Your Paper

You

October 21, 2023

Abstract

1 Introduction

Our situation is that we have obtained measurements, potentially noisy, of both the u and f. However, we do not have knowledge of the ODE parameters ϕ. This is a typical situation in for example lab work, where we have measurements and want to determine an underlying physical parameter or interpolate the data. This is often done with typical linear regression techniques.
very easy to implement a better working kernel. one would always be able to build a better kernel for a specific problem. However this would require a lot of time and effort. with this framework we have a clear way of implementing a kernel that is based on the underlying physics. We do not need to analytically compute a kernel for a specific problem. We can just take the differential equation and a kernel and apply the differential operator to the kernel. This way we can easily implement a kernel for a specific problem. The next interesting thing is that we also use the knowledge of the forcing term for our prediction and for the calculation of the ODE hyperparameters ϕ. and we can predict the forcing term as well. When comparing this to an inverse problem using a neural network we can see that the implementation is also a lot harder.
We start with a general introduction to Gaussian Processes, Regression with Gaussian Processes and kernels. We then introduce the physics informed framework and the optimization process. In the end we will look at several example problems and discuss the results and the performance of the framework.

2 Fundamentals

2.1 Gaussian Process

As defined in [8] a Gaussian Process is a collection of random variables for which every finite set of variables have a joint multivariate Gaussian distribution. For a Regression model we define a function $f(\boldsymbol{x}) : \mathcal{X} \to \mathbb{R}$ with the input set $\{\boldsymbol{X}\} = \{\boldsymbol{x_n} \in \mathcal{X}\}_{n=1}^{N}$. We now let $f : \mathcal{X} \to \mathbb{R}$ be our Gaussian Process. To fully define the GP we need to specify a mean function $m(\boldsymbol{x})$ and a covariance function or kernel $k(\boldsymbol{x}, \boldsymbol{x'}) : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ for $f(\boldsymbol{x})$:

$$\begin{aligned} m(\boldsymbol{x}) &= \mathbb{E}[f(\boldsymbol{x})] \\ k(\boldsymbol{x}, \boldsymbol{x'}) &= \mathbb{E}[(f(\boldsymbol{x}) - m(\boldsymbol{x}))(f(\boldsymbol{x'}) - m(\boldsymbol{x'}))] \end{aligned} \tag{1}$$

In this work the mean function $m(\boldsymbol{x})$ of the GP is set to zero. This is quite common, mainly for notational reasons [8].
So what we are left with for $m(\boldsymbol{x})$ and $k(\boldsymbol{x}, \boldsymbol{x'})$ is:

$$\begin{aligned} m(\boldsymbol{x}) &= 0 \\ k(\boldsymbol{x}, \boldsymbol{x'}) &= \mathbb{E}[f(\boldsymbol{x})f(\boldsymbol{x'})] \end{aligned} \tag{2}$$

We now write our Gaussian Process $f(\boldsymbol{x})$ as:

$$f(\boldsymbol{x}) \sim \mathcal{GP}(0, k(\boldsymbol{x}, \boldsymbol{x'})) \tag{3}$$

It is important to note that $k(\boldsymbol{x}, \boldsymbol{x'})$ has to be positive definite and symmetric in order to be a valid kernel function. Because of the definition of the GP we can now generate a joint Gaussian distribution for a finite set of points $\boldsymbol{X_*}$:

$$p(\boldsymbol{f_*}|\boldsymbol{X_*}) = \mathcal{N}(\boldsymbol{f_*}|\boldsymbol{0}, \boldsymbol{K_*}) \tag{4}$$

with $[\boldsymbol{K}]_{i,j} = k(\boldsymbol{x_{i*}}, \boldsymbol{x_{j*}})$.

2.2 Regression

The goal of regression is given a training dataset $\mathcal{D} = \{(\boldsymbol{x_i}, y_i)|i = 1, \ldots, N\}$ we want to make predictions for new test points $\boldsymbol{x_*}$ which are not in the training set. There are several methods which can be used to tackle this kind of problem. For example the basic linear regression using least squares and specific basis functions or a neural network that does regression. In this work we make use of the Gaussian process which is a non-parametric model. We will start with a Gaussian Process prior and then use the rules for Gaussian multivariate distributions to define a posterior predictive distribution, which we can then use to make predictions for new test points.

The general setup for regression is that we make observations/measurements y of an underlying function $f(\boldsymbol{x})$ at points $\boldsymbol{x}$ which are corrupted by a zero mean Gaussian noise with variance σ_n^2:

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{X}) + \mathcal{N}(0|\sigma_n^2 \boldsymbol{I}) \tag{5}$$

When we set the GP Prior onto the function f we have to add the normally distributed noise. To get the prior for the observations $\boldsymbol{y}$ we simply need to change the covariance function to $\text{cov}(\boldsymbol{y_p}, \boldsymbol{y_q}) = k(\boldsymbol{x_p}, \boldsymbol{x_q}) + \sigma_n^2 \delta_{pq}$.
As described before in equation 4 we can generate a Gaussian random vector for new test data $\boldsymbol{X_*}$. Both $\boldsymbol{y}$ and $\boldsymbol{f_*}$ are now multivariate Gaussians. We can therefore introduce the joint distribution of the observed data and the values of f at the test points $\boldsymbol{X_*}$ as:

$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{f_*} \end{bmatrix} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix} K(\boldsymbol{X}, \boldsymbol{X}) + \sigma_n^2 \boldsymbol{I} & K(\boldsymbol{X}, \boldsymbol{X_*}) \\ K(\boldsymbol{X_*}, \boldsymbol{X}) & K(\boldsymbol{X_*}, \boldsymbol{X_*}) \end{bmatrix}\right) \tag{6}$$

From this joint Gaussian prior we can then calculate the predictive posterior distribution by applying the rules for conditioning Gaussians. A good description for this conditioning can be found in [1] (chapter 2.3.1). The predictive distribution is then given by:

$$\begin{aligned} p(\boldsymbol{f_*}|\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{X_*}) &= \mathcal{N}(\boldsymbol{f_*}|\boldsymbol{\mu_*}, \boldsymbol{\Sigma_*}) \\ \boldsymbol{\mu_*} &= K(\boldsymbol{X_*}, \boldsymbol{X})[K(\boldsymbol{X}, \boldsymbol{X}) + \sigma_n^2 \boldsymbol{I}]^{-1} \boldsymbol{y} \\ \boldsymbol{\Sigma_*} &= K(\boldsymbol{X_*}, \boldsymbol{X_*}) - K(\boldsymbol{X_*}, \boldsymbol{X})[K(\boldsymbol{X}, \boldsymbol{X}) + \sigma_n^2 \boldsymbol{I}]^{-1} K(\boldsymbol{X}, \boldsymbol{X_*}) \end{aligned} \tag{7}$$

We now have obtained the posterior predictive mean $\boldsymbol{\mu_*}$ and the posterior covariance σ for the new test inputs $\boldsymbol{X_*}$.

