fvGP

Release 4.2.0+7.gda631ba

Marcus Michael Noack

Apr 26, 2024

API

fvGP is an API for flexible HPC single and multi-task Gaussian processes over Euclidean and non-Euclidean (strings, molecules, materials) spaces.

The *GP* class is the core of most the functionality in fvGP and therefore in gpCAM.

fvGP can use HGDL optimization on supercomputers for advanced GP use cases.

The *fvGP* class is a multi-output GP framework which inherits most of its functionality from the *GP* base class.

The fvGP package holds the world-record for exact scalable GPs. It was run on 5 million data points in 2023.

CHAPTER

ONE

GP

class fvgp.GP(*x_data*, *y_data*, *init_hyperparameters=None*, *hyperparameter_bounds=None*, *noise_variances=None*, *compute_device='cpu'*, *gp_kernel_function=None*, *gp_kernel_function_grad=None*, *gp_noise_function=None*, *gp_noise_function_grad=None*, *gp_mean_function=None*, *gp_mean_function_grad=None*, *gp2Scale=False*, *gp2Scale_dask_client=None*, *gp2Scale_batch_size=10000*, *calc_inv=False*, *online=False*, *ram_economy=False*, *args=None*, *info=False*)

This class provides all the tools for a single-task Gaussian Process (GP). Use fvGP for multitask GPs. However, the fvGP class inherits all methods from this class. This class allows for full HPC support for training via the HGDL package.

V ... number of input points

D ... input space dimensionality

N ... arbitrary integers (N1, N2,...)

Parameters

- **x_data** (*np.ndarray or list of tuples*) – The input point positions. Shape (V x D), where D is the *input_space_dim*. If dealing with non-Euclidean inputs x_data should be a list, not a numpy array.
- **y_data** (*np.ndarray*) – The values of the data points. Shape (V,1) or (V).
- **init_hyperparameters** (*np.ndarray, optional*) – Vector of hyperparameters used by the GP initially. This class provides methods to train hyperparameters. The default is a random draw from a uniform distribution within hyperparameter_bounds, with a shape appropriate for the default kernel (D + 1), which is an anisotropic Matern kernel with automatic relevance determination (ARD). If sparse_node or gp2Scale is enabled, the default kernel changes to the anisotropic Wendland kernel.
- **hyperparameter_bounds** (*np.ndarray, optional*) – A 2d numpy array of shape (N x 2), where N is the number of needed hyperparameters. The default is None, in which case the hyperparameter_bounds are estimated from the domain size and the initial y_data. If the data set changes significantly, the hyperparameters and the bounds should be changed/retrained. Initial hyperparameters and bounds can also be set in the train calls. The default only works for the default kernels.
- **noise_variances** (*np.ndarray, optional*) – An numpy array defining the uncertainties/noise in the data *y_data* in form of a point-wise variance. Shape (len(y_data), 1) or (len(y_data)). Note: if no noise_variances are provided here, the gp_noise_function callable will be used; if the callable is not provided, the noise variances will be set to *abs(np.mean(y_data) / 100.0*. If noise covariances are required, also make use of the gp_noise_function.

- **compute_device** (*str, optional*) – One of "cpu" or "gpu", determines how linear system solves are run. The default is "cpu". For "gpu", pytorch has to be installed manually. If gp2Scale is enabled but no kernel is provided, the choice of the compute_device becomes much more important. In that case, the default kernel will be computed on the cpu or the gpu which will significantly change the compute time depending on the compute architecture.
- **gp_kernel_function** (*Callable, optional*) – A symmetric positive semi-definite covariance function (a kernel) that calculates the covariance between data points. It is a function of the form k(x1,x2,hyperparameters, obj). The input x1 is a N1 x D array of positions, x2 is a N2 x D array of positions, the hyperparameters argument is a 1d array of length D+1 for the default kernel and of a different user-defined length for other kernels obj is an *fvgp.GP* instance. The default is a stationary anisotropic kernel (*fvgp.GP.default_kernel*) which performs automatic relevance determination (ARD). The output is a covariance matrix, an N1 x N2 numpy array.
- **gp_kernel_function_grad** (*Callable, optional*) – A function that calculates the derivative of the *gp_kernel_function* with respect to the hyperparameters. If provided, it will be used for local training (optimization) and can speed up the calculations. It accepts as input x1 (a N1 x D array of positions), x2 (a N2 x D array of positions), hyperparameters (a 1d array of length D+1 for the default kernel), and a *fvgp.GP* instance. The default is a finite difference calculation. If 'ram_economy' is True, the function's input is x1, x2, direction (int), hyperparameters (numpy array), and a *fvgp.GP* instance, and the output is a numpy array of shape (len(hps) x N). If 'ram economy' is False,the function's input is x1, x2, hyperparameters, and a *fvgp.GP* instance. The output is a numpy array of shape (len(hyperparameters) x N1 x N2). See 'ram_economy'.
- **gp_mean_function** (*Callable, optional*) – A function that evaluates the prior mean at a set of input position. It accepts as input an array of positions (of shape N1 x D), hyperparameters (a 1d array of length D+1 for the default kernel) and a *fvgp.GP* instance. The return value is a 1d array of length N1. If None is provided, *fvgp.GP._default_mean_function* is used.
- **gp_mean_function_grad** (*Callable, optional*) – A function that evaluates the gradient of the *gp_mean_function* at a set of input positions with respect to the hyperparameters. It accepts as input an array of positions (of size N1 x D), hyperparameters (a 1d array of length D+1 for the default kernel) and a *fvgp.GP* instance. The return value is a 2d array of shape (len(hyperparameters) x N1). If None is provided, either zeros are returned since the default mean function does not depend on hyperparameters, or a finite-difference approximation is used if *gp_mean_function* is provided.
- **gp_noise_function** (*Callable optional*) – The noise function is a callable f(x,hyperparameters,obj) that returns a positive symmetric definite matrix of shape(len(x),len(x)). The input x is a numpy array of shape (N x D). The hyperparameter array is the same that is communicated to mean and kernel functions. The obj is a *fvgp.GP* instance.
- **gp_noise_function_grad** (*Callable, optional*) – A function that evaluates the gradient of the *gp_noise_function* at an input position with respect to the hyperparameters. It accepts as input an array of positions (of size N x D), hyperparameters (a 1d array of length D+1 for the default kernel) and a *fvgp.GP* instance. The return value is a 3-D array of shape (len(hyperparameters) x N x N). If None is provided, either zeros are returned since the default noise function does not depend on hyperparameters. If *gp_noise_function* is provided but no gradient function, a finite-difference approximation will be used. The same rules regarding ram economy as for the kernel definition apply here.
- **gp2Scale** (*bool, optional*) – Turns on gp2Scale. This will distribute the covariance computations across multiple workers. This is an advanced feature for HPC GPs up to 10

million data points. If gp2Scale is used, the default kernel is an anisotropic Wendland kernel which is compactly supported. The noise function will have to return a *scipy.sparse* matrix instead of a numpy array. There are a few more things to consider (read on); this is an advanced option. If no kernel is provided, the compute_device option should be revisited. The kernel will use the specified device to compute covariances. The default is False.

