Project Draft note

Non-Gaussian and Gaussian Process for Regression

Fuad Hasan and Zijie (Ian) Liang

2025-11-10

1 Code Setup

Begin with project management:

```julia
using Pkg
lab_dir = dirname(@__FILE__)
Pkg.activate(lab_dir)
# Pkg.instantiate() # uncomment this the first time you run the lab to install
packages, then comment it back
```

Load all required packages:

```julia
using Distances # use to compute pairwise Euclidean distance
using Interpolations
using Distributions
using GaussianProcesses # for the useful syntax
using LaTeXStrings
using LinearAlgebra
using Optim
using Plots
using Random
```

2 Background and Reading

This is the draft note for our project. In this project we will discuss the important contents included the definition of regression, multiple Non-Gaussian regression methods, 1D Gaussian method for the regression, and 2D Gaussian methoed.

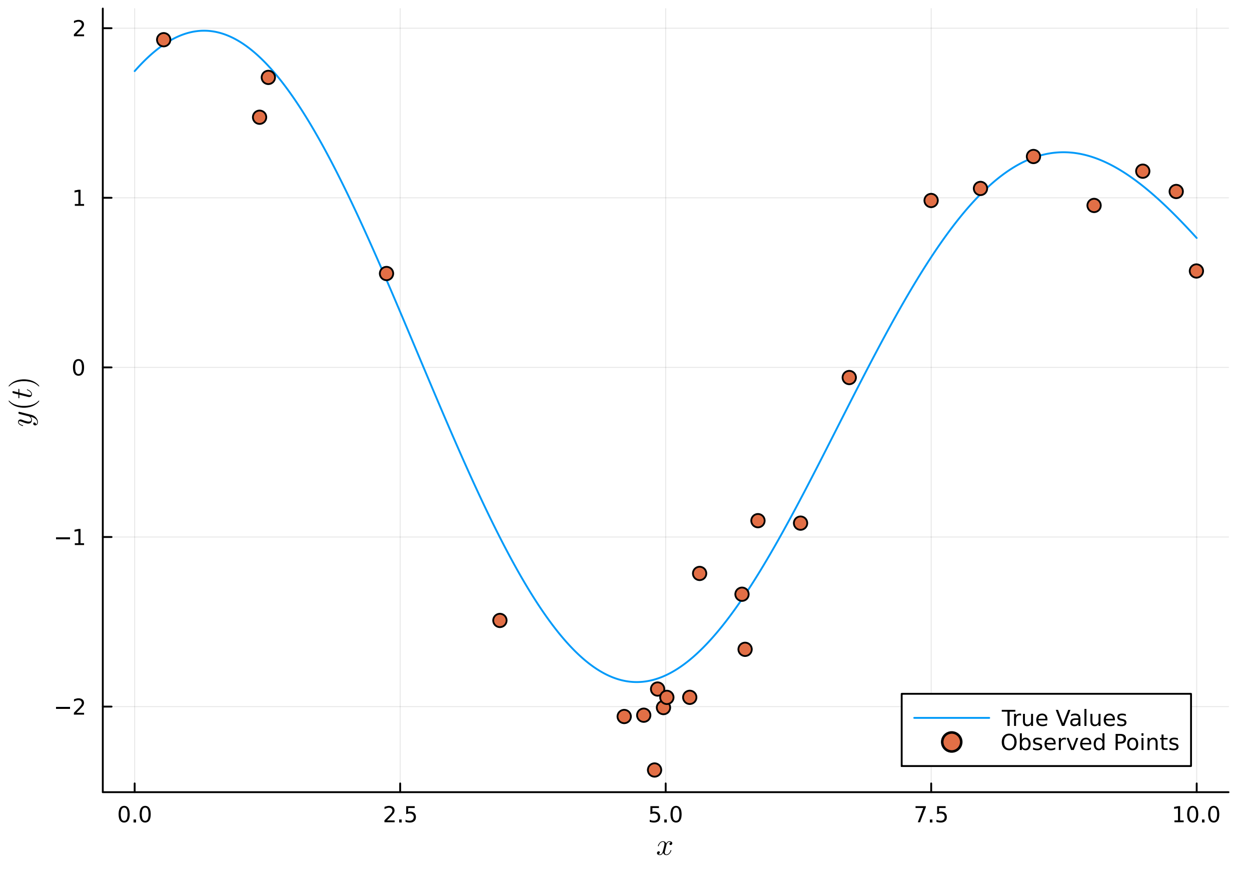
3 Regression Motivation

When testing the environmental data, sometimes we are interested in the nearby data. The normal operation is to used the observation to predict/simulate the surroundance data that we haven't observed. Take the following examples: 1. Rainfall: Only a few rain gauges cover a watershed, but we need the rainfall data for the whole watershed to drive hydrologic models; 2. Temperature or Air Quality: Satellite retrievals are patchy or have missing pixels so we need to fill gaps; 3. Urban Planning/Infrastructure: Sensor data (traffic, vibration, flood depth) are discrete but need continuuous spatial representation.

3.1 Example

With the help by the note from Dr. James Doss-Gollin's previous class J. Doss-Gollin [1], we generate some basic model as example to show the interpolation regression example and basic concept.

```
f(t) = @. sin(2 * pi * t / 7.2 + 0.9) + sin(2 * pi * t / 2/5 + 1.3)

begin
    N = 25              # number of observations
    σy = 0.25           # noise parameter
    x0 = 0              # lower bound
    x1 = 10             # upper bound
    Random.seed!(70005)               # set seed
    x = sort(rand(Uniform(x0, x1), N))      # points we observe at
    xprime = collect(range(x0, x1, length=1000))   # locations to estimate
    y = f.(x) .+ rand(Normal(0, σy), N)            # observed values with noise
    baseplot = plot(xlabel=L"$x$", ylabel=L"$y(t)$", legend=:bottomright)
end

p = deepcopy(baseplot)
plot!(p, xprime, f.(xprime), label="True Values")
scatter!(p, x, y, label="Observed Points")
```



Without the blue line, how do we predict the point between `x = 2.5` and `x = 5.0`? We could use interpolation to simulate the observation.