2.3 Kernels

In [8] a kernel is defined as a function $k(\boldsymbol{x}, \boldsymbol{x'})$, which maps two input pairs $\boldsymbol{x} \in \mathcal{X}$ and $\boldsymbol{x'} \in \mathcal{X}'$ into $\mathbb{R}$. To be a valid covariance function the kernel needs to satisfy two main conditions. First it needs to be symmetric so that $k(\boldsymbol{x}, \boldsymbol{x'}) = k(\boldsymbol{x}, \boldsymbol{x'})$ and second it needs to be positive semidefinite (PSD) The general covariance function used in this work is the squared exponential covariance function (SE) as defined in [8]:

$$k(\boldsymbol{x_p}, \boldsymbol{x_q}) = \sigma_f^2 \exp(-0.5(\boldsymbol{x_p} - \boldsymbol{x_q})^\intercal M(\boldsymbol{x_p} - \boldsymbol{x_q})) \tag{8}$$

with the variance hyperparameter σ_f^2 and the matrix $M = \text{diag}(\boldsymbol{l})^{-2}$ containing the vector with the characteristic length scales l_i. The hyperparameters are collected in the vector $\boldsymbol{\theta} = (\boldsymbol{l}, \sigma_f^2)$.
For the one dimensional case the SE covariance function simply becomes:

$$k(x_p, x_q) = \sigma_f^2 \exp\left(-\frac{0.5}{l^2}(x_p - x_q)^2\right) \tag{9}$$

For a two-dimensional case with time being the second dimension, two independent length scale parameters are used for the separate space and time domains $\boldsymbol{l} = (l_x, l_t)$. For this case, we can re-write equation 7 the covariance function with $\boldsymbol{x_p} = (x_p, t_p)^\intercal$ and $\boldsymbol{x_q} = (x_q, t_q)^\intercal$ as the product of two single SE covariance functions:

$$k(\boldsymbol{x_p}, \boldsymbol{x_q}) = \sigma_f^2 \exp\left(-\frac{0.5}{l_x^2}(x_p - x_q)^2\right) \exp\left(-\frac{0.5}{l_t^2}(t_p - t_q)^2\right) \tag{10}$$

There are many kernels one can choose from. A good overview over some kernel functions, as well as how they can be modified, can be found in [3] as well as in [6]. Later we will modify the SE kernels with linear operators, more specifically with differential operators. For this the SE kernel is a good choice, because it is infinitely differentiable with rather simple and smooth derivatives. This is important for the overall numerical stabililty when calculating the gradients of the modified kernel in the optimization process.

2.4 Marginal likelihood

For the optimization of the hyperparameters θ and ϕ the negative marginal log likelihood (*nmll*) [8] is minimized. The *nmll*

$$\log p(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{\theta}) = -\frac{1}{2}\boldsymbol{y}^{\intercal}\boldsymbol{K}_{\boldsymbol{y}}^{-1}\boldsymbol{y} - \frac{1}{2}\log|\boldsymbol{K}_{\boldsymbol{y}}| - \frac{n}{2}\log(2\pi) \tag{11}$$

with $K_y = K_f + \sigma_n^2\boldsymbol{I}$. As described in [8], the *mll* consists of three parts, which can each be understood separately. The first term $\frac{1}{2}\boldsymbol{y}^{\intercal}\boldsymbol{K}_{\boldsymbol{y}}^{-1}\boldsymbol{y}$ is in charge of the data fit, the second term $\log|K_y|$ is the complexity penalty and the last term is a normalization term.

2.5 Linear operators and GPs

A useful property of Gaussian Processes is that their linear transformation is still a Gaussian Process. This main

3 Physics informed framework

We choose our problem setup similar/eual to the one proposed in [7]. At the top we have some sort of linear differential equation of a physical model:

$$\mathcal{L}_{\boldsymbol{x}}^{\phi}u(\boldsymbol{x}) = f(\boldsymbol{x}) \tag{12}$$

which can be written as a linear operator $\mathcal{L}_{\boldsymbol{x}}^{\phi}$ acting on the function $u(\boldsymbol{x})$. $\mathcal{L}_{\boldsymbol{x}}^{\phi}$ is a differential operator that resembles the differential equation with ϕ being a list containing the a-priori unknown parameters which also define the differential equation. The function $f(\boldsymbol{x})$ acts as the forcing term and $u(\boldsymbol{x})$ is the solution of the differential equation.

We now start by assuming that the solution of the differential equation $u(\boldsymbol{x})$ is a Gaussian Process of the form:

$$u(\boldsymbol{x}) \sim \mathcal{GP}(\boldsymbol{0}, k_{uu}(\boldsymbol{x},\boldsymbol{x'};\boldsymbol{\theta})) \tag{13}$$

with $k_{uu}(\boldsymbol{x},\boldsymbol{x'};\boldsymbol{\theta})$ being the kernel depending on its hyperparameters θ for the $\mathcal{GP}$ (on) $u(\boldsymbol{x})$. With the known linear relation between u and f from equation 12, we can now say that $f(\boldsymbol{x})$ must also be $\mathcal{GP}$ [8] of form:

$$f(\boldsymbol{x}) \sim \mathcal{GP}(\boldsymbol{0}, k_{ff}(\boldsymbol{x},\boldsymbol{x'};\boldsymbol{\theta})) \tag{14}$$

with now k_{ff} being the kernel defining the $\mathcal{GP}$ (on) $f(\boldsymbol{x})$. As shown in [4], [10] and [7] k_{ff} is defined by:

$$k_{ff}(\boldsymbol{x},\boldsymbol{x'};\boldsymbol{\theta},\boldsymbol{\phi}) = \mathcal{L}_{\boldsymbol{x}}^{\phi}\mathcal{L}_{\boldsymbol{x'}}^{\phi}k_{uu}(\boldsymbol{x},\boldsymbol{x'};\boldsymbol{\theta}) \tag{15}$$

We also need the mixed terms between $u(x)$ and $f(x)$, namely k_{uf} and k_{fu} which are formed by:

$$\begin{aligned} k_{uf}(\boldsymbol{x},\boldsymbol{x'};\boldsymbol{\theta},\boldsymbol{\phi}) &= \mathcal{L}_{\boldsymbol{x'}}^{\phi}k_{uu}(\boldsymbol{x},\boldsymbol{x'};\boldsymbol{\theta}) \\ k_{fu}(\boldsymbol{x},\boldsymbol{x'};\boldsymbol{\theta},\boldsymbol{\phi}) &= \mathcal{L}_{\boldsymbol{x}}^{\phi}k_{uu}(\boldsymbol{x},\boldsymbol{x'};\boldsymbol{\theta}) \end{aligned} \tag{16}$$

To simplify the notation, the dependence on the hyperparameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ is not written explicitly anymore. With this construction we now have two Gaussian Processes. As described in [4] we can now create a joint Gaussian process of u and f in the form: (not sure if I understood this correctly)

$$\begin{bmatrix} u \\ f \end{bmatrix} \sim \mathcal{GP}\left(\boldsymbol{0}, \begin{bmatrix} k_{uu} & k_{uf} \\ k_{fu} & k_{ff} \end{bmatrix}\right) \tag{17}$$

with $k_{uu} : \mathcal{U} \times \mathcal{U}$, $k_{uf} : \mathcal{U} \times \mathcal{F}$, $k_{fu} : \mathcal{F} \times \mathcal{U}$ and $k_{ff} : \mathcal{F} \times \mathcal{F}$ We now look at noisy observations of both u and f,with noise σ_u and σ_f, of the form:

$$\begin{aligned} y_{i,u} &= u_i(\boldsymbol{x_{i,u}}) + \mathcal{N}(0|\sigma_u^2) \\ y_{i,f} &= u_i(\boldsymbol{x_{i,f}}) + \mathcal{N}(0|\sigma_f^2) \end{aligned} \tag{18}$$