- **gp2Scale_dask_client** (*dask.distributed.Client, optional*) – A dask client for gp2Scale to distribute covariance computations over. Has to contain at least 3 workers. On HPC architecture, this client is provided by the job script. Please have a look at the examples. A local client is used as default.
- **gp2Scale_batch_size** (*int, optional*) – Matrix batch size for distributed computing in gp2Scale. The default is 10000.
- **calc_inv** (*bool, optional*) – If True, the algorithm calculates and stores the inverse of the covariance matrix after each training or update of the dataset or hyperparameters, which makes computing the posterior covariance faster (5-10 times). For larger problems (>2000 data points), the use of inversion should be avoided due to computational instability and costs. The default is False. Note, the training will always use Cholesky or LU decomposition instead of the inverse for stability reasons. Storing the inverse is a good option when the dataset is not too large and the posterior covariance is heavily used.
- **online** (*bool, optional*) – A new setting that allows optimization for online applications. Default=False. If True, calc_inv will be set to true, and the inverse and the logdet() of full dataset will only be computed once in the beginning and after that only updated. This leads to a significant speedup because the most costly aspects of a GP are entirely avoided.
- **ram_economy** (*bool, optional*) – Only of interest if the gradient and/or Hessian of the marginal log_likelihood is/are used for the training. If True, components of the derivative of the marginal log-likelihood are calculated subsequently, leading to a slow-down but much less RAM usage. If the derivative of the kernel (or noise function) with respect to the hyperparameters (gp_kernel_function_grad) is going to be provided, it has to be tailored: for ram_economy=True it should be of the form f(x1[, x2], direction, hyperparameters, obj) and return a 2d numpy array of shape len(x1) x len(x2). If ram_economy=False, the function should be of the form f(x1[, x2,] hyperparameters, obj) and return a numpy array of shape H x len(x1) x len(x2), where H is the number of hyperparameters. CAUTION: This array will be stored and is very large.
- **args** (*any, optional*) – args will be a class attribute and therefore available to kernel, noise and prior mean functions.
- **info** (*bool, optional*) – Provides a way how to see the progress of gp2Scale, Default is False

x_data

Datapoint positions

Type
np.ndarray

y_data

Datapoint values

Type
np.ndarray

noise_variances

Datapoint observation (co)variances

Type
np.ndarray

prior.**hyperparameters**

Current hyperparameters in use.

Type
np.ndarray

prior.**K**

Current prior covariance matrix of the GP

Type
np.ndarray

prior.**m**

Current prior mean vector.

Type
np.ndarray

marginal_density.**KVinv**

If enabled, the inverse of the prior covariance + nose matrix V inv(K+V)

Type
np.ndarray

marginal_density.**KVlogdet**

logdet(K+V)

Type
float

likelihood.**V**

the noise covariance matrix

Type
np.ndarray

update_gp_data(*x_new*, *y_new*, *noise_variances_new=None*, *append=True*)

This function updates the data in the gp object instance. The data will only be overwritten of *overwrite = True*, otherwise the data will be appended. This is a change from earlier versions. Now, the default is not to overwrite the existing data.

Parameters

- **x_new** (*np.ndarray*) – The point positions. Shape (V x D), where D is the *input_space_dim*.
- **y_new** (*np.ndarray*) – The values of the data points. Shape (V,1) or (V).
- **noise_variances_new** (*np.ndarray, optional*) – An numpy array defining the uncertainties in the data *y_data* in form of a point-wise variance. Shape (len(y_data), 1) or (len(y_data)). Note: if no variances are provided here, the noise_covariance callable will be used; if the callable is not provided the noise variances will be set to *abs(np.mean(y_data)) / 100.0*. If you provided a noise function, the noise_variances_new will be ignored.
- **append** (*bool, optional*) – Indication whether to append to or overwrite the existing dataset. Default = True. In the default case, data will be appended.

train(*objective_function=None*, *objective_function_gradient=None*, *objective_function_hessian=None*, *hyperparameter_bounds=None*, *init_hyperparameters=None*, *method='global'*, *pop_size=20*, *tolerance=0.0001*, *max_iter=120*, *local_optimizer='L-BFGS-B'*, *global_optimizer='genetic'*, *constraints=()*, *dask_client=None*)

This function finds the maximum of the log marginal likelihood and therefore trains the GP (synchronously). This can be done on a remote cluster/computer by specifying the method to be 'hgdl' and providing a dask client. However, in that case fvgp.GP.train_async() is preferred. The GP prior will automatically be updated with the new hyperparameters after the training.

Parameters

- **objective_function** (*callable, optional*) – The function that will be MINIMIZED for training the GP. The form of the function is f(hyperparameters=hps) and returns a scalar. This function can be used to train via non-standard user-defined objectives. The default is the negative log marginal likelihood.
- **objective_function_gradient** (*callable, optional*) – The gradient of the function that will be MINIMIZED for training the GP. The form of the function is f(hyperparameters=hps) and returns a vector of len(hps). This function can be used to train via non-standard user-defined objectives. The default is the gradient of the negative log marginal likelihood.
- **objective_function_hessian** (*callable, optional*) – The hessian of the function that will be MINIMIZED for training the GP. The form of the function is f(hyperparameters=hps) and returns a matrix of shape(len(hps),len(hps)). This function can be used to train via non-standard user-defined objectives. The default is the hessian of the negative log marginal likelihood.
- **hyperparameter_bounds** (*np.ndarray, optional*) – A numpy array of shape (D x 2), defining the bounds for the optimization. A 2d numpy array of shape (N x 2), where N is the number of hyperparameters. The default is None, in which case the hyperparameter_bounds are estimated from the domain size and the y_data. If the data set changes significantly, the hyperparameters and the bounds should be changed/retrained. The default only works for the default kernels.
- **init_hyperparameters** (*np.ndarray, optional*) – Initial hyperparameters used as starting location for all optimizers with local component. The default is a random draw from a uniform distribution within the bounds.
- **method** (*str or Callable, optional*) – The method used to train the hyperparameters. The options are *global*, *local*, *hgdl*, *mcmc*, and a callable. The callable gets a *gp.GP* instance and has to return a 1d np.ndarray of hyperparameters. The default is *global* (scipy's differential evolution). If method = "mcmc", the attribute fvgp.GP.mcmc_info is updated and contains convergence and distribution information.
- **pop_size** (*int, optional*) – A number of individuals used for any optimizer with a global component. Default = 20.
- **tolerance** (*float, optional*) – Used as termination criterion for local optimizers. Default = 0.0001.
- **max_iter** (*int, optional*) – Maximum number of iterations for global and local optimizers. Default = 120.
- **local_optimizer** (*str, optional*) – Defining the local optimizer. Default = *L-BFGS-B*, most *scipy.optimize.minimize* functions are permissible.
- **global_optimizer** (*str, optional*) – Defining the global optimizer. Only applicable to method = hgdl. Default = *genetic*

