New Interpolation Methods for Synthetization and Prediction

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com

Version 2.0, January 2023

Contents

1 Introduction

I describe little-known original interpolation methods with applications to real-life datasets. These simple techniques are easy to implement and can be used for regression or prediction. They offer an alternative to model-based statistical methods. Applications include interpolating ocean tides at Dublin, predicting temperatures in the Chicago area with geospatial data, and a problem in astronomy: planet alignments and frequency of these events. In one example, the 5-min data can be replaced by 80-min measurements, with the 5-min increments reconstructed via interpolation, without noticeable loss. Thus, my algorithm can be used for data compression.

The first technique has strong ties to Fourier methods. In addition to the above applications, I show how it can be used to efficiently interpolate complex mathematical functions such as Bessel and Riemann zeta. For those familiar with MATLAB or Mathematica, this is an opportunity to play with the MPmath library in Python and see how it compares with the traditional tools in this context. In the process, I also show how the methodology can be used to generate synthetic data [Wiki], be it time series or geospatial data.

Depending on the parameters, in the geospatial context, the interpolation is either close to nearest-neighbor methods, kriging [Wiki] (also known as Gaussian process regression), or a truly original and hybrid mix of additive and multiplicative techniques. There is an option not to interpolate at locations far away from the training set, where regression or interpolation results may be meaningless, regardless of the technique used.

The second technique is based on ordinary least squares – the same method used to solve polynomial regression – but instead of highly unstable polynomials leading to overfitting, I focus on generic functions that avoid these pitfalls, using an iterative greedy algorithm [Wiki] to find the optimum. In particular, a solution based on orthogonal functions leads to a particularly simple implementation with a direct solution.

2 First method

The general principle is simple. We want to interpolate a function $g(t)$ at certain points $t = \rho_1, \rho_2, \ldots$ belonging to a set R called the root set. These points are the roots of some function ψ. We create a function $w(t, \rho)$ which is equal to zero only if $t = \rho$ and $\rho \in R$. The functions ψ and w are chosen so that when $t \to \rho \in R$, the limit $\psi(t)/w(t, \rho)$ – a quotient where both the numerator and denominator are zero – exists and is different from zero. The limit in question is denoted as $\lambda(\rho)$. The interpolated function, denoted as $f(t)$ and defined by (1), is by construction identical to $g(t)$ when $t \in R$. This leads to the formulation

$$f(t) = \psi(t) \cdot \sum_{\rho \in R} \frac{f(\rho)}{\lambda(\rho)} \cdot \frac{1}{w(t,\rho)}, \quad \text{with } \lambda(\rho) = \lim_{t \to \rho} \frac{\psi(t)}{w(t,\rho)}. \tag{1}$$

Here $w(t, \rho) = 0$ if and only if $t = \rho$. The functions ψ and w must be chosen so that the limit in Formula (1) always exists and is different from zero. Typically, $w(t, \rho)$ measures how close t and ρ are to each other. If the summation is infinite and the series is conditionally convergent [Wiki] – as opposed to absolutely convergent – then the roots ρ need to be properly ordered. This is discussed in section 2.1. Convergence of the series may also require that $w(t, \rho) \to \infty$ fast enough as $|\rho| \to \infty$ and t is fixed.

In one dimension, the limit can be computed using l'Hôpital's rule [Wiki]:

$$\lambda(\rho) = \frac{\psi'(\rho)}{w'(\rho,\rho)}, \quad \text{with } \psi'(t) = \frac{\partial \psi(t)}{\partial t} \text{ and } w'(t,\rho) = \frac{\partial w(t,\rho)}{\partial t}.$$

Multiple applications of l'Hôpital's rule may be required for roots with multiplicity [Wiki]. The symbol ∂ stands for the partial derivative [Wiki], here with respect to t. In higher dimensions, the limit usually does not exist except under certain circumstances, see [2] and section 2.3.

2.1 Example with infinite summation

I start with some mathematics leading to interesting formulas. Then I use the formulas to interpolate time series, with a cool application. Let $\psi(t) = \sin \pi t$ and $R = \{\rho_0, \rho_1, \dots\} = \mathbb{N}$ so that $\rho_k = k$. With $w(t, \rho) = t^2 - \rho^2$, you get:

$$f(t) = \frac{\sin \pi t}{\pi} \cdot \left[\frac{f(0)}{t} + 2t \sum_{k=1}^{\infty} (-1)^k \frac{f(k)}{t^2 - k^2} \right]. \tag{2}$$

Formula (2) is valid for any even function f that can be written as

$$f(t) = \sum_{k=0}^{\infty} \alpha_k \cos \beta_k t \ \text{ with } |\beta_k| < \pi, \ \text{ or } f(t) = \int_{-\infty}^{\infty} \alpha(u) \cos(\beta(u)t) du \ \text{ with } |\beta(u)| < \pi. \tag{3}$$

Convergence and the fact that the left-hand side of (2) matches the right-hand side is rooted in the theory of Fourier series. For details, see here. A similar formula exists for odd functions. Note that f is even if $f(-t) = f(t)$, and f is odd if $f(-t) = -f(t)$. By combining the two formulas for odd and even functions, you get a formula that works for all functions regardless of parity. Again, limitations apply for convergence towards $f(t)$: the function f must be a sum of two terms, one involving cosines as in (3), and a similar one involving sines. In my general solution, you must replace π by $\pi/2$ in (3) – and the same in the sine term – for the generalized version of formula (2) to be valid.

The formula assumes that the interpolation nodes are integers. But a different grid could be used with a transformation such as $t' = a + bt$. You then interpolate f using known values of $f(t')$ where $(t' - a)/b$ is an integer, rather than interpolating f using known values of $f(t)$ where t is an integer. With an appropriate choice for a, you can extend the interpolation formula beyond the limitation previously discussed. For unevenly spaced nodes, use a non-linear mapping $\varphi(t)$ instead of $a + bt$. The function φ should be strictly monotonic, and thus invertible.

The Python implementation is in section 4.1, and available as `interpol_fourier.py` on my GitHub repository, here. With the linear transformation $a+bt$, I use nodes that are not integers to interpolate the math functions. I included advanced complex-valued math functions (Bessel, Riemann zeta) with complex arguments for those interested in scientific computing. It also illustrates how to use the mpmath library in Python, which is an alternative to Matlab.

Figure 1 shows the interpolation of the real part of the Riemann zeta function $\zeta(\sigma + it)$ [Wiki] on the critical line [Wiki], that is when $\sigma = \frac{1}{2}$. According to the famous Riemann Hypothesis, that's where all the non-trivial zeros lie. Actually, the first time I used my interpolation formula was in this context, with integer nodes. The approximation here is based on a more granular grid, and more accurate. See details in my book "Synthetic Data and Generative AI", and chapter 15 in particular [1].

Interestingly, this function looks quite similar to many real-life time series: in the end, it just a special combination of sine and cosine terms with various amplitudes and incompatible periods. It is thus a good candidate for time series synthetization, able to mimic many real examples by choosing the right interval and mapping for t. The ocean tide data and the distance between Earth and Venus (section 2.2) fit in that category, though they involve a small number of terms (the number is infinite for ζ).