with which we create two data sets $\{\boldsymbol{X_u}, \boldsymbol{y_u}\}$ and $\{\boldsymbol{X_f}, \boldsymbol{y_f}\}$. Similarly, as we did before for the test points in equation 6 we can now write (create) a joint Gaussian distribution of the observations $\boldsymbol{y_u}$ and $\boldsymbol{y_f}$. We collect $\boldsymbol{X_u}, \boldsymbol{X_f}$ in $\boldsymbol{X}$ and $\boldsymbol{y_u}, \boldsymbol{y_f}$ in $\boldsymbol{y}$. With the joint GP at equation 17, the joint distribution will take the following form:

$$p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}, \sigma_{n_u}^2, \sigma_{f_u}^2, \boldsymbol{\phi}) = \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix} k_{uu}(\boldsymbol{X_u}, \boldsymbol{X_u}) + \sigma_{n_u}^2 \boldsymbol{I} & k_{uf}(\boldsymbol{X_u}, \boldsymbol{X_f}) \\ k_{fu}(\boldsymbol{X_f}, \boldsymbol{X_u}) & k_{ff}(\boldsymbol{X_f}, \boldsymbol{X_f}) + \sigma_{n_f}^2 \boldsymbol{I} \end{bmatrix}\right) \tag{19}$$

We can directly use $\boldsymbol{K}$ to compute the mag log likelihood and train the model to obtain the optimized hyperparameters $\boldsymbol{\theta_*}$ and $\boldsymbol{\phi_*}$. More information about the optimization process will be provided in the next chapter.

3.1 Optimization/Finding of the hyperparameters

Arguably the most important part of the overall framework is the optimization of the hyperparameters. The optimization is done by minimizing the negative marginal log likelihood (mll) of the joint GP with respect to the hyperparameters $\boldsymbol{\theta}, \boldsymbol{\phi}$. The mll is defined as:

$$\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}, \sigma_{n_u}^2, \sigma_{f_u}^2, \boldsymbol{\phi}) = -\frac{1}{2}\boldsymbol{y}^\intercal \boldsymbol{K}^{-1}\boldsymbol{y} - \frac{1}{2}\log|\boldsymbol{K}| - \frac{n}{2}\log(2\pi) \tag{20}$$

with $\boldsymbol{K}$ being the covariance matrix of the joint GP in equation 19. For the optimization we tried different gradient based optimizers, aswell as a heuristic search method, namely the Nelder-Mead method. All the optimizers used came from the scipy.optimize.minimize function from the scipy library. We tried several gradient based optimizers like the limited memmory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS-B) algorithm, the conjugate gradient (CG) algorithm and the truncated Newton (TNC) algorithm. The best results were however obtained by using the Nelder-Mead [9]. From the gradient based methods the TNC performed the best and was most of the time nearly as good as the Nelder-Mead. However, the TNC was significantly slower and needed more restarts than the Nelder-Mead. Especially with an increasing number of training points the gradient based optimizers failed completly and the Nelder-Mead method was the only one that still worked. This could be because the Nelder Mead algorithm does not use any gradient information and that for more training points the calculation of the gradients of the mll gets unstable. Generally, it is important to add some jitter to the diagonal of $\boldsymbol{K}$ to minimize the risk of numerical instabilities when calcualting the Cholesky decomposition of $\boldsymbol{K}$. Also, without jitter a significant part of the optimization landscape is not defined, because of the gram matrix $\boldsymbol{K}$ not beeing positive definite for some hyperparameter values. The best results were obtained when using a jitter in the range of $10^{-6} to 10^{-7}$

A common problem in the optimization process of non-convex funtion is the problem of converging to a local minimum. To avoid this problem we started the optimization process for different initial values of the hperparameters. The initial values were sampled from uniform distributions, only the length scale hyperparameter l was sampled from a log uniform distribution because of it beeing a scale parameter. Because of the random initialization, and the presumably high number of local minima in the optimization landscape, the results for the hyperparameters varied. However, for a sufficient number of optimization runs the results normally reached a similar accuracy and similar hyperparamter values.

To conclude, the optimization process is the key part of the framework, because we can directly make predictions with the optimized hyperparameters, but also obtain the parameters ϕ of the underlying differential equation. A key step for improving the overall framework would be to improve the optimization process. However, this would be very time-consuming and the obtained results are sufficently accurate, aswell as the computation time beeing reasonably short, for the scope of this work.

3.2 Predictions with the Posterior

With the obtained hyperparameters $\boldsymbol{\theta}_*$ and $\boldsymbol{\phi}_*$ we can make new predictions at a test location $\boldsymbol{x}_*$. Similar to the construction in chapter 2.2 we can now formulate the predictive posterior distribution for $u(\boldsymbol{x}_*)$ and also for $f(\boldsymbol{x}_*)$ as done in [7]:

$$\begin{aligned} p(u(\boldsymbol{x}_*)|\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{\theta}_*, \boldsymbol{\phi}_*) &= \mathcal{N}(\mu_{u*}, \Sigma_u) \\ p(u(\boldsymbol{x}_*)|\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{\theta}_*, \boldsymbol{\phi}_*) &= \mathcal{N}(\mu_{f*}, \Sigma_f) \end{aligned} \tag{21}$$

with

$$\begin{aligned} \mu_u &= \boldsymbol{\kappa_u} \boldsymbol{K}^{-1} \boldsymbol{y},\ \Sigma_* = k_{uu}(\boldsymbol{x_*}, \boldsymbol{x_*}) - \boldsymbol{\kappa_u} \boldsymbol{K}^{-1} \boldsymbol{\kappa_u}^\intercal \\ \mu_f &= \boldsymbol{\kappa_f} \boldsymbol{K}^{-1} \boldsymbol{y},\ \Sigma_* = k_{ff}(\boldsymbol{x_*}, \boldsymbol{x_*}) - \boldsymbol{\kappa_f} \boldsymbol{K}^{-1} \boldsymbol{\kappa_f}^\intercal \end{aligned} \tag{22}$$

with $\boldsymbol{\kappa_u}$ and $\boldsymbol{\kappa_f}$ being:

$$\begin{aligned} \boldsymbol{\kappa_u} &= [k_{uu}(\boldsymbol{x_*}, \boldsymbol{X_u}) k_{uf}(\boldsymbol{x_*}, \boldsymbol{X_f})] \\ \boldsymbol{\kappa_f} &= [k_{fu}(\boldsymbol{x_*}, \boldsymbol{X_u}) k_{ff}(\boldsymbol{x_*}, \boldsymbol{X_f})] \end{aligned} \tag{23}$$

4 Numerical/Computational implementation

In this section we will present the computational implementation in python of the kernel, the log marginal likelihood and of the predictive distribution. For greatly improving the computation time as well as the usage of automatic differentiation the jax library [2] was used. A general introduction can be found here, as well as a specific introduction to kernel derivatives with jax here. Goal of this section is to give a short overview of the computational methods used for the implementation of the Gaussian process regression and the construction of the kernel.