4 Non-Gaussian Process For Regression

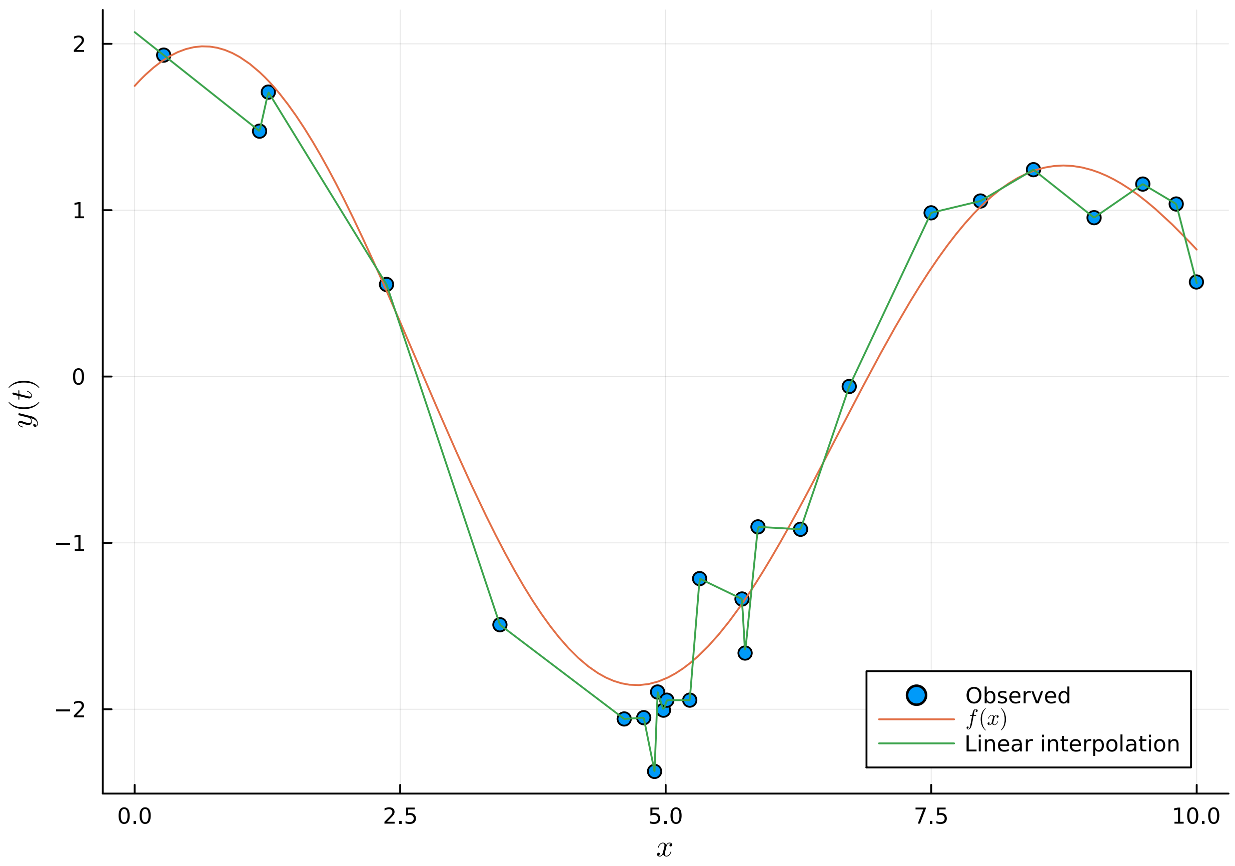
4.1 Interpolation

In order to do the prediction, the simplest way is to try a linear interpolation. Linear interpolation estimates the value of a function between two known data points by assuming the change between them is linear (a straight line). If the function is known at points (x_1, y_1) and (x_2, y_2), then for any x between them:

$$y(x) = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

The most important thing here is connecting the dots. If we have high density data, which are unbiased estimates, the model works well. However, if the data size is small, the predition will not perfom well C. Sun, N. Wang, S. Jin, and M. Wang [2].

```
itp_linear = LinearInterpolation(x, y; extrapolation_bc = Line())
f_linear(x) = itp_linear(x)
p = deepcopy(baseplot)
scatter!(p, x, y, label = "Observed")
plot!(p, f, x0, x1, label = L"$f(x)$")
plot!(f_linear, xprime, label = "Linear interpolation")
```



4.2 Inverse Distance Weighting Interpolation

Inverse Distance Weighting (IDW) is a simple and widely used spatial interpolation method G. Mei, N. Xu, and L. Xu [3]. It predicts the value at an unknown location as a `weighted average` of known data points, where weights depend on the `distance` between points — closer points have higher influence.Mathematically, the predicted value $\hat{z}(x_0)$ at location x_0 is given by:

$$\hat{z}(x_0) = \frac{\sum_{i=1}^{N} w_i \, z(x_i)}{\sum_{i=1}^{N} w_i}$$

where the weights w_i are defined as:

$$w_i = \frac{1}{d(x_0, x_i)^p}$$

Here:
- $z(x_i)$ = known value at location x_i
- $d(x_0, x_i)$ = distance between x_0 and x_i (usually Euclidean)
- p = power parameter controlling how quickly influence decreases with distance
- Small $p \longrightarrow$ smoother interpolation
- Large $p \longrightarrow$ closer points dominate

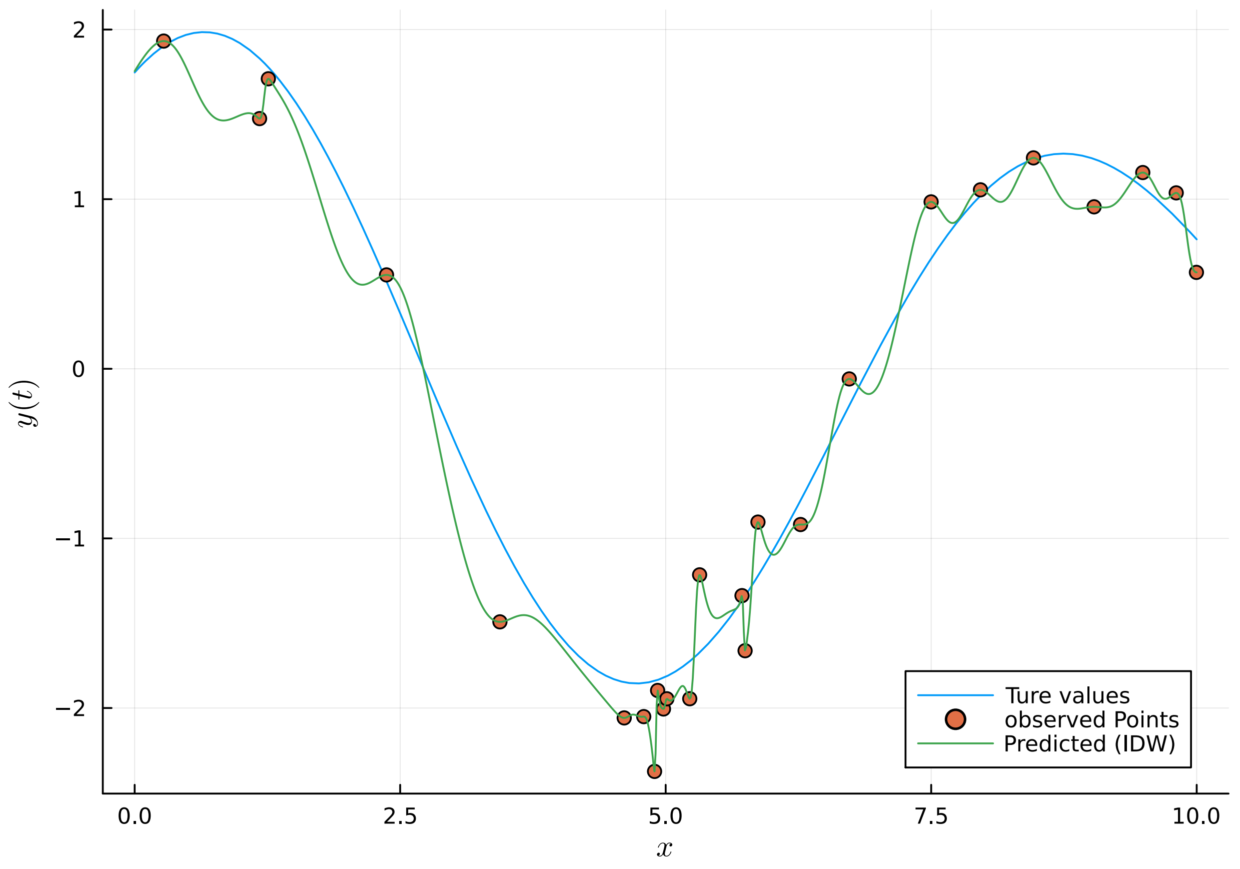
The weights are normalized so that they sum to 1:

$$\sum_{i=1}^{N} w_i = 1$$

IDW is deterministic and easy to compute, but it assumes the spatial relationship depends only on distance — it does not provide uncertainty estimates or account for directional trends.

```julia
calc_dist(x1, x2) = Distances.pairwise(Distances.Euclidean(), x1, x2)

function idw(x, y, xprime) where T <: Real
 dist = Distances.pairwise(Distances.Euclidean(), x, xprime)
 weights = dist .^ (-2)
 weights_norm = weights ./ sum(weights, dims = 1)
 return weights_norm' * y

end

yprime = idw(x, y, xprime)
p = deepcopy(baseplot)
plot!(p, f, x0, x1, label = "Ture values")
scatter!(p, x, y, label = "observed Points")
plot!(p, xprime, yprime, label = "Predicted (IDW)")
```

```
WARNING: method definition for idw at D:\Me\Study\Rice\PhD\Class\Statistical-Physical
Methods for Hydroclimate Extremes and Catastrophes\CEVE 543 labs\Project-draft-
note\Project Draft Note.qmd:166 declares type variable T but does not use it.
```



Since we have some good observations, the model prediction is really well. There are some weird vibration due to the high concentration of the data in a small area.

4.3 K Nearest Neighbors (KNN)

`K Nearest Neighbors (KNN)` is a variation of the Inverse Distance Weighting (IDW) method S. Zhang [4]. Instead of using *all* available data points, KNN considers only the `K closest points` around the prediction location. This helps reduce the influence of faraway points, making the interpolation more local and often more accurate.

Let $\hat{z}(x_0)$ be the predicted value at an unknown location x_0.
Among all N known points $(x_i, z(x_i))$, we first find the K nearest neighbors based on the distance $d(x_0, x_i)$.

The prediction is then computed as a weighted average over these K neighbors:

$$\hat{z}(x_0) = \frac{\sum_{i=1}^{K} w_i \, z(x_i)}{\sum_{i=1}^{K} w_i}$$

where the weights are defined using the same distance-decay principle as IDW:

$$w_i = \frac{1}{d(x_0, x_i)^p}$$

Here:
- $d(x_0, x_i)$ = distance between the prediction point and the i-th neighbor
- p = power parameter controlling the rate of distance decay
- K = number of nearest neighbors used for interpolation

By setting all weights outside the K nearest points to zero before normalization,
the KNN interpolation limits the effect of distant data, improving computational efficiency
and reducing unrealistic "global" influence in sparse datasets.

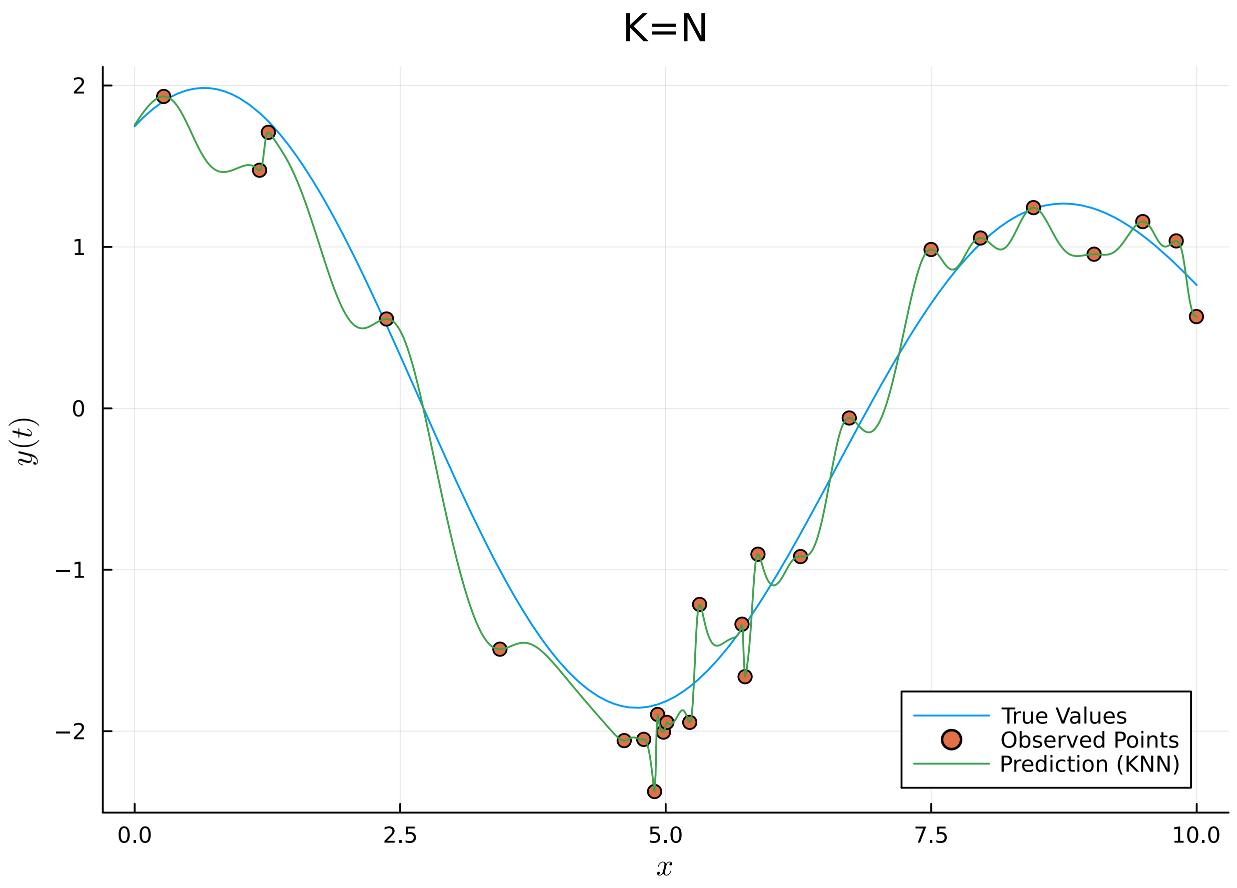
```
function knn(x, y, xprime, K) where T <: Real
 dist = Distances.pairwise(Distances.Euclidean(), x, xprime)
 weights = dist .^ (-2)

 # truncate some weights to zero
ranks = hcat([invperm(sortperm(col; rev = true)) for col in eachcol(weights)]...)
    weights[ranks .> K] .= 0

    # normalize
    weights_norm = weights ./ sum(weights, dims = 1)
    return weights_norm' * y

end
```

```
WARNING: method definition for knn at D:\Me\Study\Rice\PhD\Class\Statistical-Physical
Methods for Hydroclimate Extremes and Catastrophes\CEVE 543 labs\Project-draft-
note\Project Draft Note.qmd:214 declares type variable T but does not use it.
```

```
knn (generic function with 1 method)
```