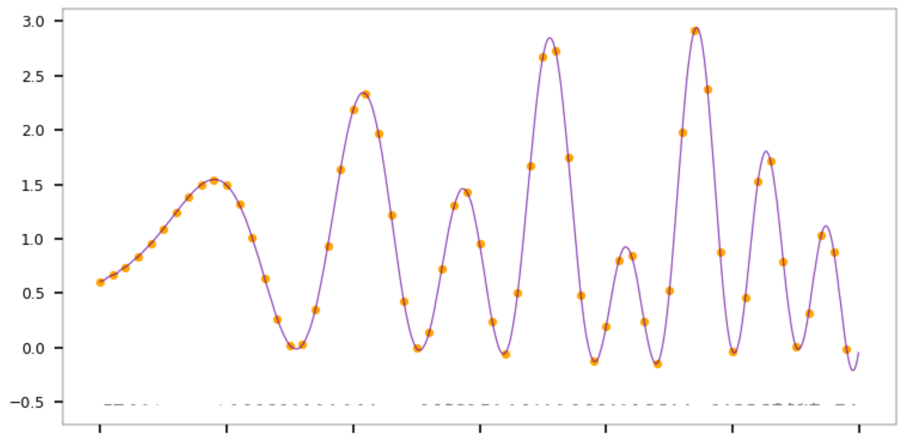

Figure 1: Interpolating the real part of $\zeta(\frac{1}{2}+it)$ based on orange points

2.2 Applications: ocean tides, planet alignment

The framework introduced in section 2.1 and the corresponding Python code can handle both math functions and time series datasets. In the code, I use the notation `g` for the exact function. The interpolated version is denoted as `interpolate`. Each interpolated value $f(t)$ is based on $2n+1$ integer nodes where the exact value $g(t)$ is assumed to be known, on both sides of t on the horizontal axis representing the time (left and right).

Here the dataset (time series) consists of ocean tides at Dublin, measured in 5-min increments. The small data extract `tides_Dublin.txt` is on my GitHub repository, here. You can download the full version here, on DigitalOcean.ie. I used it to test the methodology, since tides are easy to forecast without any statistical model. The conclusions are as follows:

- You only need data in 80-min increments to reconstruct the 5-min time series. So you can compress the dataset by a factor 16 (keeping a small fraction of it), almost without loss of information.
- The 5-min data and the 5-min interpolated values are very close to each other. When they differ, the interpolated values seem better than the observed ones. The interpolation removes the noise.
- What I did is known as time series disaggregation in the literature. It is useful to recover unobserved 5-min pollution levels and rainfall data from hourly observations, and in other contexts.
- What I did also amounts to generating 16 shifted subsets of 80-min interpolated tides, the shift being 5 minutes. Each subset is a synthetic dataset in itself. It would be very useful if the 5-min data was not known, offering various synthetic copies of the 80-min dataset. The reason I did it despite the fact that the 5-min data is known, is to test the accuracy of the interpolated values. All of them (regardless of time) were generated using one single subset of real observations with 80-min increments, as if no intermediate values were known.

The dataset is stored in the table `temp`. It is equivalent to an array where the index, rather than being an integer, is a multiple of 1/16. For that reason, I used a dictionary rather than an array in Python. Interpolated values are computed using observations where the index – representing the time – is an integer (one out of 16 observations). This is also true for interpolating the math functions in section 2.1. Finally, each interpolated value is based on $2n+1$ nodes (exact values where the index is an integer) with $n = 8$, spread over a $(8+8)\times 80$ minutes time period (that is, about 21 hours). By design, every 80 minutes – when the index is an integer – the interpolated and exact values are perfectly identical: this corresponds to the orange dots in Figure 2.

In Figure 2, the red curve represents the observed (exact) 5-min tides. The blue curve represents the interpolated values. Except in a few instances, they are indistinguishable to the naked eye. The small black bars at the bottom represents the error, in absolute value. The same black bars are found in Figure 1 related to the Riemann zeta function. However in that case the error is so small the the minuscule bars look like a glitch in the picture.

Note that no statistical model is involved in my method. It is still possible to compute various confidence or prediction intervals for the interpolated values, using bootstrapping techniques. It is discussed at length in my articles and books, and in the literature in general. For a parametric model to predict ocean tides, see the first chapter in my book on synthetic data [1].

The whole method can be seen as a regression technique to predict values within the range of observations, for time series or for observations ordered in a certain way (here by time). In some sense, it is a model-free regression that uses only one feature: the time. Can it be generalized to handle multiple features, or in other words, multivariate data? The answer is yes. See section 2.3 and 2.4 for an application to temperature interpolation, based on two features: longitude and latitude.

Exercise 1 – When are planets aligned?

We first create a dataset with daily measurements of the distance between Earth and Venus, and interpolate the distance to test how little data is needed for good enough performance: can you reconstruct daily data from monthly observations? What about quarterly or yearly observations? Then, the purpose is to assess how a specific class of models is good at synthetizing not only this type of data, but at the same time other types of datasets like the ocean tides in Figure 2 or the Riemann zeta function in Figure 1.

The planetary fact sheet published by the NASA contains all the information needed. It is available here. I picked up Venus and Earth because they are among the planets with the lowest eccentricities in the solar system. For simplicity, assume that the two orbits are circular. Also assume that at a time denoted as $t = 0$, the Sun, Venus and Earth were aligned and on the same side (with Venus between Earth and the Sun).

Note that all the major planets revolve around the sun in the same direction. Let $\theta_V, \theta_E, R_V, R_E$ be respectively the orbital periods of Venus and Earth, and the distances from Sun for Venus and Earth. From the NASA table, these quantities are respectively 224.7 days, 365.2 days, 108.2×10^6 km, and 149.6×10^6 km. Let $d_V(t)$ be the distance at time t, between Earth and Venus. You first need to convert the orbitial periods into angular velocities $\omega_V = 2\pi/\theta_V$ and $\omega_E = 2\pi/\theta_E$ per day. Then elementary trigonometry leads to the formula

$$d_V^2(t) = R_E^2\left[1 + \left(\frac{R_V}{R_E}\right)^2 - 2\frac{R_V}{R_E}\cos\Big((\omega_V - \omega_E)t\Big)\right]. \tag{4}$$

The distance is thus periodic, and minimum and equal to $R_E - R_V$ when $(\omega_V - \omega_E)t$ is a multiple of 2π. This happens roughly every 584 days.

Steps to complete

The exercise consists of the following steps:

Step 1: Use formula (4) to generate daily values of $d_V(t)$, for 10 consecutive years, starting at $t = 0$.

Step 2: Use the Python code in section 4.1 applied to your data. Interpolate daily data using one out of every 30 observations. Conclude whether or not using one measurement per month is good enough to reconstruct the daily observations. See how many nodes (the variable `n` in the code) you need to get a decent interpolation.

Step 3: Add planet Mars. The three planets (Venus, Earth, Mars) are aligned with the sun and on the same side when both $(\omega_V - \omega_E)t$ and $(\omega_M - \omega_E)t$ are almost exact multiples of 2π, that is, when both the distance $d_M(t)$ between Earth and Mars, and $d_V(t)$ between Earth and Venus, are minimum. In short, it happens when $g(t) = d_V(t) + d_M(t)$ is minimum. Assume it happened at $t = 0$. Plot the function $g(t)$, for a period of time long enough to see a global minimum (thus, corresponding to an alignment). Here ω_M is the angular velocity of Mars, and its orbit is approximated by a circle.

Step 4: Repeat steps 1 and 2 but this time for $g(t)$. Unlike $d_V(t)$, the function $g(t)$ is not periodic. Alternatively, use Jupiter instead of Venus, as this leads to alignments visible to the naked eye in the night sky: the apparent locations of the two planets coincide.