- **constraints** (*tuple of object instances, optional*) – Equality and inequality constraints for the optimization. If the optimizer is *hgdl* see *hgdl.readthedocs.io*. If the optimizer is a scipy optimizer, see the scipy documentation.
- **dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed training if HGDL is used. If None is provided, a new *dask.distributed.Client* instance is constructed.

train_async(*objective_function=None*, *objective_function_gradient=None*, *objective_function_hessian=None*, *hyperparameter_bounds=None*, *init_hyperparameters=None*, *max_iter=10000*, *local_optimizer='L-BFGS-B'*, *global_optimizer='genetic'*, *constraints=()*, *dask_client=None*)

This function asynchronously finds the maximum of the log marginal likelihood and therefore trains the GP. This can be done on a remote cluster/computer by providing a dask client. This function submits the training and returns an object which can be given to *fvgp.GP.update_hyperparameters()*, which will automatically update the GP prior with the new hyperparameters.

Parameters

- **objective_function** (*callable, optional*) – The function that will be MINIMIZED for training the GP. The form of the function is f(hyperparameters=hps) and returns a scalar. This function can be used to train via non-standard user-defined objectives. The default is the negative log marginal likelihood.
- **objective_function_gradient** (*callable, optional*) – The gradient of the function that will be MINIMIZED for training the GP. The form of the function is f(hyperparameters=hps) and returns a vector of len(hps). This function can be used to train via non-standard user-defined objectives. The default is the gradient of the negative log marginal likelihood.
- **objective_function_hessian** (*callable, optional*) – The hessian of the function that will be MINIMIZED for training the GP. The form of the function is f(hyperparameters=hps) and returns a matrix of shape(len(hps),len(hps)). This function can be used to train via non-standard user-defined objectives. The default is the hessian of the negative log marginal likelihood.
- **hyperparameter_bounds** (*np.ndarray, optional*) – A numpy array of shape (D x 2), defining the bounds for the optimization. A 2d numpy array of shape (N x 2), where N is the number of hyperparameters. The default is None, in which case the hyperparameter_bounds are estimated from the domain size and the y_data. If the data set changes significantly, the hyperparameters and the bounds should be changed/retrained. The default only works for the default kernels.
- **init_hyperparameters** (*np.ndarray, optional*) – Initial hyperparameters used as starting location for all optimizers with local component. The default is a random draw from a uniform distribution within the bounds.
- **max_iter** (*int, optional*) – Maximum number of epochs for HGDL. Default = 10000.
- **local_optimizer** (*str, optional*) – Defining the local optimizer. Default = *L-BFGS-B*, most *scipy.optimize.minimize* functions are permissible.
- **global_optimizer** (*str, optional*) – Defining the global optimizer. Only applicable to method = hgdl. Default = *genetic*
- **constraints** (*tuple of hgdl.NonLinearConstraint instances, optional*) – Equality and inequality constraints for the optimization. See *hgdl.readthedocs.io*
- **dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed training if HGDL is used. If None is provided, a new

dask.distributed.Client instance is constructed.

Returns

- Optimization object that can be given to *fvgp.GP.update_hyperparameters()*
- **to update the prior GP** (*object instance*)

`stop_training`(*opt_obj*)

This function stops the training if HGDL is used. It leaves the dask client alive.

Parameters
opt_obj (*HGDL object instance*) – HGDL object instance returned by *fvgp.GP.train_async()*

`kill_training`(*opt_obj*)

This function stops the training if HGDL is used, and kills the dask client.

Parameters
opt_obj (*HGDL object instance*) – HGDL object instance returned by *fvgp.GP.train_async()*

`update_hyperparameters`(*opt_obj*)

This function asynchronously finds the maximum of the marginal log_likelihood and therefore trains the GP. This can be done on a remote cluster/computer by providing a dask client. This function just submits the training and returns an object which can be given to *fvgp.GP.update_hyperparameters()*, which will automatically update the GP prior with the new hyperparameters.

Parameters
opt_obj (*HGDL object instance*) – HGDL object instance returned by *fvgp.GP.train_async()*

Returns
The current hyperparameters

Return type
np.ndarray

`set_hyperparameters`(*hps*)

Function to set hyperparameters.

Parameters
hps (*np.ndarray*) – A 1-d numpy array of hyperparameters.

`get_hyperparameters`()

Function to get the current hyperparameters.

Returns
hyperparameters

Return type
np.ndarray

`get_prior_pdf`()

Function to get the current prior covariance matrix.

Returns
A dictionary containing information about the GP prior distribution

Return type
dict

log_likelihood(*hyperparameters=None*)

Function that computes the marginal log-likelihood

Parameters
hyperparameters (*np.ndarray, optional*) – Vector of hyperparameters of shape (N). If not provided, the covariance will not be recomputed.

Returns
log marginal likelihood of the data

Return type
float

test_log_likelihood_gradient(*hyperparameters*)

Function to test your gradient of the log-likelihood and therefore of the kernel function.

Parameters
hyperparameters (*np.ndarray, optional*) – Vector of hyperparameters of shape (N).

Return type
analytical and finite difference gradient to compare

posterior_mean(*x_pred*, *hyperparameters=None*, *x_out=None*)

This function calculates the posterior mean for a set of input points.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **hyperparameters** (*np.ndarray, optional*) – A numpy array of the correct size depending on the kernel. This is optional in case the posterior mean has to be computed with given hyperparameters, which is, for instance, the case if the posterior mean is a constraint during training. The default is None which means the initialized or trained hyperparameters are used.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space (most often 1).