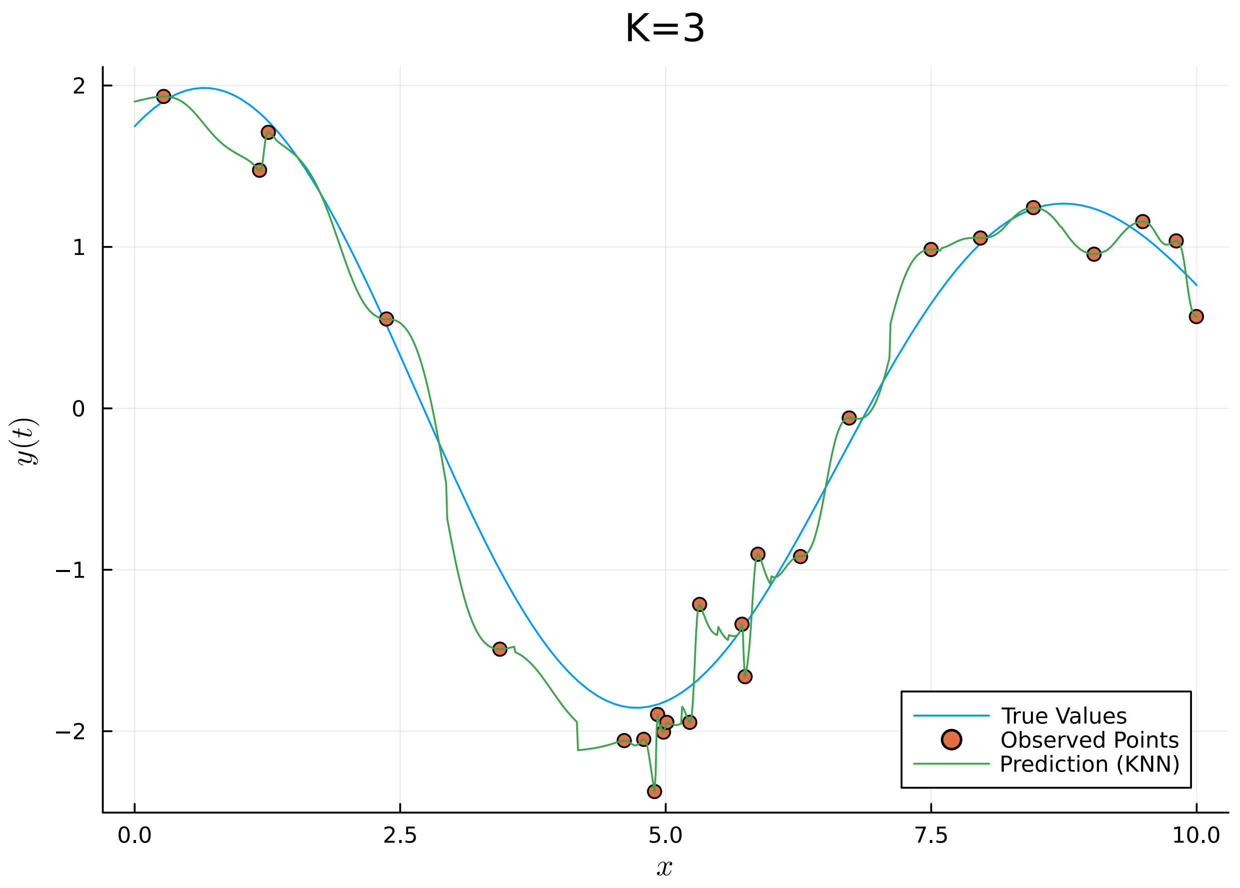
So now with $K = N$, the IDW estimation will be

```
K = N
yprime = knn(x, y, xprime, K)
p = deepcopy(baseplot)
plot!(p, f, x0, x1, label = "True Values", title = "K=N")
scatter!(p, x, y, label = "Observed Points")
plot!(p, xprime, yprime, label = "Prediction (KNN)")
```



What if we reduce the number of K, the estimation will be changed again

```
K = 3
yprime = knn(x, y, xprime, K)
p = deepcopy(baseplot)
plot!(p, f, x0, x1, label = "True Values", title = "K=$K")
scatter!(p, x, y, label = "Observed Points")
plot!(p, xprime, yprime, label = "Prediction (KNN)")
```



The prediction line looks wired when the `K` value changed. There are another method to run the interpolation like the high-order polynomial. However, one of the best way for the estimation is `Gaussian Processes`.

5 1D Gaussian Process

Until this portion, we have seen the non-gaussian process.The orange dots were the observed points that has obviously some noise(as say discharge measurement will have some noise, a discharge value we are saying 22, maybe 22.4) and blue line in the previous graphs was the true underlying function that can fit the observed data best(say fitting the discharge value). But, in reality we actually don't know how the blue line will look like. And also, in most regression methods, we assume that y = aX + b. The thing is that, all of these methods just give a single best-fit curve (green curves). Here, the Gaussian process comes handy. We can call it a Bayesian way of modeling nonlinear relationships. Before seeing the data (prior), many possible functions are considered. After seeing the data, the curves that pass close to the observed points receive higher likelihood, so the uncertainty shrinks near those points (posterior).

How Gaussian process can be useful?

Ans: Say with some real example, we have some air temperature readings in some stations. But we want to estimate the temperature between two stations. Or, say, we measure river discharge at some points at times t = 1, 3, and 5 hours. But we want to know what happened at t=2 and 4 hours(testing/prediction points)? In the first case, the nearby points will have a similar kind of temperature (spatial relation), and in case number 2, the nearby times will have a similar kind of discharge (temporal correlation).

So, in this kind of scenario, a Gaussian process can come in handy. If we have some observed data, and we want to predict what happened in unknown data points, this GP process can: 1. Naturally handle the spatial and temporal relationship. 2. Can provide smooth interpolation between known points. What do we understand from this? In general, the GP doesn't assume that there is a fixed equation like say y=aX+b, rather, they consider the function(function means it randomly take some curve from gaussian distribution) itself to be random, nearby points are correlated (determined through a kernel function), to smoothly predict what is happening between prediction/testing data points where we don't have any observation, and show how uncertain band of those predictions.

5.1 Steps of 1D Gaussian Process

5.1.a GP Prior Assumption

We assume the function $f(x)$ we're trying to learn is drawn from a **Gaussian Process**:

$$f(x) \sim \mathcal{GP}\ \big(\ m(x), K(x, x')\ \big)$$

where

- $m(x)$ is the **mean function**, often assumed constant (μ),
- $K(x, x')$ is the **covariance (kernel) function**, measuring similarity between inputs.

So, for any finite set of inputs

$$x = [x_1, ..., x_N], \quad x' = [x'_1, ..., x'_{N'}],$$

the **joint distribution** of function values is multivariate normal:

$$\begin{bmatrix} y' \\ y \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} a \\ b \end{bmatrix}, \begin{bmatrix} A & B \\ B^T & C \end{bmatrix} + \sigma_y^2 I\right)$$

5.1.b Meaning of Each Symbol

Symbol	Meaning	Size
y	training outputs	$N \times 1$
y'	test outputs	$N' \times 1$
$a = m(x')$	mean at test points	$N' \times 1$
$b = m(x)$	mean at training points	$N \times 1$
$A = K(x', x')$	test–test covariance	$N' \times N'$
$B = K(x', x)$	test–train covariance	$N' \times N$
$C = K(x, x)$	train–train covariance	$N \times N$
$\sigma_y^2 I$	observation-noise term	$N \times N$

5.2 Kernel Function

We use the **squared-exponential (RBF)** kernel:

$$K(x, x' \mid \sigma, \ell) = \sigma^2 \exp\left[-\frac{(x - x')^2}{2\ell^2}\right]$$

where:

- σ^2: **signal variance** (controls amplitude) - How high or low a curve can go.
- ℓ: **length scale** (controls smoothness - Small ℓ means wiggly function that change rapidly, large ℓ means smooth function changing slowly).

if x and x' are close enough, that means points will have high covariance, points that are far, will have low covariance.

