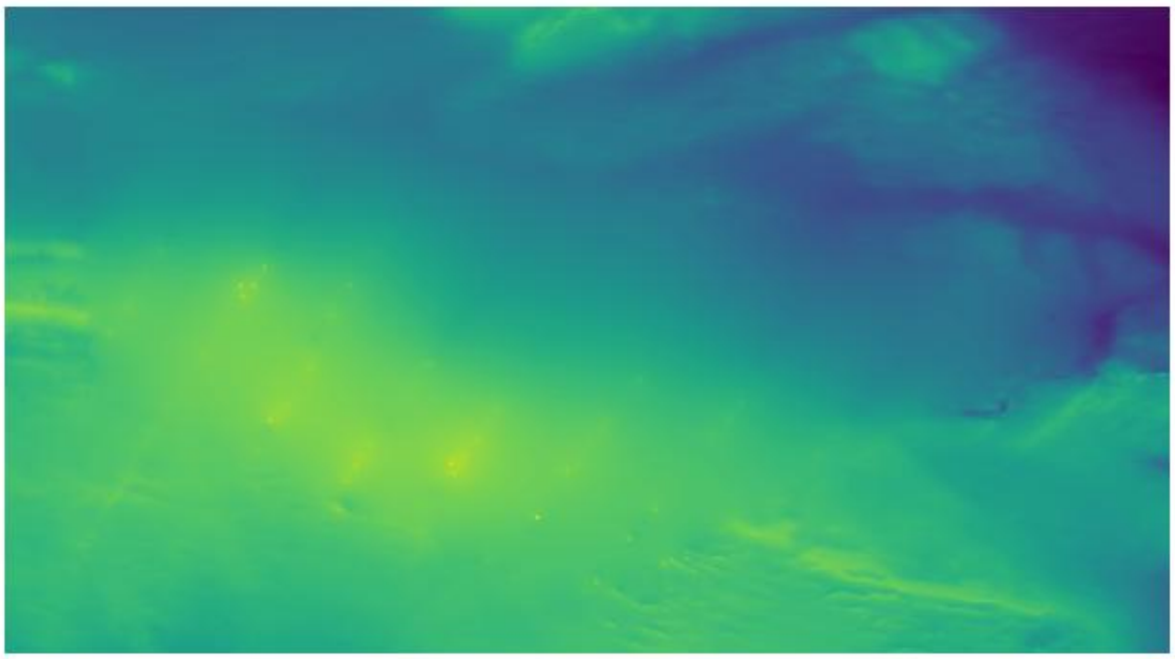

Figure 9. Example dataset interpolated using the Total Variation inpainter (*tv* option in interpolate_netcdf4.py).

Minimizes the Total Variation formula within the area to inpaint:

$$\nabla(f(u_t)) = -div N_\epsilon(\nabla u_t)$$

Where:

$$N_\epsilon(u) = \frac{u}{\sqrt{\|u\|^2 + \epsilon^2}}$$

Intuitively, it tends to preserve/continue high gradients better than *harmonic*, since the evolution of the optimizer can be considered a type of anisotropic diffusion.

However, it will not take into account isolated points, and should only be used for filling gaps with no data fully surrounded with reference data.

Suitable for

- Filling continuous gaps of data (i.e., not suitable for scattered data interpolation).

Advantages

- Provides similar results to the *harmonic* inpainter, but tends to better preserve level lines of the surroundings.

Disadvantages

- Does not work well with sparsely sampled data: isolated data points will not contribute much to the interpolation.