Returns
Solution points and function values

Return type
dict

posterior_mean_grad(*x_pred*, *hyperparameters=None*, *x_out=None*, *direction=None*)

This function calculates the gradient of the posterior mean for a set of input points.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **hyperparameters** (*np.ndarray, optional*) – A numpy array of the correct size depending on the kernel. This is optional in case the posterior mean has to be computed with given hyperparameters, which is, for instance, the case if the posterior mean is a constraint during training. The default is None which means the initialized or trained hyperparameters are used.

- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.
- **direction** (*int, optional*) – Direction of derivative, If None (default) the whole gradient will be computed.

Returns
Solution

Return type
dict

posterior_covariance(*x_pred*, *x_out=None*, *variance_only=False*, *add_noise=False*)

Function to compute the posterior covariance.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.
- **variance_only** (*bool, optional*) – If True the computation of the posterior covariance matrix is avoided which can save compute time. In that case the return will only provide the variance at the input points. Default = False.
- **add_noise** (*bool, optional*) – If True the noise variances will be added to the posterior variances. Default = False.

Returns
Solution

Return type
dict

posterior_covariance_grad(*x_pred*, *x_out=None*, *direction=None*)

Function to compute the gradient of the posterior covariance.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.
- **direction** (*int, optional*) – Direction of derivative, If None (default) the whole gradient will be computed.

Returns
Solution

Return type
dict

joint_gp_prior(*x_pred*, *x_out=None*)

Function to compute the joint prior over f (at measured locations) and f_pred at x_pred.

Parameters

- **x_pred** (*np.ndarray or list*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Solution

Return type
dict

joint_gp_prior_grad(*x_pred*, *direction*, *x_out=None*)

Function to compute the gradient of the data-informed prior.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **direction** (*int*) – Direction of derivative.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Solution

Return type
dict

entropy(*S*)

Function computing the entropy of a normal distribution res = entropy(S); S is a 2d np.ndarray array, a covariance matrix which is non-singular.

Parameters
S (*np.ndarray*) – A covariance matrix.

Returns
Entropy

Return type
float

gp_entropy(*x_pred*, *x_out=None*)

Function to compute the entropy of the gp prior probability distribution.

Parameters
x_pred (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces. Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Entropy

Return type
float

gp_entropy_grad(*x_pred*, *direction*, *x_out=None*)

Function to compute the gradient of entropy of the prior in a given direction.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **direction** (*int*) – Direction of derivative.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Entropy gradient in given direction

Return type
float

kl_div_grad(*mu1*, *dmu1dx*, *mu2*, *S1*, *dS1dx*, *S2*)

This function computes the gradient of the KL divergence between two normal distributions when the gradients of the mean and covariance are given. a = kl_div(mu1, dmudx,mu2, S1, dS1dx, S2); S1, S2 are 2d numpy arrays, matrices have to be non-singular, mu1, mu2 are mean vectors, given as 2d arrays

kl_div(*mu1*, *mu2*, *S1*, *S2*)

Function to compute the KL divergence between two Gaussian distributions.

Parameters

- **mu1** (*np.ndarray*) – Mean vector of distribution 1.
- **mu2** (*np.ndarray*) – Mean vector of distribution 2.
- **S1** (*np.ndarray*) – Covariance matrix of distribution 1.
- **S2** (*np.ndarray*) – Covariance matrix of distribution 2.

Returns
KL divergence

Return type
float

gp_kl_div(*x_pred*, *comp_mean*, *comp_cov*, *x_out=None*)

Function to compute the kl divergence of a posterior at given points and a given normal distribution.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **comp_mean** (*np.ndarray*) – Comparison mean vector for KL divergence. len(comp_mean) = len(x_pred)
- **comp_cov** (*np.ndarray*) – Comparison covariance matrix for KL divergence. shape(comp_cov) = (len(x_pred),len(x_pred))
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Solution

Return type
dict

gp_kl_div_grad(*x_pred*, *comp_mean*, *comp_cov*, *direction*, *x_out=None*)

Function to compute the gradient of the kl divergence of a posterior at given points.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **comp_mean** (*np.ndarray*) – Comparison mean vector for KL divergence. len(comp_mean) = len(x_pred)
- **comp_cov** (*np.ndarray*) – Comparison covariance matrix for KL divergence. shape(comp_cov) = (len(x_pred),len(x_pred))
- **direction** (*int*) – The direction in which the gradient will be computed.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Solution

Return type
dict

mutual_information(*joint*, *m1*, *m2*)

Function to calculate the mutual information between two normal distributions, which is equivalent to the KL divergence(joint, marginal1 * marginal1).

Parameters

- **joint** (*np.ndarray*) – The joint covariance matrix.
- **m1** (*np.ndarray*) – The first marginal distribution
- **m2** (*np.ndarray*) – The second marginal distribution

Returns
Mutual information

Return type
float

gp_mutual_information(*x_pred*, *x_out=None*)

Function to calculate the mutual information between the random variables f(x_data) and f(x_pred). The mutual information is always positive, as it is a KL divergence, and is bounded from below by 0. The maxima are expected at the data points. Zero is expected far from the data support. :param x_pred: A numpy array of shape (V x D), interpreted as an array of input point positions or a list for

GPs on non-Euclidean input spaces.

Parameters
x_out (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Solution

Return type
dict

gp_total_correlation(*x_pred*, *x_out=None*)

Function to calculate the interaction information between the random variables f(x_data) and f(x_pred). This is the mutual information of each f(x_pred) with f(x_data). It is also called the Multiinformation. It is best used when several prediction points are supposed to be mutually aware. The total correlation is always positive, as it is a KL divergence, and is bounded from below by 0. The maxima are expected at the data points. Zero is expected far from the data support.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Solution – Total correlation between prediction points, as a collective.

Return type
dict

gp_relative_information_entropy(*x_pred*, *x_out=None*)

Function to compute the KL divergence and therefore the relative information entropy of the prior distribution defined over predicted function values and the posterior distribution. The value is a reflection of how much information is predicted to be gained through observing a set of data points at x_pred.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Solution – Relative information entropy of prediction points, as a collective.

Return type
dict

gp_relative_information_entropy_set(*x_pred*, *x_out=None*)

Function to compute the KL divergence and therefore the relative information entropy of the prior distribution over predicted function values and the posterior distribution. The value is a reflection of how much information is predicted to be gained through observing each data point in x_pred separately, not all at once as in *gp_relative_information_entropy*.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Solution – Relative information entropy of prediction points, but not as a collective.