Step 5: A possible general model for this type of time series is

$$f(t) = \sum_{k=1}^{m} A_k \sin(\theta_k t + \varphi_k) + \sum_{k=1}^{m} A'_k \cos(\theta'_k t + \varphi'_k) \tag{5}$$

where the $A_k, A'_k, \theta_k, \theta'_k, \varphi_k, \varphi'_k$ are the parameters, representing amplitudes, frequencies and phases. Show that this parameter configuration is redundant: you can simplify while keeping the full modeling capability, by setting $\varphi_k = \varphi'_k = 0$ and re-parameterize. Hint: use the angle sum formula (Google it).

Step 6: Try 10^6 parameter configurations of the simplified model based on formula (5) with $\varphi_k = \varphi'_k = 0$, to synthetize time series via Monte-Carlo simulations. For each simulated time series, measure how close it is to the ocean tide data (obtained by setting `mode='Data'` in the Python code), the functions $g(t)$ and $d_V(t)$ in this exercise, and the Riemann zeta function pictured in Figure 1 (obtained by setting `mode='Math.Zeta'` in the Python code). Use a basic proximity metric of your choice to asses the quality of the fit, and use it on the transformed time series obtained after normalization (to get zero mean and unit variance). A possible comparison metric is a combination of lag-1, lag-2 and lag-3 auto-correlations.

Step 7: Because of the curse of dimensionality [Wiki], Monte-Carlo is a very poor technique here as we are dealing with 8 parameters. On the other hand, you can get very good approximations with just 4 parameters, with a lower risk of overfitting. Read section 1.3.3 in my book on synthetic data [1] about a better inference procedure, applied to ocean tides. Also read chapter 13 on synthetic universes featuring non-standard gravitation laws to generate different types of synthetic time series. Finally, read chapter 6 on shape generation and comparison: it features a different type of metric to measure the distance between two objects, in this case the time series (their shape: real versus synthetic version).

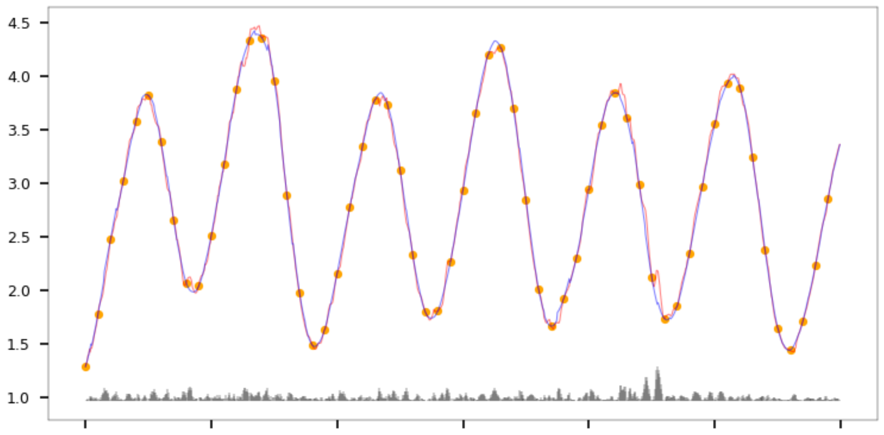

Figure 2: Tides at Dublin (5-min data), with 80 mins between interpolating nodes

2.3 Problem in two dimensions

In this section I discuss a basic example with a finite summation, where everything works nicely. However, it is a fundamental and very important case, as it applies to all regression problems. Also, it easily generalizes to higher dimensions. I use the notation $t = (x, y)$ and $z = f(t) = f(x, y)$. Let us assume that $\psi(x, y)$ has n roots $\rho_k = (x_k, y_k)$ with $k = 1, \dots, n$. The setting is as follows: we have a dataset with n observations (z_k, x_k, y_k) for $k = 1, \dots, n$. Here z_k is the response or dependent variable, and x_k, y_k are the two features, also called independent variables or predictors. I use

$$\psi(t) = \psi(x, y) = \prod_{k=1}^{n} w_k(x, y), \quad \lambda(\rho_k) = \lambda(x_k, y_k) = \prod_{i \neq k} w_i(x, y),$$

with the notation $w_k(x, y) = w(t, \rho_k) = w(x, y; x_k, y_k)$. I provide a specific example in formula (7). For now, let us keep in mind that by construction, $w(x, y; x', y') = 0$ if and only if $(x, y) = (x', y')$. It follows that

$$z = f(x, y) = \sum_{k=1}^{n} \gamma_k f(x_k, y_k), \quad \text{with } \gamma_k = \prod_{i \neq k} \frac{w_i(x, y)}{w_i(x_k, y_k)} = \prod_{i \neq k} \frac{w(x, y; x_i, y_i)}{w(x_k, y_k; x_i, y_i)}. \tag{6}$$

Thus, $z_k = f(x_k, y_k)$, for $k = 1, \dots, n$. Given a new observation (x, y), the predicted response z, based on the n data points in the training set, is provided by formula (6). If (x, y) is already in the training set, then the predicted z will be exact.

Unlike in traditional kernel-based methods, here the choice of the "distance" or "kernel" function w is critical. Some adaptations preserve the fact that $z_k = f(x_k, y_k)$ for $k = 1, \cdots, n$, while providing significantly better predictions and smoothness for observations outside the training set. This makes the method a suitable alternative to regression techniques. In particular, I implemented the following upgrades:

- Replacing γ_k by $\gamma'_k = \gamma_k/(1 + \gamma_k)$. It guarantees that these coefficients lie between 0 and 1.
- Replacing γ'_k by $\gamma^*_k = \gamma'_k / w_k^\kappa(x, y)$ where $\kappa \geq 0$ is an hyperparameter. This reduces the impact of the point (x_k, y_k) if it is too far away from (x, y).
- Normalizing γ^*_k so that their sum is equal to 1. This eliminates additive bias outside the training set.

These transformations make the technique somewhat hybrid: a combination of multiplicative, additive, and nearest neighbor methods. Further improvement is obtained by completely ignoring a point (x_k, y_k) when interpolating $f(x, y)$, if $w_k(x, y) > \delta$. Here $\delta > 0$ is an hyperparameter. It may result in the inability to make a prediction for a point (x, y) far away from all training set points: this is actually a desirable feature, not a defect.