We are only providing the formula of RBF kernal as it is widely used. But, here we are introducing some other types of kernals and their functionality very briefly.

There are different types of kernal.

(a) RBF (Squared Exponential kernal): Makes the function very smooth and used for natural process like (temperature, elevation etc.).

(b) Matern kernal: Allows more rough function or curve than the RBF, used for wind speed, river flow etc.

(c) Linear kernal: Assumes the function is basically a straight line, used for mostly trend patterns say growth trend.

(d) Periodic kernal: Creates repeating, cyclic functions, mostly used for tides.

(e) Polynomial kernal: Creates polynomial shaped function, example usage can be for population growth.

Another excellent thing about kernals are that, it is also possible to do kernals addition and multiplication, but we are not adding the details in this note. Some of the references links will be helpful for more details digiLab AI [5].

Kernel Parameters

Of course, we don't initially know what **σ (signal variance)** or **ℓ (length scale)** should be - we just guess some starting values. The Gaussian Process then uses the training data to automatically adjust these parameters so that the model best explains the observations. This process is called *optimization*: the GP tries different values of σ and ℓ, checking which ones make the data most likely under the model. The search continues until the fit stops improving. To keep the optimization stable and ensure both parameters remain positive, we usually optimize over **log(σ)** and **log(ℓ)** instead of their direct values. The final optimized σ and ℓ describe how variable and how smooth the data is, respectively.

5.3 Prediction (Posterior Distribution)

The Gaussian process can be compared to Bayesian concept. The GP says, "before seeing any data, I believe there are infinite number of curves/functions that can fit our data shaped by mean function (center) and kernel (smoothness)". Then we observe the temperature (likelihood) and the curves that passes close to observed points get high likelihood, and curves far away from the data gets low likelihood. And finally, for the posterior, we get final distribution over funtions with mean and uncertainty. The conditional (posterior) distribution of the **test outputs** `y′` given the **training outputs** `y` is **Gaussian**:

$$p(y' \mid y) = \mathcal{N}(m, S)$$

where:

$$m = a + B\left(C + \sigma_y^2 I\right)^{-1}(y - b)$$

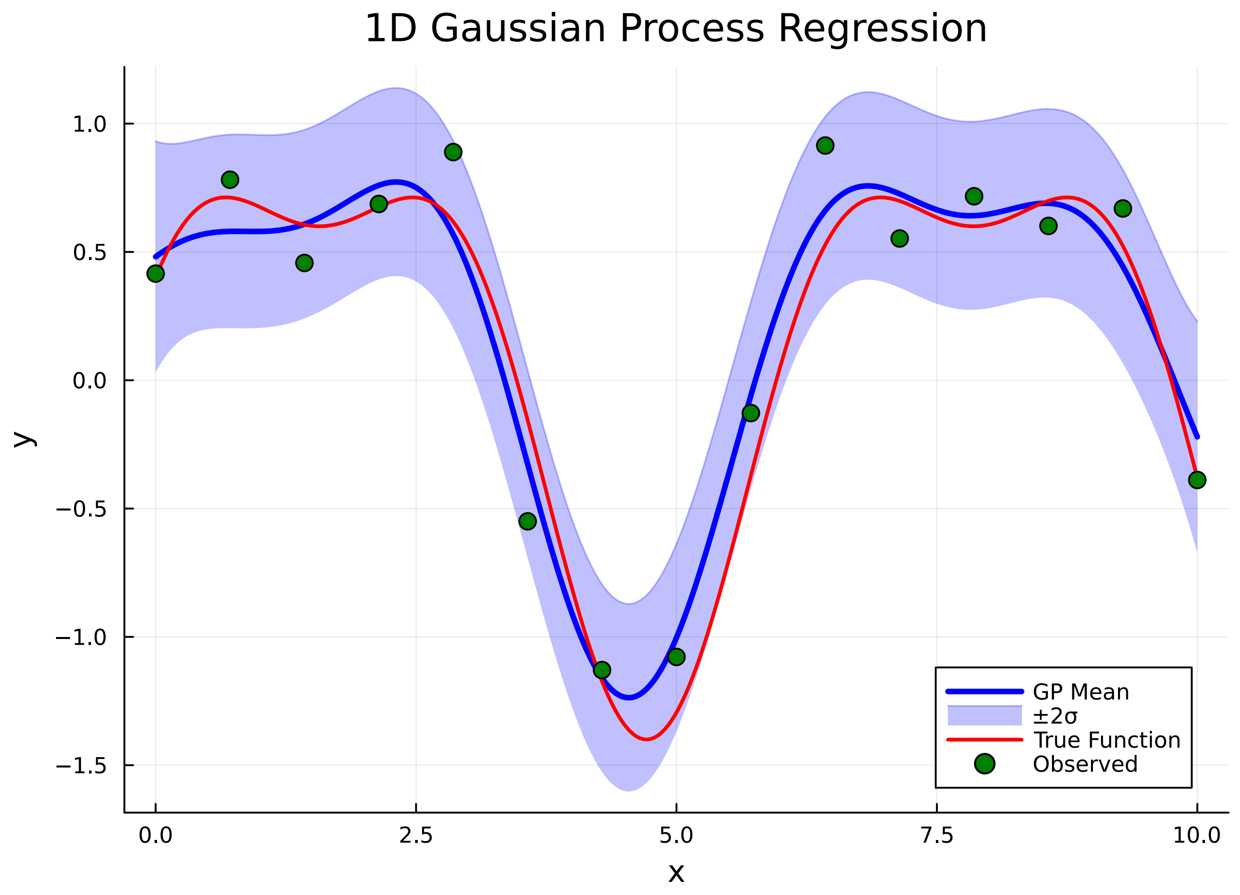
$$S = A - B\left(C + \sigma_y^2 I\right)^{-1}B^T$$

Here:

- **m** $\longrightarrow$ The **posterior mean** - it gives the predicted average function value at each test point.
- **S** $\longrightarrow$ The **posterior covariance matrix** - it gives the uncertainty between predicted points. (The diagonal of **S** determines the width of the shaded uncertainty region around the mean.)

In Gaussian Process, we assume the function has a constant mean (μ), meaning both training and testing points share the same average value. This helps the model focus on variations around this mean rather than the overall level.

```
# Squared-exponential kernel, let σ_f=1.0, ℓ=1.0

kernel(x1, x2; σ_f=1.0, ℓ=1.0) = σ_f^2 * exp(-(x1 - x2)^2 / (2ℓ^2))

# Build full kernel matrix for a vector
function kernel_matrix(x1, x2; σ_f=1.0, ℓ=1.0)
    K = zeros(length(x1), length(x2))
    for i in 1:length(x1)
        for j in 1:length(x2)
            K[i,j] = kernel(x1[i], x2[j]; σ_f=σ_f, ℓ=ℓ)
        end
    end
    return K
end

#  GP Prediction (y' | y)

function gp_predict(x, y, xnew; σ_f=1.0, ℓ=1.0, σ_n=0.1)
    # Build blocks A, B, C from GP theory
    A = kernel_matrix(xnew, xnew; σ_f=σ_f, ℓ=ℓ)          # M × M
    B = kernel_matrix(xnew, x;     σ_f=σ_f, ℓ=ℓ)          # M × N
    C = kernel_matrix(x,     x;     σ_f=σ_f, ℓ=ℓ)          # N × N

    # Add noise to C
    C += σ_n^2 * I

    # Posterior mean:  m = B * C^{-1} * y
    μ = B * (C \ y)

    # Posterior covariance: S = A - B C^{-1} Bᵀ
    Σ = A - B * (C \ B')

    # Uncertainty (std dev)
    σ = sqrt.(diag(Σ))
    return μ, σ
end


# Training data (observed)

Random.seed!(1)
f(x) = sin.(x) .+ 0.4*cos.(2x)                    # true function

x = collect(range(0, 10, length=15))
y = f(x) .+ 0.25 .* randn(length(x))              # noisy observations


# Prediction points (many)

xnew = collect(range(0, 10, length=300))

#  Run GP

μ, σ = gp_predict(x, y, xnew; σ_f=1.0, ℓ=1.2, σ_n=0.25)

#  Plot

plot(xnew, μ, lw=3, color=:blue, label="GP Mean")
plot!(xnew, μ .+ 2σ, fillrange=μ .- 2σ, color=:blue, alpha=0.25, label="±2σ")

plot!(xnew, f(xnew), lw=2, color=:red, label="True Function")
scatter!(x, y, color=:green, ms=5, label="Observed")

xlabel!("x")
ylabel!("y")
title!("1D Gaussian Process Regression")
```