Return type
dict

`posterior_probability`(*x_pred*, *comp_mean*, *comp_cov*, *x_out=None*)

Function to compute probability of a probabilistic quantity of interest, given the GP posterior at given points.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **comp_mean** (*np.ndarray*) – A vector of mean values, same length as x_pred.
- **comp_cov** (*np.nparray*) – Covariance matrix, in R^{len(x_pred) times len(x_pred)}
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Solution – The probability of a probabilistic quantity of interest, given the GP posterior at a given point.

Return type
dict

`posterior_probability_grad`(*x_pred*, *comp_mean*, *comp_cov*, *direction*, *x_out=None*)

Function to compute the gradient of the probability of a probabilistic quantity of interest, given the GP posterior at a given point.

Parameters

- **x_pred** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions or a list for GPs on non-Euclidean input spaces.
- **comp_mean** (*np.ndarray*) – A vector of mean values, same length as x_pred.
- **comp_cov** (*np.nparray*) – Covariance matrix, in R^{len(x_pred) times len(x_pred)}
- **direction** (*int*) – The direction to compute the gradient in.
- **x_out** (*np.ndarray, optional*) – Output coordinates in case of multitask GP use; a numpy array of size (N x L), where N is the number of output points, and L is the dimensionality of the output space.

Returns
Solution – The gradient of the probability of a probabilistic quantity of interest, given the GP posterior at a given point.

Return type
dict

`normalize_y_data`(*y_data*)

Function to normalize the y_data. The user is responsible to normalize the noise accordingly. This function will not update the object instance.

Parameters
y_data (*np.ndarray*) – Numpy array of shape (U).

crps(*x_test*, *y_test*)

This function calculates the continuous rank probability score. Note that in the multitask setting the user should perform their input point transformation beforehand.

Parameters

- **x_test** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions.
- **y_test** (*np.ndarray*) – A numpy array of shape (V x 1). These are the y data to compare against.

Returns
CRPS

Return type
float

rmse(*x_test*, *y_test*)

This function calculates the root mean squared error. Note that in the multitask setting the user should perform their input point transformation beforehand.

Parameters

- **x_test** (*np.ndarray*) – A numpy array of shape (V x D), interpreted as an array of input point positions.
- **y_test** (*np.ndarray*) – A numpy array of shape (V x 1). These are the y data to compare against

Returns
RMSE

Return type
float

make_2d_x_pred(*bx*, *by*, *resx=100*, *resy=100*)

This is a purely convenience-driven function calculating prediction points on a grid. :param bx: A numpy array of shape (2) defining lower and upper bounds in x direction. :type bx: np.ndarray :param by: A numpy array of shape (2) defining lower and upper bounds in y direction. :type by: np.ndarray :param resx: Resolution in x direction. Default = 100. :type resx: int, optional :param resy: Resolution in y direction. Default = 100. :type resy: int, optional

Returns
prediction points

Return type
np.ndarray

make_1d_x_pred(*b*, *res=100*)

This is a purely convenience-driven function calculating prediction points on a 1d grid.

Parameters

- **b** (*np.ndarray*) – A numpy array of shape (2) defineing lower and upper bounds
- **res** (*int, optional*) – Resolution. Default = 100

Returns
prediction points

Return type
np.ndarray

CHAPTER

TWO

FVGP

class fvgp.**fvGP**(*x_data*, *y_data*, *init_hyperparameters=None*, *hyperparameter_bounds=None*, *output_positions=None*, *noise_variances=None*, *compute_device='cpu'*, *gp_kernel_function=None*, *gp_deep_kernel_layer_width=5*, *gp_kernel_function_grad=None*, *gp_noise_function=None*, *gp_noise_function_grad=None*, *gp_mean_function=None*, *gp_mean_function_grad=None*, *gp2Scale=False*, *gp2Scale_dask_client=None*, *gp2Scale_batch_size=10000*, *calc_inv=False*, *online=False*, *ram_economy=False*, *args=None*, *info=False*)

This class provides all the tools for a multitask Gaussian Process (GP). This class allows for full HPC support for training. After initialization, this class provides all the methods described for the GP class.

V … number of input points

Di… input space dimensionality

Do… output space dimensionality

No… number of outputs

N … arbitrary integers (N1, N2,…)

The main logic of fvGP is that any multitask GP is just a single-task GP over a Cartesian product space of input and output space, as long as the kernel is flexible enough, so prepare to work on your kernel. This is the best way to give the user optimal control and power. At various instances, for instances prior-mean function, noise function, and kernel function definitions, you will see that the input ``x" is defined over this combined space. For example, if your input space is a Euclidean 2d space and your output is labelled [0,1], the input to the mean, kernel, and noise function might be

x =

[[0.2, 0.3,0],[0.9,0.6,0],

[0.2, 0.3,1],[0.9,0.6,1]]

This has to be understood and taken into account when customizing fvGP for multitask use.

Parameters

- **x_data** (*np.ndarray*) – The input point positions. Shape (V x D), where D is the *input_space_dim*.
- **y_data** (*np.ndarray*) – The values of the data points. Shape (V,No).
- **init_hyperparameters** (*np.ndarray, optional*) – Vector of hyperparameters used by the GP initially. This class provides methods to train hyperparameters. The default is an array that specifies the right number of initial hyperparameters for the default kernel, which is a deep kernel with two layers of width fvgp.fvGP.gp_deep_kernel_layer_width. If you specify another kernel, please provide init_hyperparameters.