4.1 Kernel

For constructing and deriving the kernel as described in equations 15 and 16 we used two different computational approaches. The first approach was to analytically calculate the derivatives arising from the differential operators and the second one was to use automatic differentiation. For a general overview the process is shown with the specific example ODE of a damped oscillator with the differential operator $\mathcal{L}_t^\phi = m\frac{\partial^2 u(t)}{\partial t^2} + \gamma \frac{\partial u(t)}{\partial t} + ku(t)$.

$$\begin{aligned} k_{ff}(\boldsymbol{x}, \boldsymbol{x'}; \boldsymbol{\theta}, \boldsymbol{\phi}) &= \mathcal{L}_{\boldsymbol{x}}^\phi \mathcal{L}_{\boldsymbol{x'}}^\phi k_{uu}(\boldsymbol{x}, \boldsymbol{x'}; \boldsymbol{\theta}) \\ &= \left[m\frac{\partial^2}{\partial t^2} + \gamma \frac{\partial}{\partial t} + k \right] \left[m\frac{\partial^2}{\partial t'^2} + \gamma \frac{\partial}{\partial t'} + k \right] k_{uu} \\ &= \left[m^2 \frac{\partial^4}{\partial t^2 \partial t'^2} + m\gamma \frac{\partial^3}{\partial t^2 \partial t'} + m\gamma \frac{\partial^3}{\partial t'^2 \partial t} + \gamma^2 \frac{\partial^2}{\partial t \partial t'} + k\gamma \frac{\partial}{\partial t'} + k\gamma \frac{\partial}{\partial t} + mk \frac{\partial^2}{\partial t'^2} + mk \frac{\partial^2}{\partial t^2} + k^2 \right] k_{uu} \\ &= \left[m^2 \frac{\partial^4}{\partial t^2 \partial t'^2} + \gamma^2 \frac{\partial^2}{\partial t \partial t'} + 2mk \frac{\partial^2}{\partial t'^2} + k^2 \right] k_{uu} \end{aligned} \tag{24}$$

with $k_{uu} = \sigma_f^2 \exp\left(-\frac{0.5}{l^2}(t - t')^2 \right)$ beeing the general SE kernel introduced in section 2.3. Note that this can be done with any other kernel as well. The analytical computation is shown in 1.

Listing 1: Analytical computation of the derivatives

```
    b = params[3]
    k = params[4]
    gamma = 0.5 / params[0]**2
    #dt^2 dt'^2
    dif = (x-y)
    k_xxyy = (16*gamma**4* dif**4 - 48*gamma**3*dif**2 + 12*gamma**2)
    #dt'^2
    k_yy = 2*gamma*(2*gamma * (x-y)**2 - 1)
    #dtdt'
    k_xy = (2*gamma - 4*gamma**2*dif**2)
    #no div
    k_normal = rbf_kernel_single(x, y, params)
    return jnp.squeeze(m**2 * k_xxyy + 2*m*k*k_yy + b**2 * k_xy + k**2) * rbf_kernel_single(x

k_ff = jit(vmap(vmap(k_ff, (None, 0, None)), (0, None, None)))
```

For the SE kernel the computations are rather simple and can easily be done analytically. However, if one chooses a complex kernel, the analytical computation can become rather complicated and time-consuming. Therefore, we also implemented the kernel derivatives with help of autograd. Namely, we used the jax library for the automatic differentiation. The code for the autograd computation of the derivatives is shown in 2. The big advantage of this approach is that it is sufficient to do the calculations as done in 24 and implement it as shown in the code. With this the choice of the general kernel k_{uu} is only restricted within the jax library. This means that the kernel can be any function that is traceable by the jax autograd and jit functions.

Listing 2: Computation of the derivatives using autograd

```
def k_ff_jax(x,y, params):
    x, y = jnp.squeeze(x), jnp.squeeze(y)
    m = params[2]
    g = params[3]
    k = params[4]
    #dt^2 dt'^2
    dk_yy = grad(grad(rbf_kernel_single, argnums=1), argnums=1)
    dk_xxyy = grad(grad(dk_yy, argnums=0), argnums=(0)) (x, y, params)
    #dt'^2
    dk_yy = grad(grad(rbf_kernel_single, argnums=1), argnums=1) (x, y, params)
    #dtdt'
    dk_xy = grad(grad(rbf_kernel_single, argnums=1), argnums=(0)) (x, y, params)
    #k^2
    k_normal = rbf_kernel_single(x, y, params)
    return m**2 * dk_xxyy + 2*m*k*dk_yy + g**2 * dk_xy + k**2 * k_normal

k_ff_jax = jit(vmap(vmap(k_ff_jax, (None, 0, None)), (0, None, None)))
```

When comparing the accuracy and the speed of the two approaches there was no notable difference. In particular, the two functions were nearly identical in terms of computation time. For the majority of the examples we used the analytical approach, because it is more transparent and easier to verify.

4.2 Log marginal likelihood and predictive distribution

A general problem of Gaussian processes is the numerical calculation of the inverse of the covariance matrix $\boldsymbol{K}$. This problem becomes increasingly apparent with a high number of training points. Especially in the training process the inverse of $\boldsymbol{K}$ has to be calculated several times. However, with the vectorized implementation, as well as the use of just in time (jit) compiled functions from the jax library the computation time was not a major problem. Additionally, the number of training points was held at a rather low number (highest number was 400 for the 3d wave equation). Still, the calculation of the inverse can be a source of numerical instabilities. To minimize numerical errors and to avoid a direct inversion of $\boldsymbol{K}$ the calculations involving the inverse were rewritten using the cholesky decomposition.
The cholesky decomposition of a positive definite, symmetric matrix $\boldsymbol{A}$ is given by:

$$\boldsymbol{K} = \boldsymbol{L}\boldsymbol{L}^\intercal \tag{25}$$

with $\boldsymbol{L}$ being a lower triangular matrix [8]. With this the general term involving the inversion of $\boldsymbol{K}$ can be written as:

$$\boldsymbol{u}^\intercal \boldsymbol{K}^{-1} \boldsymbol{v} = \boldsymbol{u}^\intercal (\boldsymbol{L}\boldsymbol{L}^\intercal)^{-1} \boldsymbol{v} = \boldsymbol{u}^\intercal \boldsymbol{L}^\intercal \backslash (\boldsymbol{L} \backslash \boldsymbol{v}) \tag{26}$$

shown with two general vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ and $\backslash$ as the left matrix divide operator. For this we used that we can rewrite $(\boldsymbol{L}\boldsymbol{L}^\intercal)^{-1}\boldsymbol{v}$ as a solution $\boldsymbol{x}$ of a linear system of the form $\boldsymbol{L}\boldsymbol{L}^\intercal \boldsymbol{x} = \boldsymbol{v}$, which we now solve for $\boldsymbol{x}$. This is a very stable and efficient way of calculating the equation. The implementation of the log marginal likelihood is shown in 3.

Listing 3: Log marginal likelihood

```python
    def log_marginal_likelohood(self, params):
        """computes the log marginal likelihood of the GP"""
        K = self.gram_matrix(self.X, self.Y, params, self.noise)
        L = jnp.linalg.cholesky(K + self.jitter * jnp.eye(len(K)))

        alpha = jnp.linalg.solve(L.T, jnp.linalg.solve(L, self.targets))
        mll = 1/2 * jnp.dot(self.targets.T, alpha) + 0.5*jnp.sum(
            jnp.log(jnp.diagonal(L))) + len(self.X)/2 * jnp.log(2*jnp.pi)
        return jnp.squeeze(mll)
```

For the predictive distribution we again rewrote equation 22 as described in equation 26. The implementation of the predictive distribution is shown in 4.