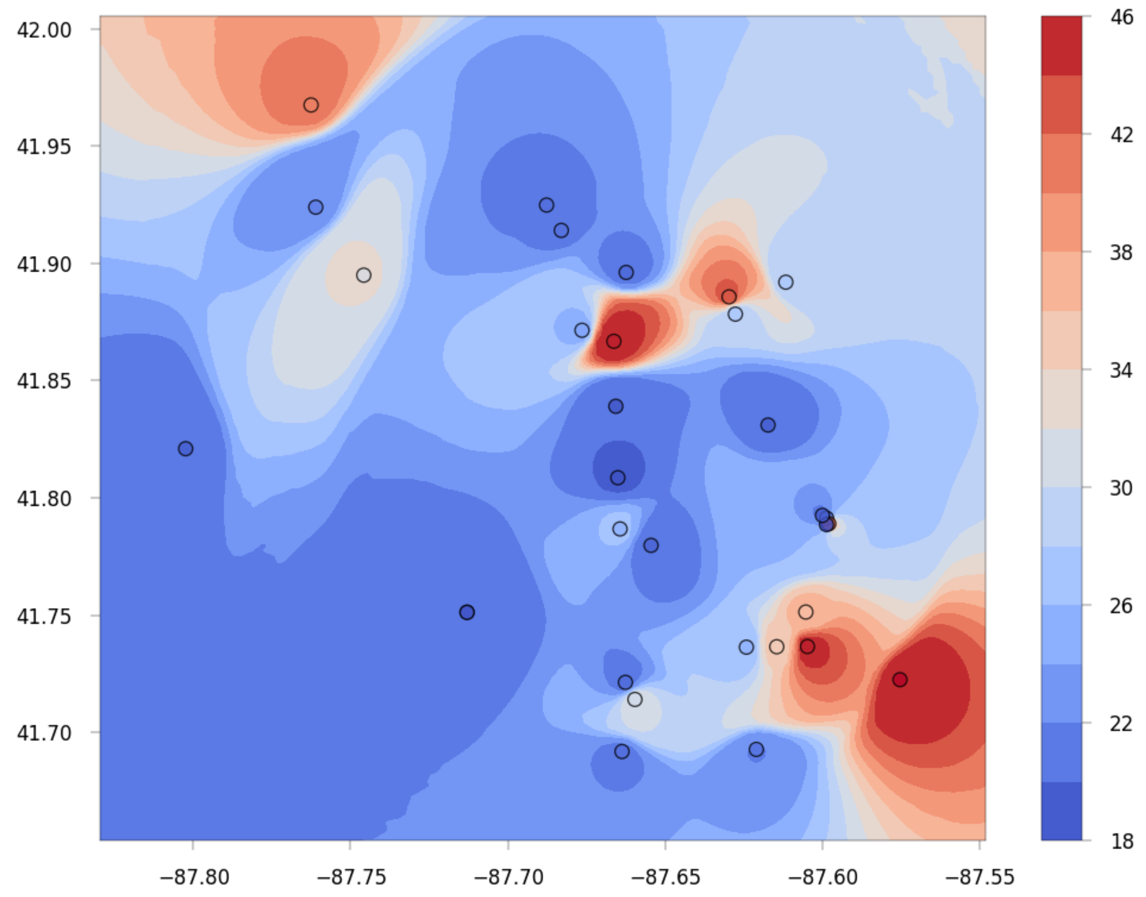

Figure 3: Temperature data: interpolation with my method (observed values at dots)

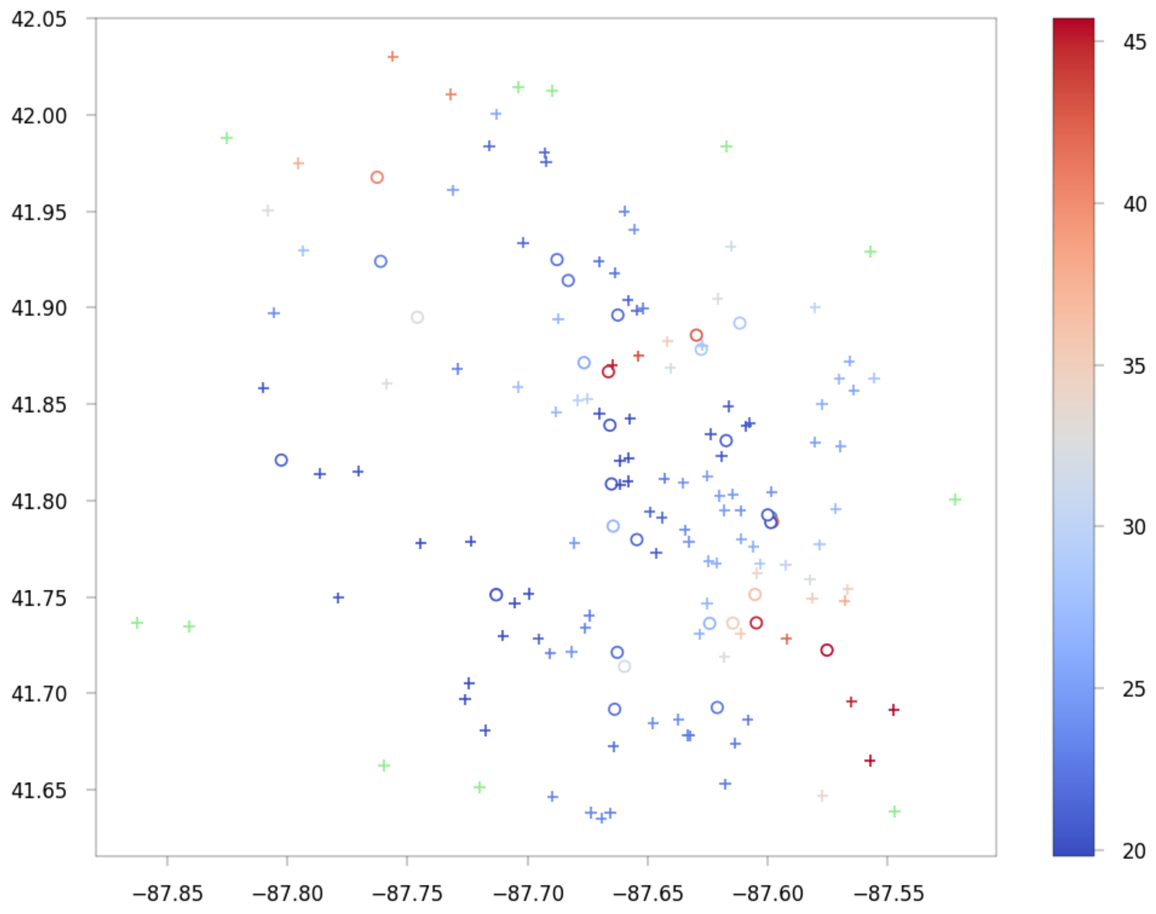

Figure 4: My method: round dots represent observed values, "+" are interpolated

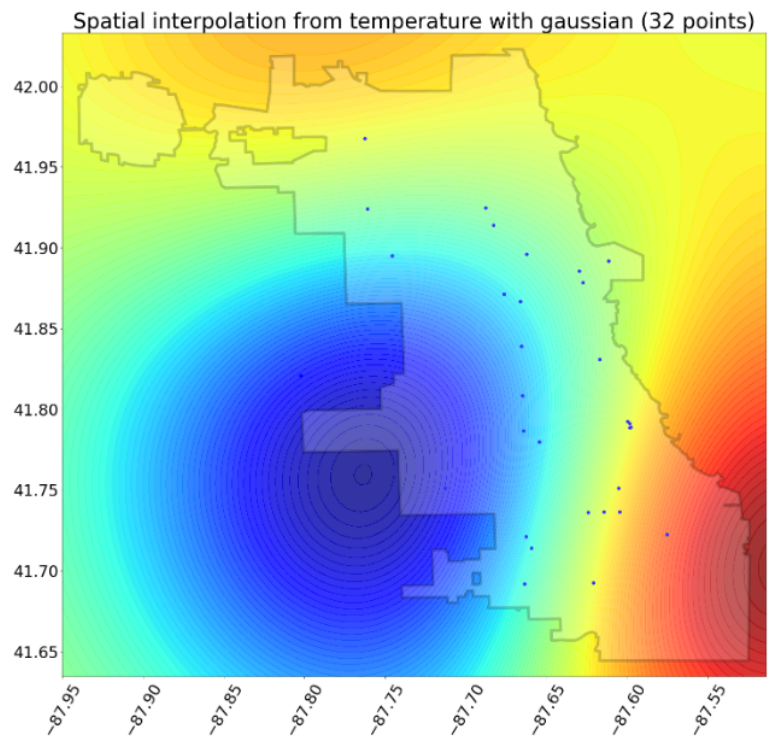

Figure 5: Temperature dataset: interpolation using ordinary kriging

2.4 Spatial interpolation of the temperature dataset

To test the method presented in section 2.3, I used the streaming high frequency temperature data in Chicago, retrieved from Array of Things. The data was analyzed here in 2019 using CyberGISX, a set of GIS tools [Wiki] developed in Python by the University of Illinois. They used ordinary kriging. The dataset has 3 fields: latitude (shown on the vertical axis), longitude (horizontal axis) and temperature (the color).