Continous Curvature Splines in Tension (CCST) Inpainter

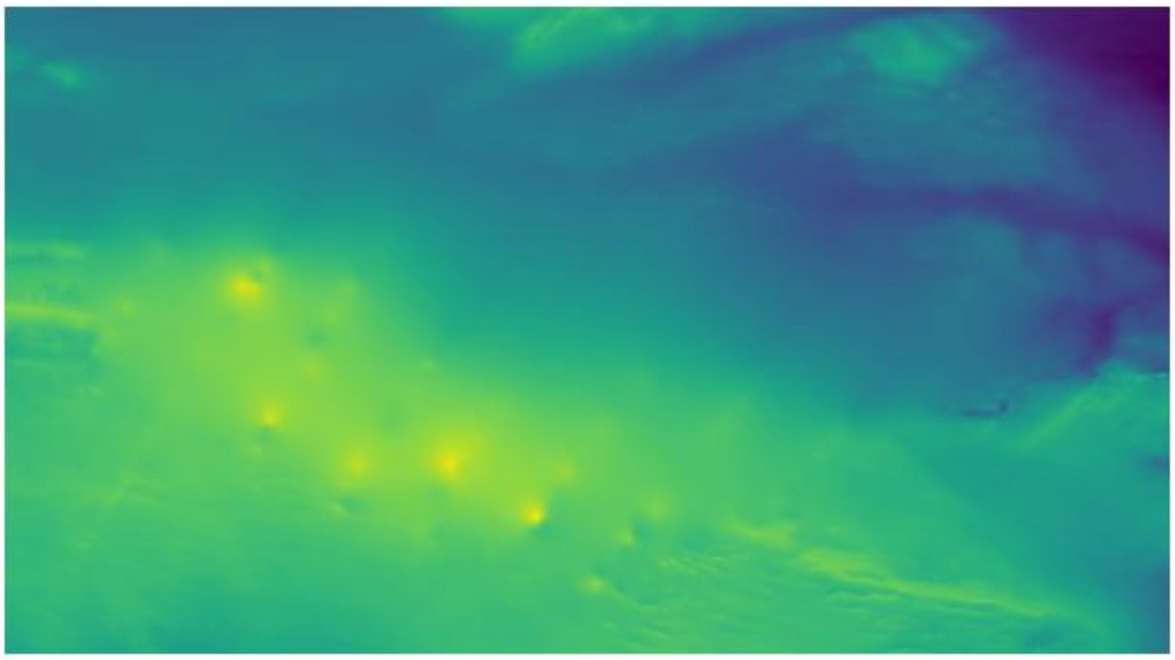

Figure 10. Example dataset interpolated using the Continous Curvature Splines in Tension inpainter (*ccst* option in interpolate_netcdf4.py).

Implements the method in [Smith90]. The PDE guiding this interpolant is the following:

$$\nabla(f(u_t)) = (1-t)\nabla^4 u_t - t\nabla^2 u_t = 0 \quad (1)$$

If we take a look to equation (1), we will identify that $\nabla^2 u_t$ is the harmonic equation (same as in Harmonic Inpainter). Also, in the other term, we find $\nabla^4 u_t = \nabla^2 \nabla^2 u_t$, the "harmonic of the harmonic", that is, the biharmonic surface. And, in both terms, they are affected by a constant t.

The *tension* parameter t allows tuning the influence of an harmonic and a biharmonic surface in the final result. Therefore:

- $t = 0$ equals a biharmonic surface.
- $t = 1$ equals an harmonic surface (same result as in Harmonic Inpainter).
- A value of t between 0 and 1 is a mixture of both harmonic/biharmonic interpolants.

In a nutshell, if we chop off the peak of a mountain at a given altitude, and we try to interpolate it using this method, $t = 0$ would probably reconstruct the peak of the mountain (note that this means that it will **overshoot** the input data), while $t = 1$ would reconstruct a flat area. A t between 0 and 1 would be a mix of both results.

Note that this is a re-implementation/variant of the method in [Smith90], which in turn is the method implemented in GMT surface.

[Smith90] Smith, W. H. F, and P. Wessel, 1990, Gridding with continuous curvature splines in tension, Geophysics, 55, 293-305.

Suitable for

- Getting a higher order interpolating surface, similar to what we achieve with the *purbf* method.
- Achieving **the same** results as using the *purbf* with a thin plate spline RBF (tension == 0) for datasets where the number of reference data points is much larger than the number of points to interpolate with smaller memory requirements and computational cost.

Advantages

- It provides an "easy to tune" mix of a harmonic and a biharmonic interpolant.

Disadvantages

- Slower execution time than other inpainters.
- Depending on the parameters, it may overshoot the data.