Listing 4: Predictive distribution

```python
        best_result = min(results, key=lambda x: x.fun)
        print(best_result)

        return best_result

    def predict_model(self, X_star):
        self.mean_u, self.var_u = self.predict_u(X_star)
        self.mean_f, self.var_f = self.predict_f(X_star)

    def predict_u(self, X_star):
        """calculates the mean and variance of the GP at the points X_star"""
        params = self.get_params()

        K = self.gram_matrix(self.X, self.Y, params, self.noise)
        L = jnp.linalg.cholesky(K+ jnp.eye(len(K)) * self.jitter)

        q_1 = self.k_uu(X_star, self.X, params)
```

5 Results

In the following we will call the physics informed model, which is used here, the informed model and the vanilla model the model without the physics informed framework and only a general SE kernel.

For testing the framework we will use ODEs and PDEs of physical models. More specifically, we will look at the three main types of PDEs, namely the Elliptic, Hyperbolic, and Parabolic type and as a starting example a damped oscillator in one dimension. For obtaining $u(\boldsymbol{x})$, we will fully define the differential equation with its initial- and boundary conditions and then use mathematicas NDSolve (numerical solution) or NSolve (analytical solution) to compute the solution $\{(\boldsymbol{x}); u(\boldsymbol{x}), f(\boldsymbol{x})\}$. The obtained solutions will be used as the ground truth. The training and validation points will be sampled from the solution domain $[\boldsymbol{x}_{\mathrm{min}}, \boldsymbol{x}_{\mathrm{max}}]$ using a sobol sequence[cite]. We will add some noise to the samples for the training and validation points, to analyze the impact onto the results and to simulate real world data. Furthermore, often the incorporation of a small noise improves stability in the optimization and prediction process. This will be done by creating two data sets $\{(\boldsymbol{x_u}); \boldsymbol{y_u}\}$ and $\{(\boldsymbol{x_f}); \boldsymbol{y_f}\}$ with $\boldsymbol{y_i} = u(\boldsymbol{x_i}) + \mathcal{N}(\mathbf{0}|\sigma_{ni}^2 I);\ i = u, f$. The implementation of a sobol sequence helps with evenly spaced training and validation points, which is important for the training process of the model. Through the optimization of the mll, we will obtain the parameters of the underlying equation ϕ, as well as the hyperparameters θ of the kernel. With these parameters we will then calculate the predictive mean and covariance and then compare the ground truth with the computed mean function. To quantify/benchmark the performance we will look at the marginal log likelihood, the mean squared error (MSE) and the relative L2 error of the validation set(ground truth). As another way of looking at the performance of the informed model we will compare it to a vanilla model with a general SE kernel, as described in 2.2. For this we will use the GPy library [5]. However, it is important to acknowledge that the vanilla model with the simple SE kernel only has two/three (for cases with time as a second dimension) hyperparameters, namely the length scale l and the variance σ_f^2. Additionally, the informed model takes advantage of the joint GP. This means that the informed model has more parameters to optimize and additional information from the joint GP, and therefore it is not a fair comparison. However, it is still interesting to see how the informed model performs compared to the vanilla model and to have a general benchmark. To further investigate this we will give one example with the same amount of hyperparameters, namely the Poisson equation and compare the results of the two models.

5.1 Damped oscillator

As a first example we will look at a one dimensional differential equation of a damped oscillator with mass m, damping γ and force constant k. The differential equation is given by:

$$\begin{aligned} m\frac{\partial^2 u(t)}{\partial t^2} + \gamma\frac{\partial u(t)}{\partial t} + ku(t) = f(t) \\ u(0) = 0,\ \frac{\partial u(t)}{\partial t} = 0 \end{aligned} \tag{27}$$

The first data set was generated with $m = 1, \gamma = 2, k = 2$ and a simple sinusoidal forcing term $f(t) = 4\sin(2\pi t)$. For the added noise we used $\sigma_{nu} = 10^{-4}$ and $\sigma_{nf} = 10^{-4}$ which can be seen as noise free data. We calculated the result in the time interval $t \in [0, 3]$ from with we sample the training points. The model learns the hyperparameters $\boldsymbol{\theta}$ $(l, \sigma_f, m, \gamma, k)$ to be $(0.4890, 0.5860, 0.9989, 1.9925, 1.9987)$. These results are very accurate considering we are only using 9 training points for a model with 5 hyperparameters. As elaborated before, the exact value of the final values after optimizing varies in a small range. The predictive distribution for the general solution $u(t)$ and the forcing term $f(t)$ can be seen in figure 1. Looking at the results for the informed model we can see that the prediction for $u(t)$ and for $f(t)$ are very precise. The posterior mean overlines with the validation points and the uncertanty band is so small that is not even visible. The mean squared error of $u(t)$ and $f(t)$ are $MSE_u = 1.419 \cdot 10^{-7}$ and $MSE_f = 4.609 \cdot 10^{-3}$. The predictive mean of the vanilla model can also be seen in figure 1. The MSEs for the vanilla model are $MSE_{u,va} = 5.469 \cdot 10^{-4}$ and $MSE_{f,va} = 2.692$. The MSE for the vanilla model is 3 to 4 orders of magnitude higher than the MSE of the informed model which gets very clear when looking at figure 1. The predictions from the vanilla model are not very precise with large uncertanties. This is also reflected in the MSEs. The informed model even manages to accuratly predict in areas away from any training points, as well as at the edges of the domain.

To investiagete the impact of noise we will look at the same example as before, but with added noise. We will use $\sigma_{nu} =$ and σ_{nf}. This is a significant amount of noise, but the model still manages to predict the hyperparameters as $(1, 1, 1)$. To benchmark the performance further we looked at cases with different number of training points. The results can be seen in table 1.