Figures 3 and 5 show the results: my method versus ordinary kriging. The picture corresponding to kriging covers a larger area, with vast regions without training locations. This is extrapolation rather than interpolation, and the deep blue well and strong red dome are artifacts of the method. They are much less pronounced in my picture (Figure 3). Indeed, I had to force my technique to cover an area away from the training set, beyond what is reasonable, to avoid blank (non-interpolated) zones in my image.

Figure 4 used a smaller interpolation window: the green "+" are non-interpolated locations due to their distance to the training set. Interpolating so far away from the training set, without additional information on how the real data behaves (heat domes, cold atmospheric depressions) is meaningless and does not generalize to other fields. The 32 training set locations are represented by circular dots in all three pictures. In Figure 3, the color of these dots (exact temperature) does not match the color of the background (interpolated value on nearby location) despite the appearance. There are tiny differences, not visible to the naked eye: it proves that the method works! One location in the South East corner has four training set points in very close proximity, with vastly different temperatures (the red dot is almost hidden): there you can see that the interpolation averaged out the four temperatures in question.

For $w_k(x, y)$, I used the function defined by (7), with $\alpha = 1, \beta = 2$. I did not make any efforts to find ideal parameter values, as this would defeat the purpose of designing a generic algorithm that works in many settings with as little fine-tuning as possible. For the same reason, the parameter κ in section 2.3 is set to 2, and δ is automatically computed as the smallest value that guarantees all the tested locations can be interpolated.

$$w_k(x, y) = \Big(|x - x_k|^\beta + |y - y_k|^\beta\Big)^\alpha, \text{ with } \alpha, \beta > 0. \tag{7}$$

The parameters α, β control the smoothness of the interpolated function. Choosing a small value for δ amounts to using a nearest neighbor type of interpolation. Choosing a high value for κ amounts to performing kriging. Thus the method is eclectic and encompasses various types of interpolation. The Python implementation in section 4.2 follows best practices: the data is first normalized before interpolation, divisions by zero are properly handled, and you can choose not to interpolate at locations too far away from the training set by adjusting δ.

As seen in Figure 4, the Python code also generates 4 copies of the training set; the number of copies is specified by the variable `ppo` in the code. In each copy, the location of each point is uniformly distributed in a circle around the original training set location that it represents. The radius of that circle is determined by the variable `radius` in the code. This synthetic data is used to test the performance of the algorithm. It allows you to play with different values of the radius. The final line of code computes the average distance (temperature discrepancy) between exact values in the training set and the associated values in the synthetic data, interpolated at sampled locations.

3 Second method

So far I managed to hide the underlying mathematics quite well to make the presentation easier to understand, without limiting its depth. Here the mathematics are significantly more visible. Again, Fourier series make their apparition. I cover interpolation and regression in this section, both univariate and multivariate, not just for datasets but also for mathematical functions. There is no novelty in terms of mathematical research: the originality is in the angle took to present and use the methodology. It leads to remarkably simple, elegant, and robust multivariate regression methods.

3.1 From unstable polynomial to robust orthogonal regression

A classic approach to approximate a function is to use ordinary least squares [Wiki]. Indeed, regression techniques often rely, in one way or another, on this optimization technique. When the response is denoted as $f(x)$, and the feature vector denoted as x, the problem consists of finding coefficients $\alpha_1, \dots, \alpha_n$ such that

$$f_n(x) \equiv \sum_{k=0}^{n} \alpha_k p_k(x) \tag{8}$$

is the "best" approximation to $f(x)$. As in the time series application in section 2.2, the function f can be a mathematical function, or represent observed values in a dataset. In the latter case, f is known only for a finite number of arguments x corresponding to an entry in the data set. Either way, the problem consists of interpolating f. This is usually called a regression problem when dealing with real data, and the function f allows you to predict the response, given a new observation x not in the training set, via formula (8). Also, n is not the number of observations, but the index in an iterative loop associated with an optimization algorithm.

Traditionally, $p_k(x) = x^k$ in polynomial regression. In classic multivariate regression, $p_k(x)$ is the k-th component of the feature vector x, possibly after some transformation such as normalization, with $p_0(x) = 1$ corresponding to the intercept. Here, I am approaching the problem from a different angle. The idea is to build a sequence of functions (f_n) converging to f as $n \to \infty$, starting with $f_{-1}(x) = 0$. At iteration n, the function f_n is chosen to minimize

$$\delta(\alpha_n) \equiv \int_D \Big(f_n(x) - f(x)\Big)^2 dx = \int_D \Big(\alpha_n p_n(x) + f_{n-1}(x) - f(x)\Big)^2 dx, \tag{9}$$

where D is the domain where we want the approximation to be best. If f is known only for integer values or we are dealing with a dataset, the integral is replaced by a sum, and D is the discrete set of interpolation nodes (the integers), or the set of observed features in case of a dataset. It is convenient to introduce the following notations:

$$\beta_k = \int_D p_k(x) f(x) dx, \quad \gamma_k = \int_D p_k^2(x) dx, \quad \beta_{ij} = \int_D p_i(x) p_j(x) dx.$$

Then we have $\alpha_0 = \beta_0/\gamma_0$ and for $n > 0$:

$$\alpha_n = \frac{1}{\gamma_n}\Bigg[\beta_n - \sum_{k=0}^{n-1} \alpha_k \beta_{kn}\Bigg].$$

3.2 Using orthogonal functions

If the coefficients β_{ij} are all zero, the formulas considerably simplify. This is the case if (p_k) is a sequence of orthogonal functions [Wiki]. The most well-known example is $p_{2k}(x) = \cos(k\pi x/L)$ combined with $p_{2k+1}(x) = \sin(k\pi x/L)$ for $k = 0, 1$ and so on. If $D = [-L, L]$, it corresponds to approximating or interpolating a periodic function f on D using its Fourier series [Wiki]. In this case, $\gamma_k = L$ if $k > 0$, with $\gamma_0 = 2L$.

Unfortunately, the polynomials $p_x(x) = x^k$ are not orthogonal on any interval. However, there is a process called Gram-Schmidt orthogonalization [Wiki] that turns any sequence of linearly independent functions into orthogonal ones. When applied to $1, x, x^2, x^3$ and so on, it leads to Legendre polynomials [Wiki] for the $p_k(x)$'s. They are orthogonal on $D = [-1, 1]$.

As usual, rather than minimizing the distance between f and f_n in formula (9), it is possible to use a weighted distance. All the results can be adapted. In particular, many orthogonal functions involve a weight. For instance, the Hermite polynomials [Wiki] involve the weight $w(x) = \exp(-x^2/2)$, and satisfy

$$\int_{-\infty}^{\infty} p_i(x)p_j(x)w(x)dx = 0 \text{ if } i \neq j.$$

In this case, D is the entire real line, infinite in both directions. Finally, it is also possible to directly work with a discrete set D and discrete orthogonal functions [Wiki]. The integrals become sums as usual. The general framework related to all these concepts is the Sturm-Liouville theory [Wiki].