- **hyperparameter_bounds** (*np.ndarray, optional*) – A 2d numpy array of shape (N x 2), where N is the number of needed hyperparameters. The default is None, in that case hyperparameter_bounds have to be specified in the train calls or default bounds are used. Those only work for the default kernel.
- **output_positions** (*np.ndarray, optional*) – A 2-D numpy array of shape (U x output_number), so that for each measurement position, the outputs are clearly defined by their positions in the output space. The default is np.array([[0,1,2,3,...,output_number - 1],[0,1,2,3,...,output_number - 1],...]).
- **noise_variances** (*np.ndarray, optional*) – An numpy array defining the uncertainties/noise in the data *y_data* in form of a point-wise variance. Shape y_data.shape. Note: if no noise_variances are provided here, the gp_noise_function callable will be used; if the callable is not provided, the noise variances will be set to *abs(np.mean(y_data) / 100.0*. If noise covariances are required, also make use of the gp_noise_function.
- **compute_device** (*str, optional*) – One of "cpu" or "gpu", determines how linear system solves are run. The default is "cpu". For "gpu", pytorch has to be installed manually. If gp2Scale is enabled but no kernel is provided, the choice of the compute_device becomes much more important. In that case, the default kernel will be computed on the cpu or the gpu which will significantly change the compute time depending on the compute architecture.
- **gp_kernel_function** (*Callable, optional*) – A symmetric positive semi-definite covariance function (a kernel) that calculates the covariance between data points. It is a function of the form k(x1,x2,hyperparameters, obj). The input x1 is a N1 x Di+Do array of positions, x2 is a N2 x Di+Do array of positions, the hyperparameters argument is a 1d array of length N depending on how many hyperparameters are initialized, and obj is an *fvgp.GP* instance. The default is a deep kernel with 2 hidden layers and a width of fvgp.fvGP.gp_deep_kernel_layer_width.
- **gp_deep_kernel_layer_width** (*int, optional*) – If no kernel is provided, fvGP will use a deep kernel of depth 2 and width gp_deep_kernel_layer_width. If a user defined kernel is provided this parameter is irrelevant. The default is 5.
- **gp_kernel_function_grad** (*Callable, optional*) – A function that calculates the derivative of the *gp_kernel_function* with respect to the hyperparameters. If provided, it will be used for local training (optimization) and can speed up the calculations. It accepts as input x1 (a N1 x Di+Do array of positions), x2 (a N2 x Di+Do array of positions), hyperparameters, and a *fvgp.GP* instance. The default is a finite difference calculation. If 'ram_economy' is True, the function's input is x1, x2, direction (int), hyperparameters (numpy array), and a *fvgp.GP* instance, and the output is a numpy array of shape (len(hps) x N). If 'ram economy' is False,the function's input is x1, x2, hyperparameters, and a *fvgp.GP* instance. The output is a numpy array of shape (len(hyperparameters) x N1 x N2). See 'ram_economy'.
- **gp_mean_function** (*Callable, optional*) – A function that evaluates the prior mean at a set of input position. It accepts as input an array of positions (of shape N1 x Di+Do), hyperparameters and a *fvgp.GP* instance. The return value is a 1d array of length N1. If None is provided, *fvgp.GP._default_mean_function* is used.
- **gp_mean_function_grad** (*Callable, optional*) – A function that evaluates the gradient of the ``gp_mean_function" at a set of input positions with respect to the hyperparameters. It accepts as input an array of positions (of size N1 x Di+Do), hyperparameters and a *fvgp.GP* instance. The return value is a 2d array of shape (len(hyperparameters) x N1). If None is provided, either zeros are returned since the default mean function does not depend on hyperparameters, or a finite-difference approximation is used if ``gp_mean_function" is provided.
- **gp_noise_function** (*Callable optional*) – The noise function is a callable

f(x,hyperparameters,obj) that returns a positive symmetric definite matrix of shape(len(x),len(x)). The input x is a numpy array of shape (N x Di+Do). The hyperparameter array is the same that is communicated to mean and kernel functions. The obj is a fvgp.fvGP instance.

- **gp_noise_function_grad** (*Callable, optional*) – A function that evaluates the gradient of the ``gp_noise_function" at an input position with respect to the hyperparameters. It accepts as input an array of positions (of size N x Di+Do), hyperparameters (a 1d array of length D+1 for the default kernel) and a *fvgp.GP* instance. The return value is a 3-D array of shape (len(hyperparameters) x N x N). If None is provided, either zeros are returned since the default noise function does not depend on hyperparameters. If ``gp_noise_function" is provided but no gradient function, a finite-difference approximation will be used. The same rules regarding ram economy as for the kernel definition apply here.
- **gp2Scale** (*bool, optional*) – Turns on gp2Scale. This will distribute the covariance computations across multiple workers. This is an advanced feature for HPC GPs up to 10 million datapoints. If gp2Scale is used, the default kernel is an anisotropic Wendland kernel which is compactly supported. The noise function will have to return a scipy.sparse matrix instead of a numpy array. There are a few more things to consider (read on); this is an advanced option. If no kernel is provided, the compute_device option should be revisited. The kernel will use the specified device to compute covariances. The default is False.
- **gp2Scale_dask_client** (*dask.distributed.Client, optional*) – A dask client for gp2Scale to distribute covariance computations over. Has to contain at least 3 workers. On HPC architecture, this client is provided by the jobscript. Please have a look at the examples. A local client is used as default.
- **gp2Scale_batch_size** (*int, optional*) – Matrix batch size for distributed computing in gp2Scale. The default is 10000.
- **calc_inv** (*bool, optional*) – If True, the algorithm calculates and stores the inverse of the covariance matrix after each training or update of the dataset or hyperparameters, which makes computing the posterior covariance faster (5-10 times). For larger problems (>2000 data points), the use of inversion should be avoided due to computational instability and costs. The default is False. Note, the training will always use Cholesky or LU decomposition instead of the inverse for stability reasons. Storing the inverse is a good option when the dataset is not too large and the posterior covariance is heavily used.
- **online** (*bool, optional*) – A new setting that allows optimization for online applications. Default=False. If True, calc_inv will be set to true, and the inverse and the logdet() of full dataset will only be computed once in the beginning and after that only updated. This leads to a significant speedup because the most costly aspects of a GP are entirely avoided.
- **ram_economy** (*bool, optional*) – Only of interest if the gradient and/or Hessian of the marginal log_likelihood is/are used for the training. If True, components of the derivative of the marginal log-likelihood are calculated subsequently, leading to a slow-down but much less RAM usage. If the derivative of the kernel (or noise function) with respect to the hyperparameters (gp_kernel_function_grad) is going to be provided, it has to be tailored: for ram_economy=True it should be of the form f(x1[, x2], direction, hyperparameters, obj) and return a 2d numpy array of shape len(x1) x len(x2). If ram_economy=False, the function should be of the form f(x1[, x2,] hyperparameters, obj) and return a numpy array of shape H x len(x1) x len(x2), where H is the number of hyperparameters. CAUTION: This array will be stored and is very large.
- **args** (*any, optional*) – args will be a class attribute and therefore available to kernel, noise and prior mean functions.
- **info** (*bool, optional*) – Provides a way how to see the progress of gp2Scale, Default is

False

x_data

Datapoint positions

Type
np.ndarray

y_data

Datapoint values

Type
np.ndarray

fvgp_x_data

Datapoint positions as seen by fvgp

Type
np.ndarray

fvgp_y_data

Datapoint values as seen by fvgp

Type
np.ndarray

noise_variances

Datapoint observation (co)variances.

Type
np.ndarray

prior.**hyperparameters**

Current hyperparameters in use.

Type
np.ndarray

prior.**K**

Current prior covariance matrix of the GP

Type
np.ndarray

prior.**m**

Current prior mean vector.