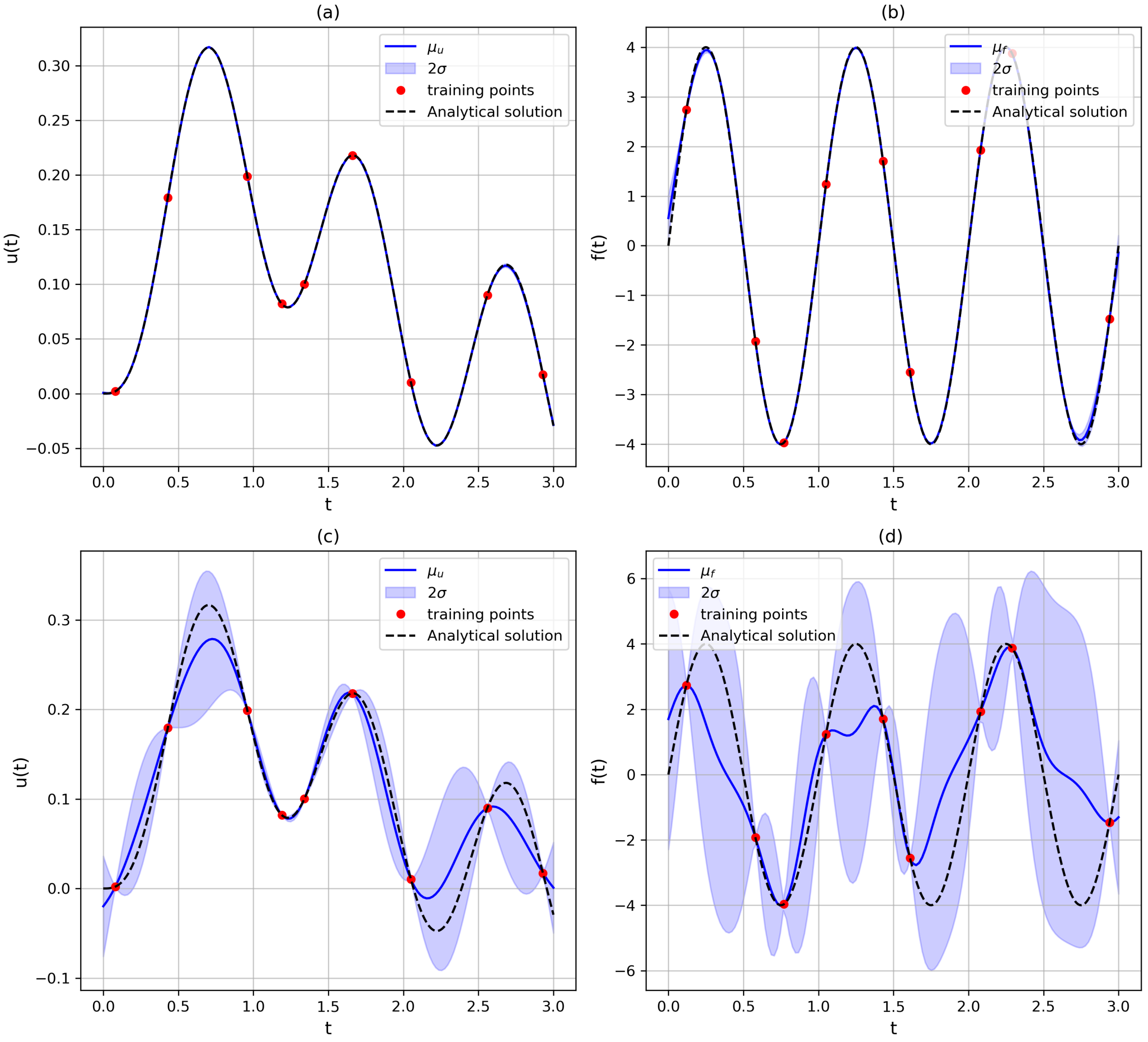

Figure 1: (a) & (b) predictive mean μ_u and μ_f for the informed model, two σ uncertanty band, training and validation points sampled from $\{\boldsymbol{x_u}, u(\boldsymbol{y_u})\}$ and $\{\boldsymbol{x_f}, u(\boldsymbol{y_f})\}$ respectivly. (c) & (d) predictive mean μ_u and μ_f for the vanilla model, two σ uncertanty band, training and validation points sampled from $\{\boldsymbol{x_u}, u(\boldsymbol{y_u})\}$ and $\{\boldsymbol{x_f}, u(\boldsymbol{y_f})\}$ respectivly. The noise σ_{nu} and σ_{nf} were both set to 10^{-8}. The MSEs of the informed model are $MSE_u = 1.419 \cdot 10^{-7}$ and $MSE_f = 4.609 \cdot 10^{-3}$. The MSEs for the vanilla model are $MSE_{u,va} = 5.469 \cdot 10^{-4}$ and $MSE_{f,va} = 2.692$.

Table 1: MSEs for the informed model with different number of training points n. The used noise was $\sigma_{nu} =$ and σ_{nf}.

n	5	10	20	50
mll	3.722	-10.049	-53.227	-32.808
L_u^2				
L_f^2				
MSE_u	$5.583 \cdot 10^{-3}$	$1.070 \cdot 10^{-7}$	$2.463 \cdot 10^{-8}$	$4.022 \cdot 10^{-8}$
MSE_f	4.974	$2.347 \cdot 10^{-3}$	$1.076 \cdot 10^{-3}$	0.105

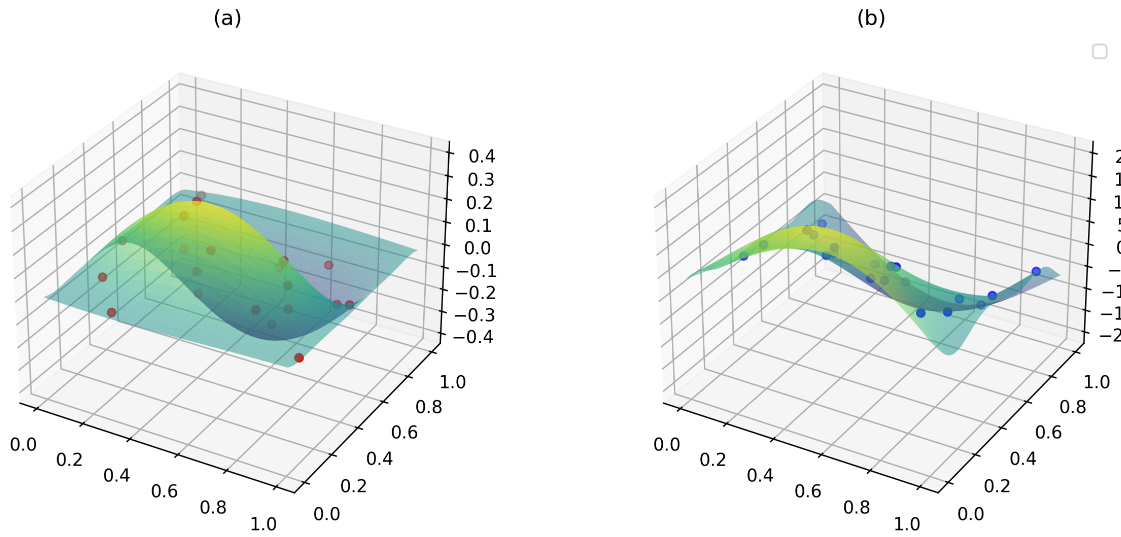

Figure 2: Plots for the informed model. (a) & (b) predictive mean $\boldsymbol{\mu_u}$ and $\boldsymbol{\mu_f}$ with training points in red. (c) & (d) predictive standard deviations $\boldsymbol{\sigma_u}$ and $\boldsymbol{\sigma_f}$. (e) & (f) difference between the predictive mean $\boldsymbol{\mu}$ and the ground truths u and f. The MSEs for the informed model are $MSE_u = \cdot 10^{-7}$ and $MSE_f = \cdot 10^{-3}$. The relative L^2 errors are $L^2_u =$ and $L^2_f =$.

5.2 Heat Equation

The one dimensional time dependent heat equation with a forcing term is:

$$\begin{aligned} \frac{\partial u(x,t)}{\partial t} - \alpha \frac{\partial^2 u(x,t)}{\partial x^2} &= f(x,t) \\ u[0,x] &= \\ u(t,0) =; u(t,L) &= \end{aligned} \tag{28}$$

with α as the thermal diffusivity.

5.3 Wave Equation

For the second example we will look at the two-dimensional time dependent wave equation. The wave equation is a hyperbolic partial differential equation that describes the propagation of waves of various types. In our case we can think of the wave equation as a string that is fixed at both ends. The wave equation is given by:

$$\begin{aligned} \frac{1}{c^2}\frac{\partial^2 u(x,y,t)}{\partial t^2} - (\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2})u(x,y,t) &= f(x,y,t) \\ u[0,x] &= \\ u(t,0) =; u(t,L) &= \end{aligned} \tag{29}$$

with c as the wave speed. The ground truth was calculated with $c = 3$ and with $f(x,t) =$. We used 20 training points for both u and f. The training data was sampled from the domain $x \in [0,1]$ and $t \in [0,1]$. The noise was set to $\sigma_{nu} = 10^{-4}$ and $\sigma_{nf} = 10^{-4}$, which can be considered as nearly noise free. The hyperparameters (l_x, l_t,σ_f, c) were optimized to be $(1,1,1)$.