Note: Why GP is called a non-parametric model?

Ans: Parametric model can have a fixed number of parameters, say for linear regression we have slope and intercept, always 2 parameters.Even if you have 100 or 1000 points, the slope and intercept are always two parameters. But, gaussian process learns the functions/curves from the data. If we give more data points, the function/curves become more detailed, covariance matrix gets bigger and prediction gets more accurate.

6 2D Gaussian Process

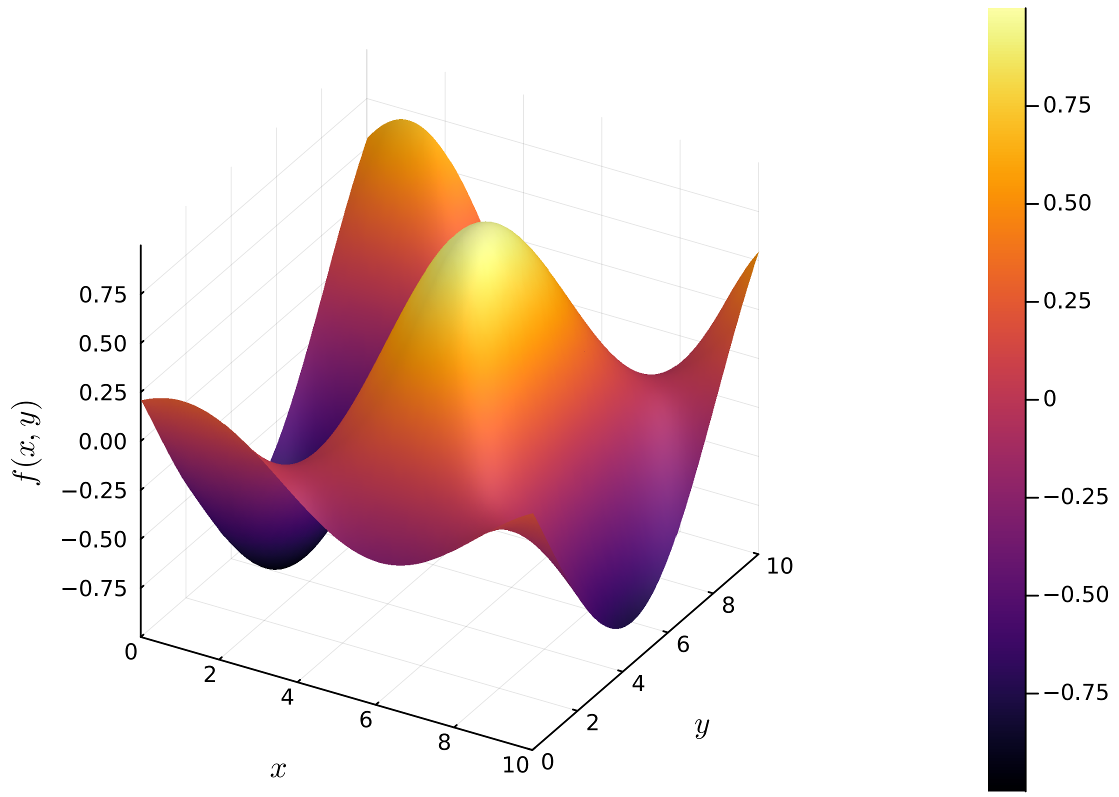
6.1 Introduction

When we try to analysis the Gaussian Process with the spatial data G. A. Achilleos [6], simple 1D method is not enough to help. The data now have elevation so the plot need to include z axis, which make the plot into 3D. In this way, 2D Gaussian Process is a good choice to use.

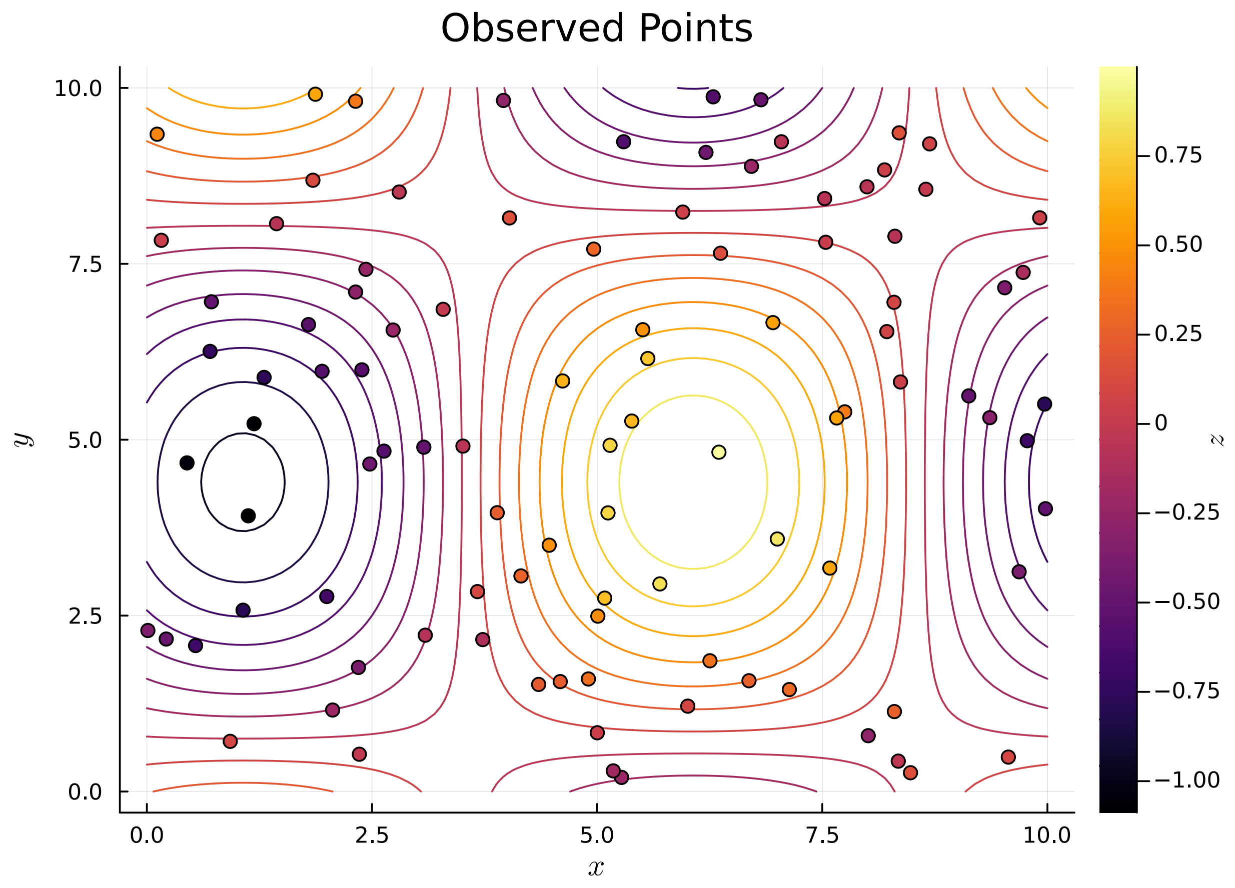
2D Gaussian Process is similar to 1D GP so we could have similar process 1. Choose a kernel depending on the relative of our data (2D) 2. Optimize the parameters of the kernel 3. Use the multivariate Normal to make predictions about the new data