Type
np.ndarray

marginal_density.**KVinv**

If enabled, the inverse of the prior covariance + nose matrix V inv(K+V)

Type
np.ndarray

marginal_density.**KVlogdet**

logdet(K+V)

Type
float

likelihood.**V**

the noise covariance matrix

Type
np.ndarray

update_gp_data(*x_new*, *y_new*, *noise_variances_new=None*, *append=True*, *output_positions_new=None*)

This function updates the data in the gp object instance. The data will NOT be appended but overwritten! Please provide the full updated data set.

Parameters

- **x_new** (*np.ndarray*) – The point positions. Shape (V x D), where D is the *input_space_dim*.
- **y_new** (*np.ndarray*) – The values of the data points. Shape (V,1) or (V).
- **noise_variances_new** (*np.ndarray, optional*) – An numpy array defining the uncertainties in the data *y_data* in form of a point-wise variance. Shape (len(y_data), 1) or (len(y_data)). Note: if no variances are provided here, the noise_covariance callable will be used; if the callable is not provided the noise variances will be set to *abs(np.mean(y_data)) / 100.0*. If you provided a noise function, the noise_variances_new will be ignored.
- **append** (*bool, optional*) – Indication whether to append to or overwrite the existing dataset. Default = True. In the default case, data will be appended.
- **output_positions_new** (*np.ndarray, optional*) – A 3-D numpy array of shape (U x output_number x output_dim), so that for each measurement position, the outputs are clearly defined by their positions in the output space. The default is np.array([[0,1,2,3,...,output_number - 1],[0,1,2,3,...,output_number - 1],...]).

CHAPTER

THREE

LOGGING

The fvGP package uses the Loguru library for sophisticated log management. This follows similar principles as the vanilla Python logging framework, with additional functionality and performance benefits. You may want to enable logging in interactive use, or for debugging purposes.

3.1 Configuring logging

To enable logging in fvGP:

```
from loguru import logger
logger.enable("fvgp")
```

To configure the logging level:

```
logger.add(sys.stdout, filter="fvgp", level="INFO")
```

See Python's reference on levels for more info.

To log to a file:

```
logger.add("file_{time}.log")
```

Loguru provides many further options for configuration.

CHAPTER

FOUR

FVGP SINGLE-TASK TEST

This is the new test for fvgp version 4.2.0 and later.

```
#!pip install fvgp==4.2.0
```

4.1 Setup

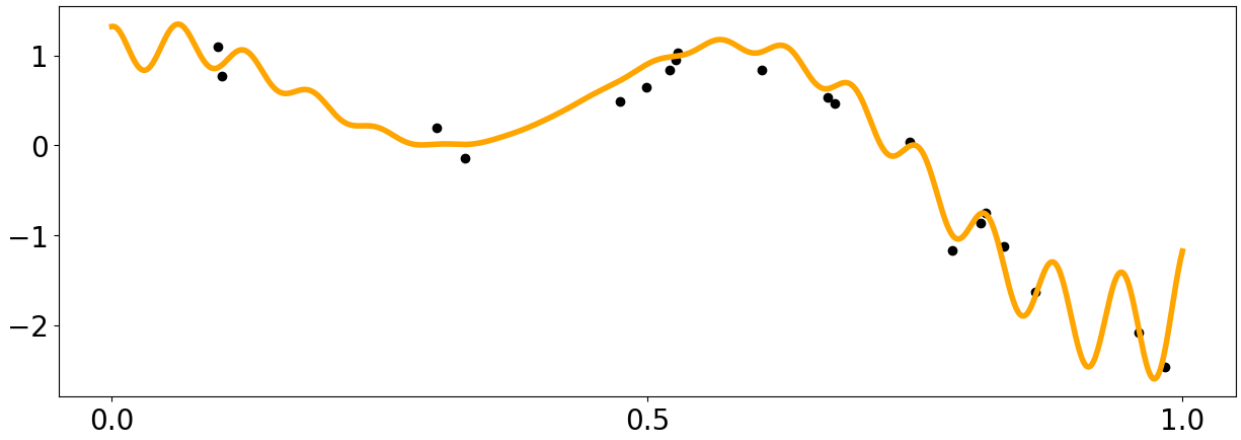
```
import numpy as np
import matplotlib.pyplot as plt
from fvgp import GP
import time


%load_ext autoreload
%autoreload 2
```

```
from itertools import product
x_pred1D = np.linspace(0,1,1000).reshape(-1,1)
```

4.2 Data

```
x = np.linspace(0,600,1000)
def f1(x):
    return np.sin(5. * x) + np.cos(10. * x) + (2.* (x-0.4)**2) * np.cos(100. * x)

x_data = np.random.rand(20).reshape(-1,1)
y_data = f1(x_data[:,0]) + (np.random.rand(len(x_data))-0.5) * 0.5

plt.figure(figsize = (15,5))
plt.xticks([0.,0.5,1.0])
plt.yticks([-2,-1,0.,1])
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.plot(x_pred1D,f1(x_pred1D), color = 'orange', linewidth = 4)
plt.scatter(x_data[:,0],y_data, color = 'black')
```

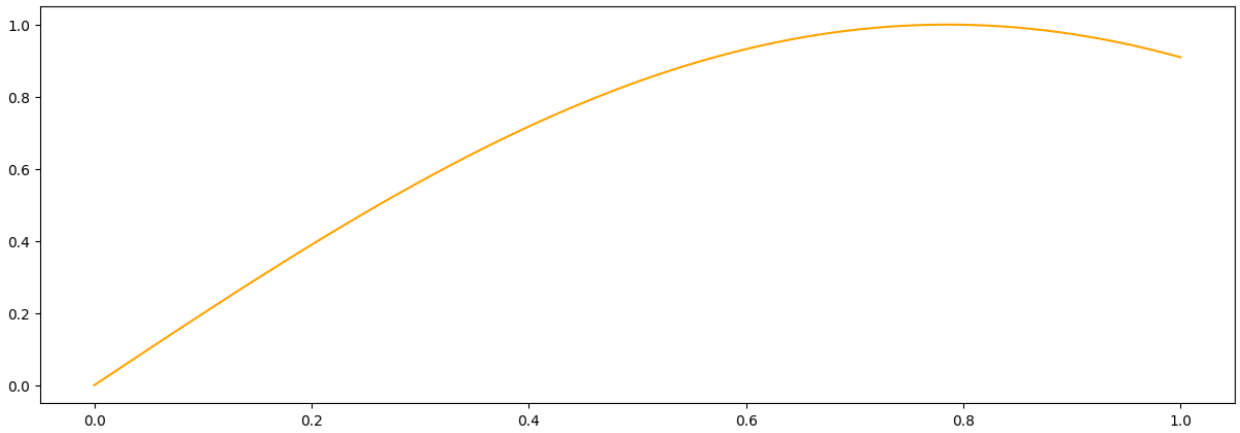
```
<matplotlib.collections.PathCollection at 0x7fc8c9d8c250>
```