5.4 Poisson Equation

In our last example we will look at a case where we do not incorporate additional ODE parameters ϕ in the form of hyperparameters into the kernel. This is to better look at the general predictive performance of the informed model in comparison to the vanilla model. The informed, aswell as the vanilla model will both not incorporate ARD, therefore there will only be one singel length scale l for both spatial dimensions. With this example we now have the same amount of optimizable hyperparameters.

We will look at the Poisson equation in two spatial dimensions with a forcing term and with dirichlet boundary conditions:

$$\begin{aligned} \nabla^2 u(x,y) &= f(x,y) \\ u[0,x] &= \\ u(t,0) = 0; u(t,L) &= 0 \end{aligned} \tag{30}$$

Overall, the performance of the infomed model is significantly better. The relative error for L_u^2 is and L_f^2 is. The predictive mean, the predictive variance and the calculated difference between the prediction and the ground truth are shown in figure 3. Comparing this to the vanilla model with relative errors of $L_u^2 = 0.0989$ and $L_f^2 = 0.0728$ we can see that the informed model is more than an order of magnitude more accurate. The predictive mean and the difference to the ground truth can be seen in figure 4. The better prediction is also reflected in the MSEs. The MSEs for the informed model are $MSE_u = . \cdot 10^{-7}$ and $MSE_f = \cdot 10^{-3}$. The MSEs for the vanilla model are $MSE_{u,va} = 9.862 \cdot 10^{-5}$ and $MSE_{f,va} = 0.130$.

The area that strikes the most when looking at the prediction with the informed model are the boundaries of the domain. In figure 5 the slices of the predictive mean together with the ground truth are shown. It is remarkable how precise the model manages to predict the boundaries of the domain, without even using more training points on or near the boundary. It is important to keep the magnitude of the plotted y axis in sight. With this example we can see that the informed models better performance is not only due to the additional hyperparameters. How big the role of the joint GP is, is hard to say, but it is clear that the informed model is notably more accurate than the vanilla model. The most notable difference between the informed and the vanilla model are the predictions at the boundary of the domain. The informed model is able to predict the domain boundaries very accuratly.

6 Conclusion and Outlook

7 Appendix

7.1 Numerical implementation of the Kernel

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* 2006.

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[3] D. Duvenaud. Automatic model construction with gaussian processes. 2014.

[4] Roman Garnett. *Bayesian Optimization.* Cambridge University Press, 2023.

[5] GPy. GPy: A gaussian process framework in python. http://github.com/SheffieldML/GPy, since 2012.

[6] Jochen Görtler, Rebecca Kehlbeck, and Oliver Deussen. A visual exploration of gaussian processes. *Distill*, 2019. https://distill.pub/2019/visual-exploration-gaussian-processes.

[7] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Machine learning of linear differential equations using gaussian processes. *Journal of Computational Physics*, 348:683–693, 2017.

[8] Carl Edward Rasmussen. *Gaussian processes for machine learning.* Adaptive computation and machine learning. 2006.

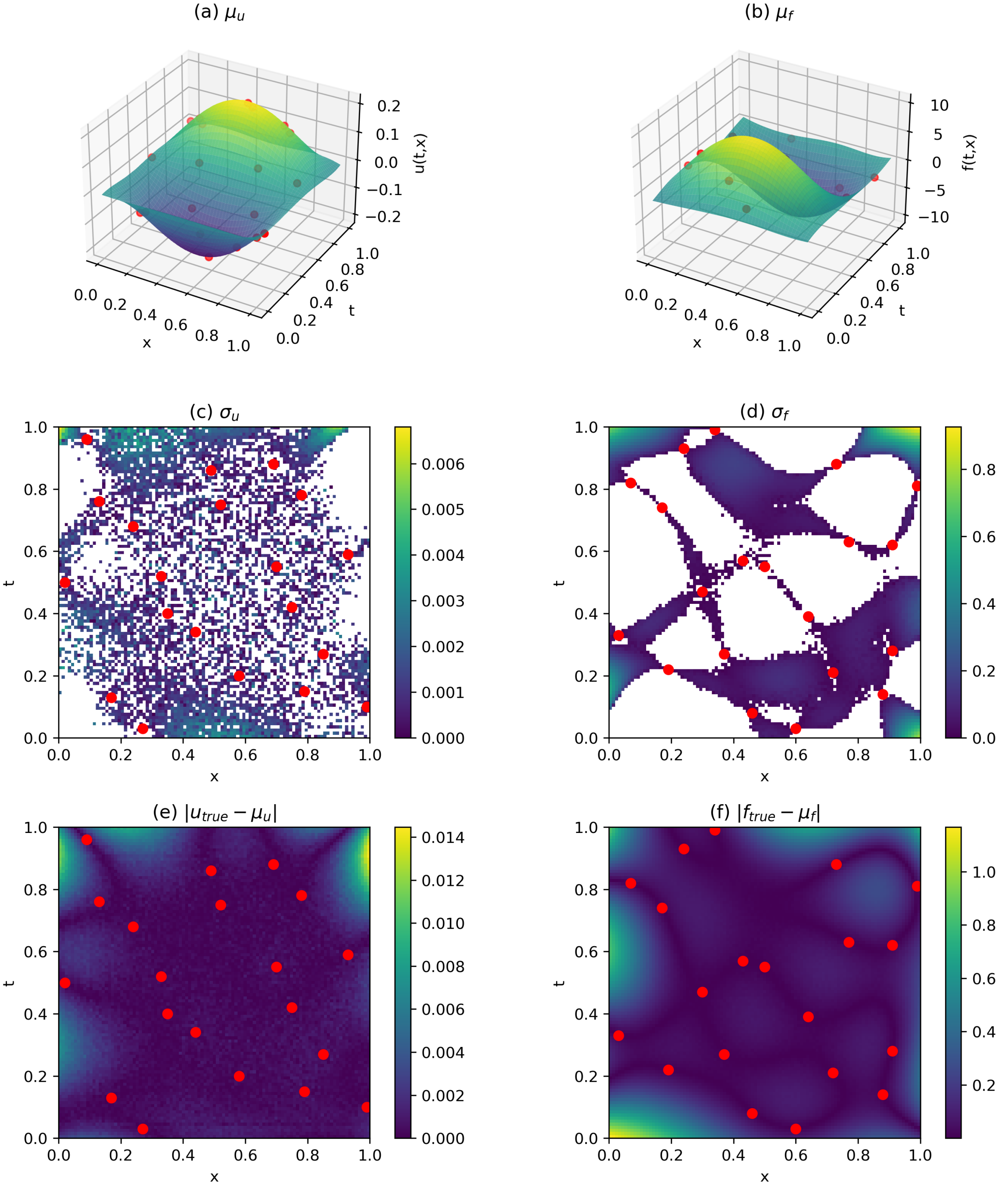

Figure 3: Plots for the informed model. (a) & (b) predictive mean $\boldsymbol{\mu_u}$ and $\boldsymbol{\mu_f}$ with training points in red. (c) & (d) predictive standard deviations $\boldsymbol{\sigma_u}$ and $\boldsymbol{\sigma_f}$. (e) & (f) difference between the predictive mean $\boldsymbol{\mu}$ and the ground truths u and f. The MSEs for the informed model are $MSE_u = \cdot 10^{-7}$ and $MSE_f = \cdot 10^{-3}$. The relative L^2 errors are $L^2_u =$ and $L^2_f =$.