Take the following examples:

```julia
f(x, y) = @. sin(2 * π * x / 10 + 0.9) * cos(2 * π * y / 15.0 + 1.3)
surface(0:0.1:10, 0:0.1:10, f; xlabel=L"$x$", ylabel=L"$y$", zlabel=L"$f(x,y)$")
```



```julia
begin
    N = 100 # number of observations
    σy = 0.125 # noise parameter
    xlims = (0, 10)
    ylims = (0, 10)
    Random.seed!(77005) # set seed
    x = rand(Uniform(xlims...), N) # points we observe at
    y = rand(Uniform(xlims...), N) # points we observe at
    z = f.(x, y) .+ rand(Normal(0, σy), N) # values we observe
    baseplot = plot(; xlabel=L"$x$", ylabel=L"$y$") # for plots
end

let
    p = deepcopy(baseplot)
    contour!(p, 0:0.1:10, 0:0.1:10, f)
    scatter!(p, x, y; zcolor=z, colorbar_title=L"$z$", title="Observed Points",
label=false)
end
```



6.2 Distance

Distance is an important aspect to consider when we do the GP. Euclidean distance is a simple method but this isn't always the right choice – for example, if we're working with spherical coordinates. There we might want to use something like Haversine distance, which measure the Great Circle distance between two points.

For now we'll stick to Euclidean distance for simplicity. Note that we specify `dims=1` so that the distances gives us the right shape (otherwise it wants to treat each column as an observation and each row as a feature, which is backwards)

```
calc_dist(x1, x2) = Distances.pairwise(Distances.Euclidean(), x1, x2; dims=1)
points = hcat(x, y)

D_x_x = calc_dist(points, points) # should be N x N
```

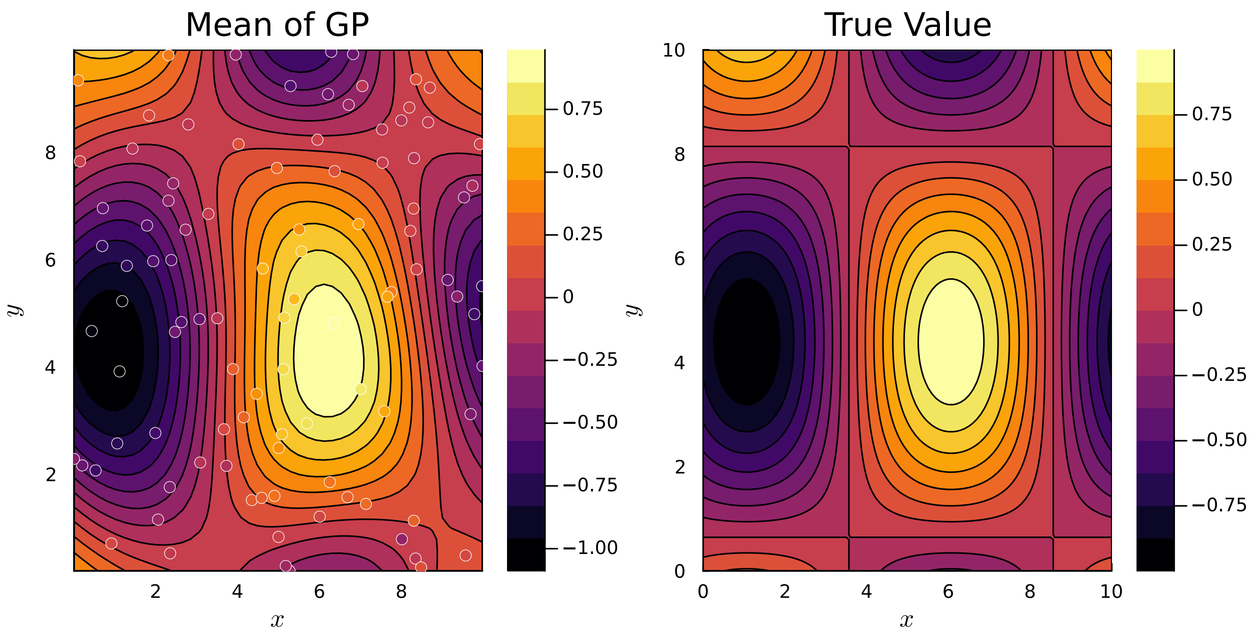
```
100×100 Matrix{Float64}:
 0.0      6.65807  4.35057  2.90869  …  4.34476  8.68652   5.3647   6.5483
 6.65807  0.0      2.41071  4.05802     7.18983  4.72714   7.94644  8.18015
 4.35057  2.41071  0.0      2.23384     5.05482  5.09693   5.93894  6.44113
 2.90869  4.05802  2.23384  0.0         5.58758  7.31103   6.62484  7.47593
 4.41814  3.81749  3.03919  1.70791     7.24932  7.96343   8.27084  9.04786
 4.04066  4.51484  2.4786   3.70141  …  2.675    4.73663   3.48019  3.9823
```

```
5.82664  1.39572  1.49779  3.63975      5.83267  3.96694   6.56158  6.78627
3.0434   5.96467  3.66717  3.98106      1.60763  6.2767    2.65582  3.64631
8.84896  5.40197  5.53284  7.70489      6.25668  0.76966   6.27022  5.58035
4.85759  2.97491  2.5443   1.96409      7.18677  7.25587   8.17051  8.8404
⋮                                    ⋱
8.28739  4.73646  4.85703  7.04243      5.9565   0.518149  6.08127  5.5251
5.37615  6.72105  4.90725  5.98813      1.42043  5.21691   1.40747  1.53738
4.93383  5.6622   3.86469  5.10672      1.89173  4.60516   2.34142  2.58112
3.14183  4.16817  2.57197  0.482333     6.05482  7.66869   7.09585  7.9567
3.82914  3.2297   1.85462  0.932483  …  6.0972   6.90495   7.09856  7.82927
4.34476  7.18983  5.05482  5.58758      0.0      6.46666   1.0608   2.21124
8.68652  4.72714  5.09693  7.31103      6.46666  0.0       6.59941  6.03407
5.3647   7.94644  5.93894  6.62484      1.0608   6.59941   0.0      1.31053
6.5483   8.18015  6.44113  7.47593      2.21124  6.03407   1.31053  0.0
```