3.3 Application to regression

You can apply the Fourier series method to multivariate regression as follows. It works best with continuous features. For discrete features, I advise to look at discrete Fourier series [Wiki] instead. You want to transform each continuous feature separately so that the values are – as closely as possible – distributed uniformly on the interval $[-1, 1]$ after the transformation. This is accomplished in two steps: first apply the transformation F_k to the k-th component of the feature vector. Then apply the transformation Q. Do this for each component $k = 1, \dots, n$. Here

- F_k is the empirical distribution function [Wiki] attached to feature k. In Python, use the function `ECDF` from the statsmodel library.
- Q is the quantile function [Wiki] of a uniform distribution on $[-1, 1]$. Thus $Q(u) = -1 + 2u$.

Now use formula (8) on the transformed data, with the p_k and α_k from the first paragraph in section 3.2, together with $L = 1$. Note that there is no matrix inversion in this procedure. It is a particular type of spline regression [Wiki]. See Python code in section 4.3. For a different type of spline regression based on exact interpolation similar to the method discussed in section 2.3, see chapter 8 in my book on synthetic data [1].

For polynomial regression, use Legendre polynomials, after transforming the features so that values stay within $[-1, 1]$, and uniformly distributed. Again, there is no matrix inversion and the procedure is fast and simple. For an other example of regression with Legendre polynomials, see [4].

The methods presented here have their roots in interpolating math functions, where successive terms in the summation formula – in this case formula (8) – have on average a decreasing impact. Otherwise, the sum, usually involving an infinite number of terms, would not converge. So it makes sense to order the features by decreasing importance. The first feature with coefficient α_1 may be chosen as the one with least residual error. The second feature with coefficient α_2 being the one yielding the best improvement to the residual error, and so on.

Also, for the same reason, in multivariate regression, the method is suited for datasets with a large n (number of features) well approximated by model (8), even with a small number of observations, see [5]. Such datasets are sometimes referred to as wide data. A more general version uses multidimensional Fourier series [3].

4 Python code

This section contains the code, both for the time series interpolation, and the geospatial temperature dataset. The time series version deals with the ocean tide dataset as well as interpolating advanced math functions (with complex values and complex arguments) using the MPmath library. You can use the code for your own datasets, and for synthetization purposes. In addition, I added some minimal code for the multivariate regression based on Fourier series.

4.1 Time series interpolation

This program deals with the interpolation method in one dimension. The code is also on my GitHub repository, here. For parameter description, see sections 2.1 and 2.2. The code can interpolate a math function or a time series (dataset with observations ordered by time, with fixed time increments) depending on the `mode` parameter. Either way, the "object" to be interpolated (function or data) is represented by the function `g` in the code. The program computes non-linear moving averages, to interpolate the value of $g(t)$ at fractional arguments of the time, say $t = 1/16, 2/16, 3/16$ and so on, when the value is known only for integer arguments.

When t is an integer, the interpolated and observed values are identical. In the case of math functions, under certain general conditions, the interpolated values are also exact when the number of nodes (determined by variable n) is infinite. In practice, very good approximations are obtained already with $n = 8$.

The parameter $1/16$ is represented by the variable incr in the code. You can change it to the inverse of a power of two, say $1/8, 1/4$ or $1/2$. Other values such as $1/7$ may cause problems: due to computer arithmetic, the instruction 7*1/7 or 9*1/9 does not return an exact integer; however 8*1/8 does. It is easy to correct this issue if you need to.

Finally, one way to reduce the number of operations is to use a hash table (dictionary in Python) to store values of $g(t)$ each time a new t is encountered. Due to the moving average, the same value $g(t)$ may be computed multiple times on different occasions. The hash table will avoid double computations, and can save time especially when computing the Zeta function.

```python
# interpol_fourier.py (author: MLTechniques.com)
import numpy as np
import mpmath
import matplotlib as mpl
from matplotlib import pyplot as plt

# https://www.digitalocean.ie/Data/DownloadTideData

mode = 'Data' # options: 'Data', 'Math.Bessel', 'Math.Zeta'

#--- read data

if mode == 'Data':

   # one column: observed value
   # time is generated by the algorithm; integer for interpolation nodes

   IN = open("tides_Dublin.txt","r")
   table = IN.readlines()
   IN.close()

   temp={}
   t = 0
   # t/t_unit is an integer every t_unit observations (node)
   t_unit = 16 # use 16 for ocean tides, 32 for planet data discussed in the classroom
   for string in table:
      string = string.replace('\n', '')
      fields = string.split('\t')
      temp[t/t_unit] = float(fields[0])
      t = t + 1
   nobs = len(temp)

else:
   t_unit = 16

#--- function to interpolate

def g(t):
   if mode == 'Data':
      z = temp[t]
   elif mode == 'Math.Bessel':
      t = 40*(t-t_min)/(t_max-t_min)
      z = mpmath.besselj(1,t)
      z = float(z.real) # real part of the complex-valued function
   elif mode == 'Math.Zeta':
      t = 4 + 40*(t-t_min)/(t_max-t_min)
      z = mpmath.zeta(complex(0.5,t))
      z = float(z.real) # real part of the complex-valued function
   return(z)

#--- interpolation function
```

```
def interpolate(t, eps):
   sum = 0
   t_0 = int(t + 0.5) # closest interpolation node to t
   pi2 = 2/np.pi
   flag1 = -1
   flag2 = -1
   for k in range(0, n):
      # use nodes k1, k2 in interpolation formula
      k1 = t_0 + k
      k2 = t_0 - k
      tt = t - t_0
      if k != 0:
         if k %2 == 0:
            z = g(k1) + g(k2)
            if abs(tt**2 - k**2) > eps:
               term = flag1 * tt*z*pi2 * np.sin(tt/pi2) / (tt**2 - k**2)
            else:
               # use limit as tt --> k
               term = z/2
            flag1 = -flag1
         else:
            z = g(k1) - g(k2)
            if abs(tt**2 - k**2) > eps:
               term = flag2 * tt*z*pi2 * np.cos(tt/pi2) / (tt**2 - k**2)
            else:
               # use limit as tt --> k
               term = z/2
            flag2 = -flag2
      else:
         z = g(k1)
         if abs(tt) > eps:
            term = z*pi2*np.sin(tt/pi2) / tt
         else:
            # use limit as tt --> k (here k = 0)
            term = z
      sum += term
   return(sum)

#--- main loop and visualizations

n = 8
   # 2n+1 is number of nodes used in interpolation
   # in all 3 cases tested (data, math functions), n >= 8 works
if mode=='Data':
   # restrictions:
   #    t_min >= n, t_max <= int(nobs/t_unit - n)
   #    t_max > t_min, at least one node between t_min and t_max
   t_min = n # interpolate between t_min and t_max
   t_max = int(nobs/t_unit - n) # must have t_max - t_min > 0
else:
   t_min = 0
   t_max = 100
incr = 1/t_unit # time increment between nodes
eps  = 1.0e-12

OUT = open("interpol_tides_Dublin.txt","w")

time = []
ze = []
zi = []

fig = plt.figure(figsize=(6,3))
mpl.rcParams['axes.linewidth'] = 0.2
mpl.rc('xtick', labelsize=6)
mpl.rc('ytick', labelsize=6)
```

```
for t in np.arange(t_min, t_max, incr):
   time.append(t)
   z_interpol = interpolate(t, eps)
   z_exact = g(t)
   zi.append(z_interpol)
   ze.append(z_exact)
   error = abs(z_exact - z_interpol)
   if t == int(t):
      plt.scatter(t,z_exact,color='orange', s=6)
   print("t = %8.5f exact = %8.5f interpolated = %8.5f error = %8.5f %3d nodes" %
       (t,z_exact,z_interpol,error,n))
   OUT.write("%10.6f\t%10.6f\t%10.6f\t%10.6f\n" % (t,z_exact,z_interpol,error))
OUT.close()

plt.plot(time,ze,color='red',linewidth = 0.5, alpha=0.5)
plt.plot(time,zi,color='blue', linewidth = 0.5,alpha=0.5)
base = min(ze) - (max(ze) -min(ze))/10
for index in range(len(time)):
   # plot error bars showing delta between exact and interpolated values
   t = time[index]
   error = abs(zi[index]-ze[index])
   plt.vlines(t,base,base+error,color='black',linewidth=0.2)
plt.savefig('tides2.png', dpi=200)
plt.show()
```