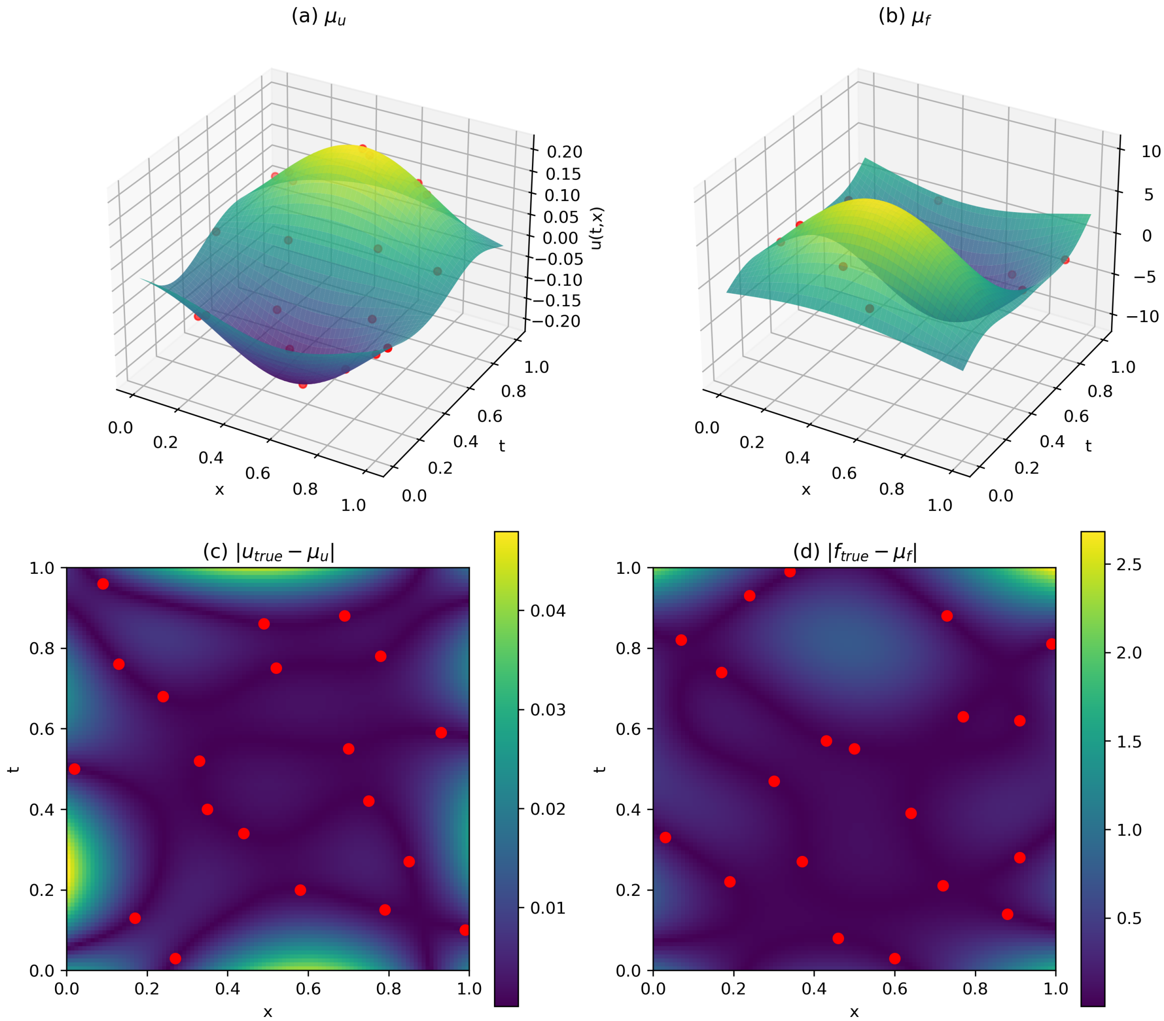

Figure 4: Plots for the vanilla model. (a) & (b) predictive mean $\boldsymbol{\mu_u}$ and $\boldsymbol{\mu_f}$ with training points in red. (c) & (d) difference between the predictive mean $\boldsymbol{\mu}$ and the ground truths u and f. The MSEs for the vanilla model are $MSE_{u,va} = 9.862 \cdot 10^{-5}$ and $MSE_{f,va} = 0.130$. The relative L^2 errors are $L_u^2 = 0.0989$ and $L_f^2 = 0.0728$.

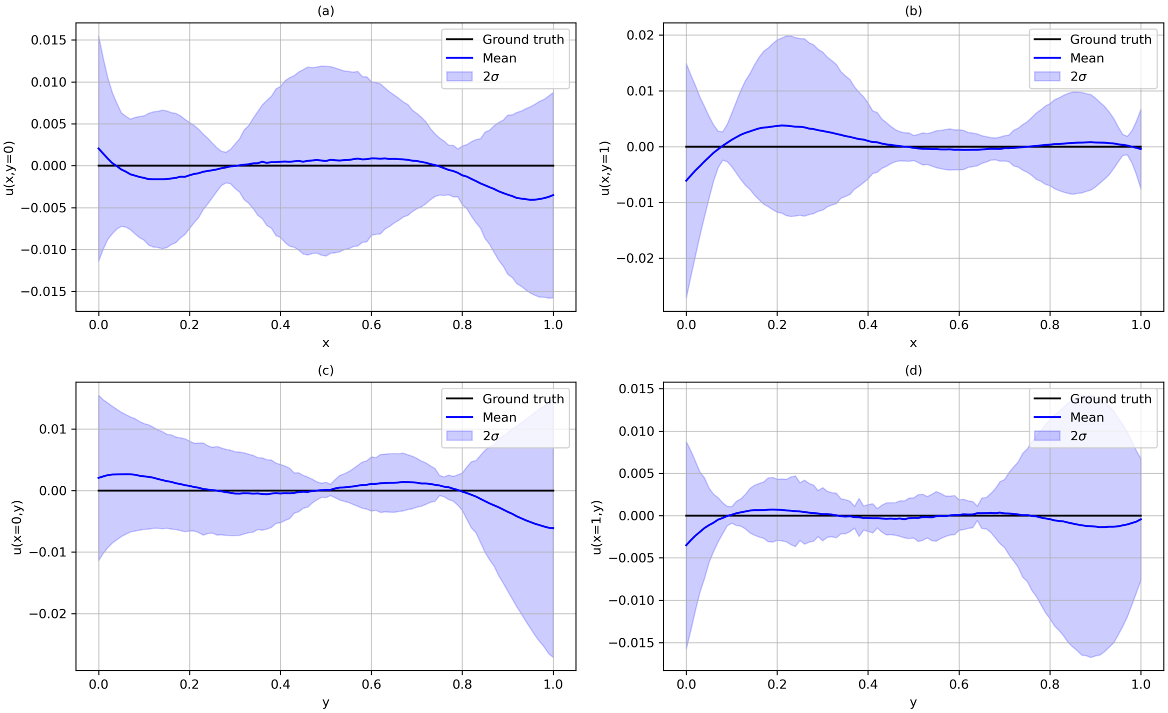

Figure 5: Plots for the investigation of the domain boundaries

[9] S. Singer and J. Nelder. Nelder-Mead algorithm. *Scholarpedia*, 4(7):2928, 2009. revision #91557.

[10] Simo Särkkä. Linear operators and stochastic partial differential equations in gaussian process regression. pages 151–158, 06 2011.