Absolutely Minimizing Lipschitz Extension (AMLE) Inpainter

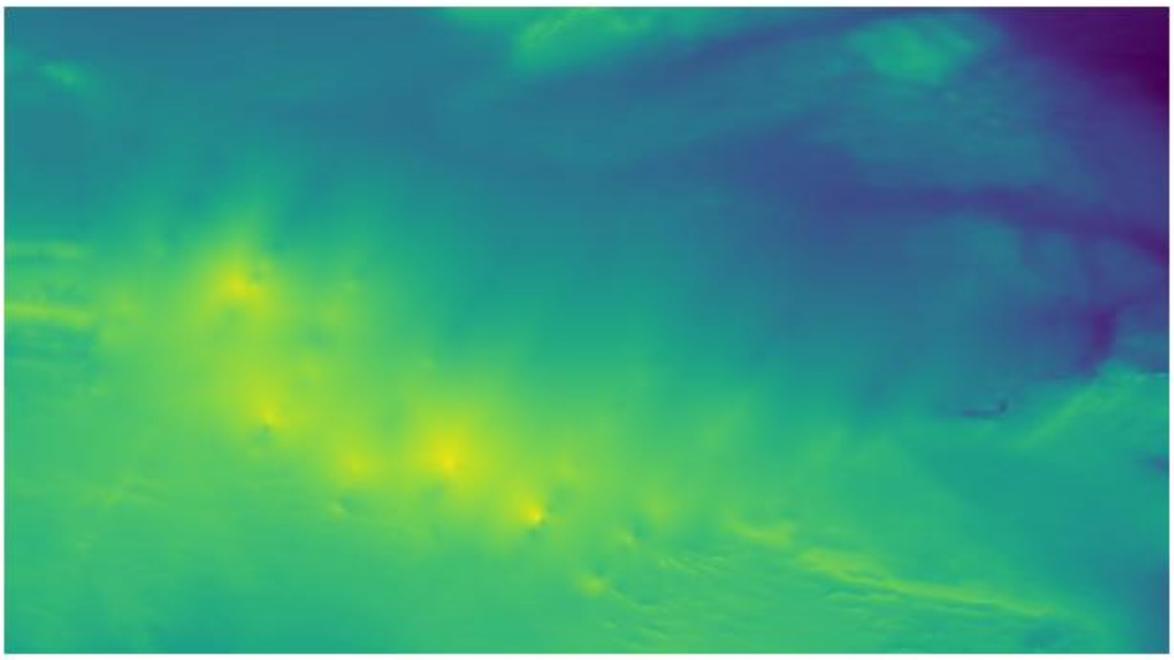

Figure 11. Example dataset interpolated using the Absolutely Minimizing Lipschitz Extension inpainter (*amle* option in interpolate_netcdf4.py).

Implements the method in [Almansa02]. Following the notation of the original reference, The PDE guiding this interpolant is the following:

$$\nabla(f(u_t)) = D^2 u_t \left(\frac{Du_t}{|Du_t|}, \frac{Du_t}{|Du_t|} \right)$$

Where Du denotes the gradient of u.

The main effort of the AMLE model is to “avoid oscillations”, i.e., to avoid the interpolated elevation to overshoot the reference values (min and max elevation value do not change). Also, it handles “isolated points'' in the reference data.

[Almansa02] Andrés Almansa, Frédéric Cao, Yann Gousseau, and Bernard Rougé. Interpolation of Digital Elevation Models Using AMLE and Related Methods. IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING, VOL. 40, NO. 2, FEBRUARY 2002

Suitable for

- Interpolating gaps in terrain data using a better interpolant, but trying not to overshoot the original data.
- Scattered data: this is the only approach that always takes into account scattered data properly (*ccst* with a tension apoaching 1 also does, but not so well if tension approaches 0…).

Advantages

- It is the only inpainter method in this package that was originally devised for interpolating heightmaps without overshooting the data.
- Contribution of isolated points is properly propagated within the area to interpolate.

Disadvantages

- Slower execution time than other inpainters.
- Depending on the dataset, it may require manual tuning of the update step size for the solver to converge.

5. Other Inpainters

Since one of the dependencies we use is OpenCV, and this library has some inpainting methods already implemented, we created interphases for using them on our heightmap interpolation problem. Note that these methods are typically used for closing small, thin gaps, as the ones you can see in the examples of the OpenCV documentation.