4.2 Geospatial temperature dataset

The Python code `interpol.py` is also on my GitHub repository, here. The functions and parameters are described in sections 2.3 and 2.4. The main function, performing interpolation on a 2-dimensional grid applied to temperatures in the Chicago area, is rather simple. The data is stored into the `data` array, mapped to the `npdata` Numpy array. It is then mapped onto a grid, represented by the `zgrid` array. The grid is used only to produce contour plots.

Four copies of the training set are generated (using `ppo=4`). They can be viewed as four synthetized versions of the training set, with locations and temperatures distributed just like in the original training set. The synthetized locations are stored in the arrays `xa` and `ya` (latitude and longitude); the synthetized temperatures obtained by interpolation are stored in the array `za`.

Interpolated values computed on locations identical to a training set location are exact, by design. Note that before interpolating, the data is transformed: it is normalized to have zero mean and unit variance, a standard practice. It is de-normalized at to end to produce the contour plots. Each interpolated value is computed using a variable number of nodes. That number depends on how many nodes are close enough to the target location. A node is a location in the training set with known temperature.

The number of nodes, for each synthetized location, is stored in the `npt` array. Using the default parameter value for `alpha` guarantees that there is always at least one node (the nearest neighbor) to compute the interpolated value. This can lead to meaningless interpolated values for locations far away from the training set. Reducing the default `alpha` results in some non-interpolated values marked as `NaN`, and it is actually recommended. The un-computed values show up as a green "+" in Figure 4.

Finally, the `interpolate` function accepts locations `x`, `y` that are either a single location or an array of locations. Accordingly, the returned value `z` – the temperature – can be a single value or an array. The `audit` parameter is used internally for testing and monitoring purposes.

```
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
from matplotlib import colors
from matplotlib import cm # color maps

data = [
# (latitute, longitude, temperature)
# source = https://cybergisxhub.cigi.illinois.edu/notebook/spatial-interpolation/
(41.878377,-87.627678,28.24),
(41.751238,-87.712990,19.83),
(41.736314,-87.624179,26.17),
(41.722457,-87.575350,45.70),
(41.736495,-87.614529,35.07),
(41.751295,-87.605288,36.47),
(41.923996,-87.761072,22.45),
(41.866786,-87.666306,45.01), # 125.01 outlier changed to 45.01
(41.808594,-87.665048,19.82),
(41.786756,-87.664343,26.21),
(41.791329,-87.598677,22.04),
(41.751142,-87.712990,20.20),
(41.831070,-87.617298,20.50),
(41.788979,-87.597995,42.15),
(41.914094,-87.683022,21.67),
(41.871480,-87.676440,25.14),
(41.736593,-87.604759,45.01), # 125.01 outlier changed to 45.01
(41.896157,-87.662391,21.16),
(41.788608,-87.598713,19.50),
(41.924903,-87.687703,21.61),
(41.895005,-87.745817,32.03),
(41.892003,-87.611643,28.30),
(41.839066,-87.665685,20.11),
(41.967590,-87.762570,40.60),
(41.885750,-87.629690,42.80),
(41.714021,-87.659612,31.46),
(41.721301,-87.662630,21.35),
(41.692703,-87.621020,21.99),
(41.691803,-87.663723,21.62),
(41.779744,-87.654487,20.88),
(41.820972,-87.802435,20.55),
(41.792543,-87.600008,20.41)
]
npdata = np.array(data)

#--- top parameters

n = len(npdata) # number of points in data set
ppo = 4       # create ppo new points around each observed point
new_obs = n * ppo
alpha = 1.0   # small alpha increases smoothing
beta = 2.0    # small beta increases smoothing
kappa = 2.0   # high kappa makes method close to kriging
eps  = 1.0e-8 # make it work if sample locations same as observed ones
np.random.seed(6)
radius = 1.2
audit = True  # so log monitoring info about the interpolation

xa = []            # latitute
ya = []            # longitude
da = []            # dist between observed and interpolated value
zd = []            # observed z
za = np.empty(new_obs) # interpolated z

#--- transform data: normalization

mu = npdata.mean(axis=0)
stdev = npdata.std(axis=0)
npdata = (npdata - mu)/stdev

#--- interpolation for sampled locations

def w(x, y, x_k, y_k, alpha, beta):
   # distance function
   z = (abs(x - x_k)**beta + abs(y - y_k)**beta)**alpha
   return(z)

# create random locations for interpolation purposes
for h in range(ppo):
   # sample points in a circle of radius "radius" around each obs
   xa = np.append(xa, npdata[:,0] + radius * np.random.uniform(-1, 1, n))
   ya = np.append(ya, npdata[:,1] + radius * np.random.uniform(-1, 1, n))
   da = np.append(da, w(xa[-n:],ya[-n:],npdata[:,0],npdata[:,1],alpha,beta))
   zd = np.append(zd, npdata[:,2])

delta = eps + max(da) # to ignore obs too far away from sampled point
npt = np.empty(new_obs) # number of points used for interpolation at location j

def interpolate(x, y, npdata, delta, audit):
   # compute interpolated z at location (x, y) based on npdata (observations)
   # also returns npoints, the number of data points used in the interpolation
   # data points (x_k, y_k) with w[(x,y), (x_k,y_k)] >= delta are ignored
   # note: (x, y) can be a location or an array of locations

   sum = 0.0
   sum_coeff = 0.0
   npoints = 0
   for k in range(n):
      x_k = npdata[k, 0]
      y_k = npdata[k, 1]
      z_k = npdata[k, 2]
      coeff = 1
      for i in range(n):
         x_i = npdata[i, 0]
         y_i = npdata[i, 1]
         if i != k:
            numerator = w(x, y, x_i, y_i, alpha, beta)
            denominator = w(x_k, y_k, x_i, y_i, alpha, beta)
            coeff *= numerator / (eps + denominator)
      dist = w(x, y, x_k, y_k, alpha, beta)
      if dist < delta:
         coeff = (eps + dist)**(-kappa) * coeff / (1 + coeff)
         sum_coeff += coeff
         npoints += 1
         if audit:
            OUT.write("%3d\t%3d\t%8.5f\t%8.5f\t%8.5f\n" % (j,k,z_k,coeff,dist))
      else:
         coeff = 0.0
      sum += z_k * coeff
   if npoints > 0:
      z = sum / sum_coeff
   else:
      z = 'NaN' # undefined
   return(z, npoints)

OUT=open("audit.txt","w") # output file for auditing / detecting issues
OUT.write("j\tk\tz_k\tcoeff\tdist\n")

for j in range(new_obs):
   (za[j], npt[j]) = interpolate(xa[j], ya[j], npdata, 0.5*delta, audit=True)

OUT.close()

#--- inverse transform (un-normalize) and visualizations

steps = 140 # to create grid with steps x steps points, to generate contours
xb = np.linspace(min(npdata[:,0])-0.50, max(npdata[:,0])+0.50, steps)
yb = np.linspace(min(npdata[:,1])-0.50, max(npdata[:,1])+0.50, steps)
xc = mu[0] + stdev[0] * xb
yc = mu[1] + stdev[1] * yb
xc, yc = np.meshgrid(xc, yc)
zgrid = np.empty(shape=(len(xb),len(yb)))

# create grid and get interpolated values at grid locations
for h in range(len(xb)):
   for k in range(len(yb)):
      x = xb[h]
      y = yb[k]
      (z, points) = interpolate(x, y, npdata, 2.2*delta, audit=False)
      if z == 'NaN':
         zgrid[h,k] = 'NaN'
      else:
         zgrid[h,k] = mu[2] + stdev[2] * z
zgridt = zgrid.transpose()

# inverse transform
xa = mu[0] + stdev[0] * xa
ya = mu[1] + stdev[1] * ya
za = mu[2] + stdev[2] * za
xb = mu[0] + stdev[0] * xb
yb = mu[1] + stdev[1] * yb
npdata = mu + stdev * npdata

def set_plt_params():
   # initialize visualizations
   fig = plt.figure(figsize =(4, 3), dpi=200)
   ax = fig.gca()
   plt.setp(ax.spines.values(), linewidth=0.1)
   ax.xaxis.set_tick_params(width=0.1)
   ax.yaxis.set_tick_params(width=0.1)
   ax.xaxis.set_tick_params(length=2)
   ax.yaxis.set_tick_params(length=2)
   ax.tick_params(axis='x', labelsize=4)
   ax.tick_params(axis='y', labelsize=4)
   plt.rc('xtick', labelsize=4)
   plt.rc('ytick', labelsize=4)
   plt.rcParams['axes.linewidth'] = 0.1
   return(fig,ax)

# contour plot
(fig, ax) = set_plt_params()
cs = plt.contourf(yc, xc, zgridt,cmap='coolwarm',levels=16)
cbar = plt.colorbar(cs)
cbar.ax.tick_params(width=0.1)
cbar.ax.tick_params(length=2)
plt.scatter(npdata[:,1], npdata[:,0], c=npdata[:,2], s=8, cmap=cm.coolwarm,
     edgecolors='black',linewidth=0.3,alpha=0.8)
plt.show()
plt.close()

# scatter plot
(fig, ax) = set_plt_params()
my_cmap = cm.get_cmap('coolwarm')
my_norm = colors.Normalize()
ec_colors = my_cmap(my_norm(npdata[:,2]))
plt.scatter(npdata[:,1], npdata[:,0], c='white', s=5, cmap=cm.coolwarm,
   edgecolors=ec_colors,linewidth=0.4)
sc=plt.scatter(ya[npt>0], xa[npt>0], c=za[npt>0], cmap=cm.coolwarm,
   marker='+',s=5,linewidth=0.4)

# show in green points not interpolated as they were too far away
plt.scatter(ya[npt==0], xa[npt==0], c='lightgreen', marker='+', s=5,
   linewidth=0.4)

cbar = plt.colorbar(sc)
cbar.ax.tick_params(width=0.1)
cbar.ax.tick_params(length=2)
# plt.ylim(min(npdata[:,0]),max(npdata[:,0]))
# plt.xlim(min(npdata[:,1]),max(npdata[:,1]))
plt.show()
```

```
#--- measuring quality of the fit

error = np.mean(abs(za[npt>0] - zd[npt>0]))
print("Error=",delta)
```