6.3 2D GP

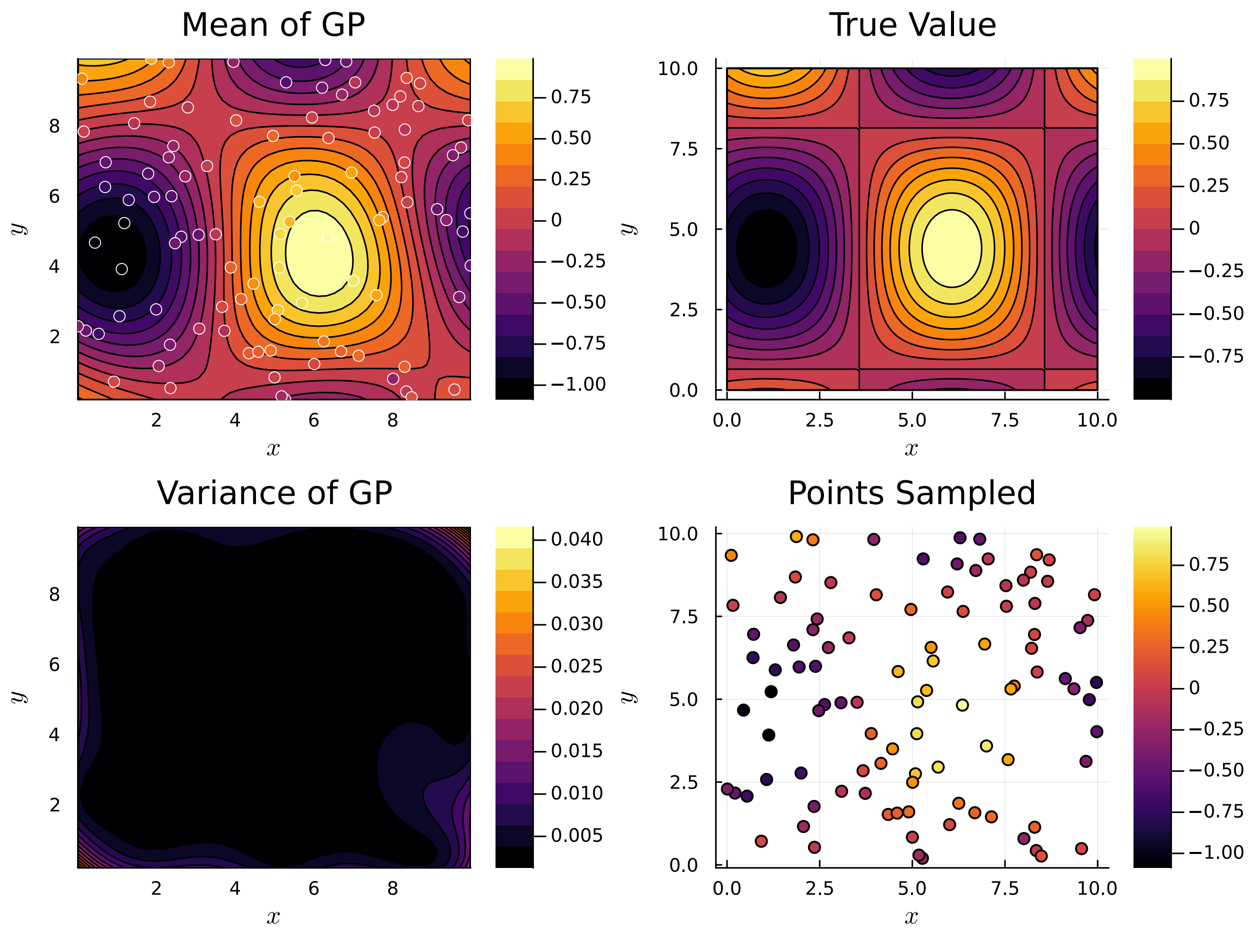
In order to build a GP, we need to specify: 1. A kernel 2. A mean function 3. A noise parameter σ_y

```
let
    σy = 0.25
    ℓ = 2.5
    σ = 2.5

    xy = vcat(x', y')
    kern = GaussianProcesses.SE(log(ℓ), log(σ)) # note: log σ, log ℓ
    μ = GaussianProcesses.MeanZero()
    gp = GP(xy, z, μ, kern, log(σy))
    p1 = deepcopy(baseplot)
    plot!(p1, gp; title="Mean of GP", fill=true)
    scatter!(
        p1, x, y; zcolor=z, label=false, markerstrokecolor=:white,
markerstrokewidth=0.5
    )
    p2 = deepcopy(baseplot)
    contour!(p2, 0:0.1:10, 0:0.1:10, f; title="True Value", fill=true)
    plot(p1, p2; size=(800, 400))
end
```



We can use stable built-in optimization methods. (There are some well-known numerical issues that Gaussian Process models can run into, and avoiding them takes luck or skill or both).

```
gp3 = let
    xy = vcat(x', y')
    kern = GaussianProcesses.SE(0.0, 0.0) # note: log σ, log ℓ
    μ = GaussianProcesses.MeanZero()
    gp = GP(xy, z, μ, kern, log(σy))
    optimize!(gp) # piece of cake!
    gp
end


let
    p1 = deepcopy(baseplot)
    plot!(p1, gp3; title="Mean of GP", fill=true)
    p2 = deepcopy(baseplot)
    contour!(p2, 0:0.1:10, 0:0.1:10, f; title="True Value", fill=true)
    p3 = deepcopy(baseplot)
    scatter!(
        p1, x, y; zcolor=z, label=false, markerstrokecolor=:white,
markerstrokewidth=0.5
    )
    plot!(p3, gp3; title="Variance of GP", fill=true, var=true)
    p4 = deepcopy(baseplot)
    scatter!(p4, x, y; zcolor=z, title="Points Sampled", label=false)
    plot(p1, p2, p3, p4; size=(800, 600), link=:both)
end
```



7 Pros and Cons

1. Pros:

- Uncertainty: predict the entire distribution, confidence interval, and monte carlo possible;
- Complexity: hyperparameters are learned by maximizing the log marginal likelihood, which penalizes over complex models;
- Computation: efficient for small/medium datasets;
- Flexible: approximate extremely complex funtions without specifying a fixed model form

2. Cons:

- Cost: computation cost + memory when n in the kernal matrix $n^{*}n$ is large
- Highly depend on kernal selections
- Not well for sharp jumps, need smooth, continuous functions

8 Reference

Bibliography

[1] J. Doss-Gollin, “1D and 2D Gaussian Process Regression.” Accessed: Dec. 01, 2025. [Online]. Available: https://jdossgollin.github.io/environmental-data-science/Spring22/10_gaussian_process/

[2] C. Sun, N. Wang, S. Jin, and M. Wang, “Study on Spatial Interpolation Method of Marine Sediment Particle Size Based on Geostatistics,” *Journal of coastal research*, vol. 108, no. sp1, pp. 125–130, 2020.

[3] G. Mei, N. Xu, and L. Xu, “Improving GPU-accelerated adaptive IDW interpolation algorithm using fast kNN search,” *SpringerPlus*, vol. 5, no. 1, pp. 1389–1322, 2016.

[4] S. Zhang, “Challenges in KNN Classification,” *IEEE transactions on knowledge and data engineering*, vol. 34, no. 10, pp. 4663–4675, 2022.

[5] digiLab AI, “Kernels and Covariance Matrices.” Accessed: Dec. 01, 2025. [Online]. Available: https://www.youtube.com/watch?v=-FB4DZzyH80

[6] G. A. Achilleos, “Interpolation and elevation errors: the impact of the DEM resolution,” vol. 9535, p. 95350, 2015, doi: 10.1117/12.2192676.