OpenCV's Telea

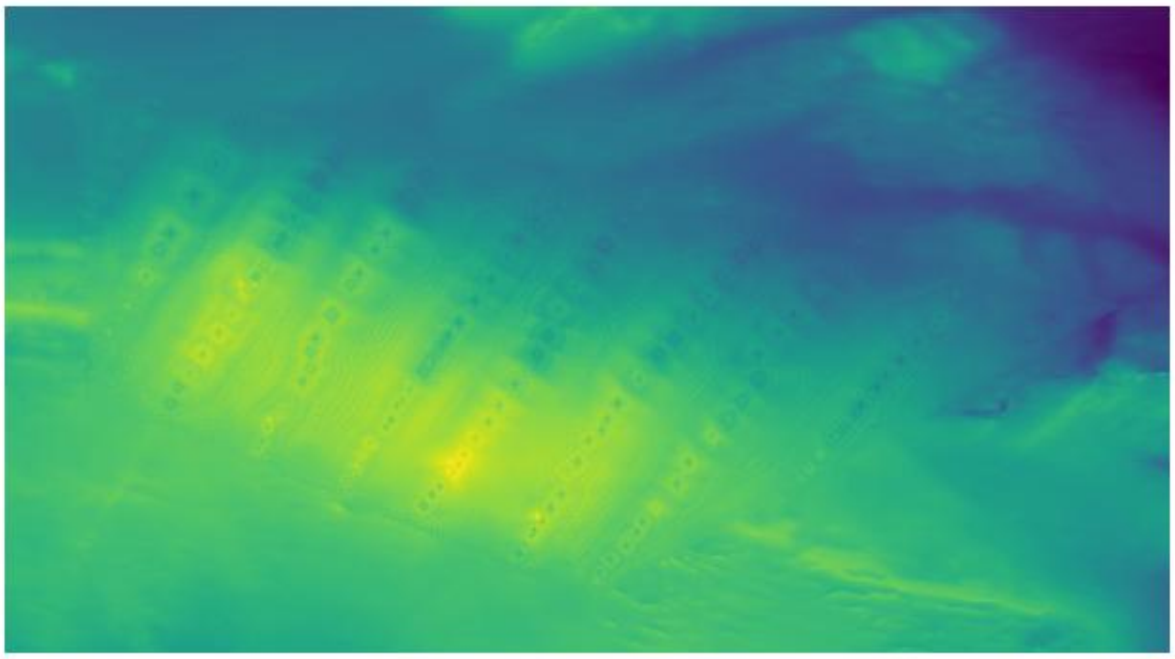

Figure 12. Example dataset interpolated using the OpenCV's Telea inpainter (*telea* option in interpolate_netcdf4.py).

The Telea variant of OpenCV's inpaint function.

Suitable for

- Interpolating “thin” continuous missing data parts fast.

Advantages

- Faster than PDE-based inpainters.

Disadvantages

- Does not consider scattered data at all.

OpenCV’s Navier-Stokes

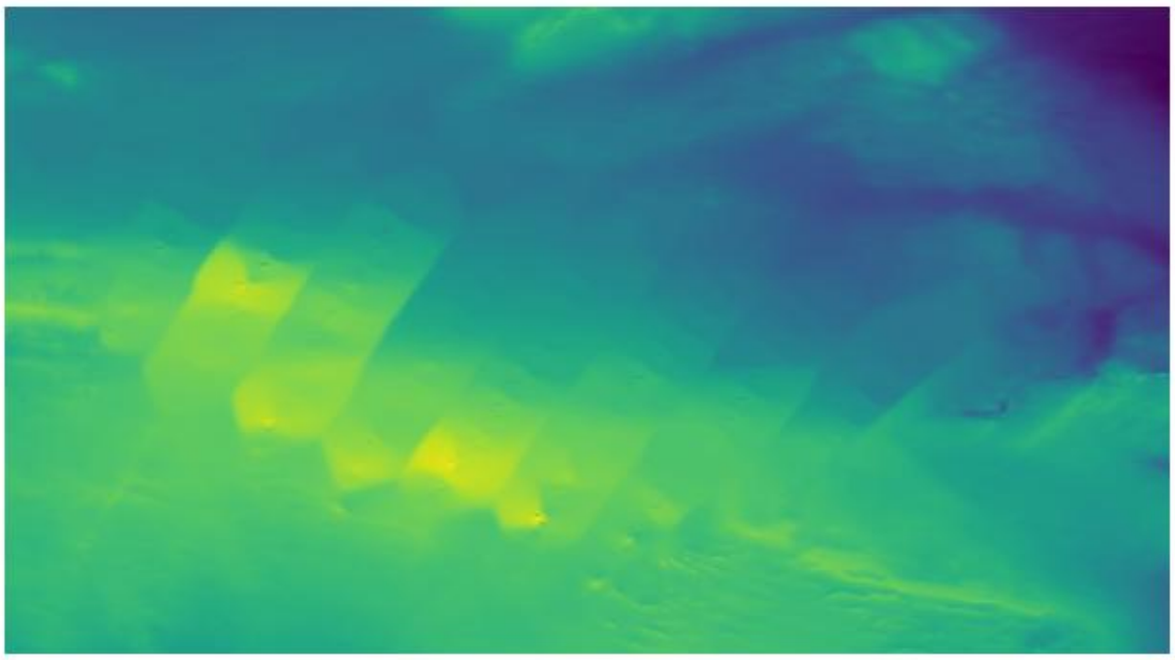

Figure 13. Example dataset interpolated using the OpenCV’s Navier-Stokes inpainter (*navier-stokes* option in interpolate_netcdf4.py).

The Navier-Stokes variant of OpenCV’s inpaint function.

Suitable for

- Interpolating “thin” continuous missing data parts fast.

Advantages

- Faster than PDE-based inpainters.

Disadvantages

- Does not consider scattered data at all.