4.3 Regression with Fourier series

The basic code here is provided to illustrate the methodology in section 3.3, for multivariate regression with sine and cosine splines. It is a minimal workable piece of code. The data is made up, and no transformer is necessary because the observed values are already in $[-1, 1]$ for the feature vector, by construction. The code is also on my GitHub repository, here. Look for `interpol_ortho.py`.

```
import numpy as np
import random

#---- make up data

data = []
nobs = 100
random.seed(69)
for i in range(nobs):
   x1 = -1 + 2*random.random()  # feature 1
   x2 = -1 + 2*random.random()  # feature 2
   z  = np.sin(0.56*x1) - 0.5*np.cos(1.53*x2) # response
   obs = [x1, x2, z]
   data.append(obs)

npdata = np.array(data)
transf_npdata = npdata # no data transformer needed here

#--- the p_k functions

def p_k(x, k):

   # if input x is an array, output z is also an array

   if k % 2 == 0:
      z = np.cos(k*x*np.pi)
   else:
      z = np.sin(k*x*np.pi)
   return(z)

#--- beta_k, alpha_k, gamma_k coefficients

intercept = np.ones(nobs)
p_0 = p_k(intercept, k = 0)
p_1 = p_k(transf_npdata[:,0], k = 1) # feature 1
p_2 = p_k(transf_npdata[:,1], k = 2) # feature 2

gamma_0 = np.dot(p_0, p_0) # dot product
gamma_1 = np.dot(p_1, p_1)
gamma_2 = np.dot(p_2, p_2)

observed_temp = npdata[:,2]
beta_0 = np.dot(p_0, observed_temp)
beta_1 = np.dot(p_1, observed_temp)
beta_2 = np.dot(p_2, observed_temp)

alpha_0 = beta_0 / gamma_0
alpha_1 = beta_1 / gamma_1
alpha_2 = beta_2 / gamma_2

#--- interpolation
```

```
predicted_temp = alpha_0 * p_0 + alpha_1 * p_1 + alpha_2 * p_2

#--- print results: predicted vs observed

for i in range(nobs):
   print("%8.5f %8.5f" %(predicted_temp[i],observed_temp[i]))

correlmatrix = np.corrcoef(predicted_temp,observed_temp)
correlation = correlmatrix[0, 1]
print("corr between predicted/observed: %8.5f" % (correlation))

#--- interpolate for new observation (with intercept = 1)

x1 = 0.234
x2 = -0.541

z_predicted = alpha_0 * p_k(1,k=0) + alpha_1 * p_k(x1,k=1) + alpha_2 * p_k(x2,k=2)
print("test interpolation: z_predict = %8.5f" %(z_predicted))
```

References

[1] Vincent Granville. *Synthetic Data and Generative AI*. MLTechniques.com, 2022. [Link]. 2, 3, 4, 9

[2] Gary R. Lawlor. A l'Hospital's rule for multivariable functions. *Preprint*, pages 1–13, 2013. arXiv:1209.0363 [Link]. 2

[3] Alfred R.Osborne. Multidimensional Fourier series. *International Geophysics*, 97:115–145, 2010. [Link]. 9

[4] Mahesh Shivanand and all. Fitting random regression models with Legendre polynomial and B-spline to model the lactation curve for Indian dairy goat of semi-arid tropic. *Journal of Animal Breeding and Genetics*, pages 414–422, 2022. [Link]. 9

[5] Fengyun Wang and all. Bivariate Fourier-series-based prediction of surface residual stress fields using stresses of partial points. *Mathematics and Mechanics of Solids*, 2018. [Link]. 9