4.3 Customizing a Gaussian Process

```
from fvgp.gp_kernels import *
def my_noise(x,hps,obj):
    #This is a simple noise function but can be arbitrarily complex using many␣
↪hyperparameters.
    #The noise function always has to return a matrix, because the noise can have␣
↪covariances.
    return np.diag(np.zeros((len(x))) + hps[2])

#stationary
def skernel(x1,x2,hps,obj):
    #The kernel follows the mathematical definition of a kernel. This
    #means there is no limit to the variety of kernels you can define.
    d = get_distance_matrix(x1,x2)
    return hps[0] * matern_kernel_diff1(d,hps[1])


def meanf(x, hps, obj):
    #This ios a simple mean function but it can be arbitrarily complex using many␣
↪hyperparameters.
    return np.sin(hps[3] * x[:,0])
#it is a good idea to plot the prior mean function to make sure we did not mess up
plt.figure(figsize = (15,5))
plt.plot(x_pred1D,meanf(x_pred1D, np.array([1.,1.,5.0,2.]), None), color = 'orange',␣
↪label = 'task1')
```

```
[<matplotlib.lines.Line2D at 0x7fc8c9e0feb0>]
```



4.4 Initialization and different training options

```
my_gp1 = GP(x_data,y_data,
            init_hyperparameters = np.ones((4))/10.,  # we need enough of those for␣
↪kernel, noise and prior mean functions
            noise_variances=np.ones(y_data.shape) * 0.01, #provding noise variances and␣
↪a noise function will raise a warning
            #hyperparameter_bounds= hps_bounds,
            compute_device='cpu',
            gp_kernel_function=skernel,
            gp_kernel_function_grad=None,
            gp_mean_function=meanf,
            gp_mean_function_grad=None,
            #gp_noise_function=my_noise,
            gp2Scale = False,
            calc_inv=False,
            ram_economy=False,
            args=None,
            )


hps_bounds = np.array([[0.01,10.], #signal variance for the kernel
                       [0.01,10.], #length scale for the kernel
                       [0.001,0.1],  #noise
                       [0.01,1.]  #mean
                      ])
my_gp1.update_gp_data(x_data, y_data, noise_variances_new=np.ones(y_data.shape) * 0.01)
print("Standard Training")
my_gp1.train(hyperparameter_bounds=hps_bounds)
print("Global Training")
my_gp1.train(hyperparameter_bounds=hps_bounds, method='global')
print("hps: ", my_gp1.get_hyperparameters())
print("Local Training")
my_gp1.train(hyperparameter_bounds=hps_bounds, method='local')
print(my_gp1.get_hyperparameters())
print("MCMC Training")
```

(continues on next page)

(continued from previous page)

```
my_gp1.train(hyperparameter_bounds=hps_bounds, method='mcmc', max_iter=1000)
print("HGDL Training")
print(my_gp1.get_hyperparameters())
my_gp1.train(hyperparameter_bounds=hps_bounds, method='hgdl', max_iter=10)
```

```
/home/marcus/Coding/fvGP/fvgp/gp.py:261: UserWarning: hyperparameter_bounds not provided.
↪ They will have to be provided in the training call.
  warnings.warn("hyperparameter_bounds not provided. "
```

```
Standard Training
Global Training
hps:  [1.32465417 0.08595755 0.03271898 0.01001098]
Local Training
[1.32465008 0.0860051  0.03271898 0.01001098]
MCMC Training
HGDL Training
[1.00494338e-02 9.14225137e+00 8.25255813e-03 5.24319320e-01]
```

```
/home/marcus/Coding/fvGP/fvgp/gp_training.py:335: RuntimeWarning: Method L-BFGS-B does␣
↪not use Hessian information (hess).
  OptimumEvaluation = minimize(
/home/marcus/VirtualEnvironments/fvgp_dev/lib/python3.10/site-packages/hgdl/local_
↪methods/local_optimizer.py:95: RuntimeWarning: Method L-BFGS-B does not use Hessian␣
↪information (hess).
  res = minimize(d.func, x0, args=args, method=method, jac=grad, hess=hess,
/home/marcus/VirtualEnvironments/fvgp_dev/lib/python3.10/site-packages/hgdl/local_
↪methods/local_optimizer.py:95: RuntimeWarning: Method L-BFGS-B does not use Hessian␣
↪information (hess).
  res = minimize(d.func, x0, args=args, method=method, jac=grad, hess=hess,
/home/marcus/VirtualEnvironments/fvgp_dev/lib/python3.10/site-packages/hgdl/local_
↪methods/local_optimizer.py:95: RuntimeWarning: Method L-BFGS-B does not use Hessian␣
↪information (hess).
  res = minimize(d.func, x0, args=args, method=method, jac=grad, hess=hess,
```

4.4.1 More advanced: Asynchronous training

Train asynchronously on a remote server or locally. You can also start a bunch of different trainings on different computers. This training will continue without any signs of life until you call 'my_gp1.stop_training(opt_obj)'

```
opt_obj = my_gp1.train_async(hyperparameter_bounds=hps_bounds)
```

```
#the result won't change much (or at all) since this is such a simple optimization
for i in range(10):
    time.sleep(2)
    my_gp1.update_hyperparameters(opt_obj)
    print(my_gp1.get_hyperparameters())
    print("")
```

```
/home/marcus/VirtualEnvironments/fvgp_dev/lib/python3.10/site-packages/hgdl/local_
↪methods/local_optimizer.py:95: RuntimeWarning: Method L-BFGS-B does not use Hessian␣
```

(continues on next page)

(continued from previous page)

```
↪information (hess).
  res = minimize(d.func, x0, args=args, method=method, jac=grad, hess=hess,
/home/marcus/VirtualEnvironments/fvgp_dev/lib/python3.10/site-packages/hgdl/local_
↪methods/local_optimizer.py:95: RuntimeWarning: Method L-BFGS-B does not use Hessian␣
↪information (hess).
  res = minimize(d.func, x0, args=args, method=method, jac=grad, hess=hess,
/home/marcus/VirtualEnvironments/fvgp_dev/lib/python3.10/site-packages/hgdl/local_
↪methods/local_optimizer.py:95: RuntimeWarning: Method L-BFGS-B does not use Hessian␣
↪information (hess).
  res = minimize(d.func, x0, args=args, method=method, jac=grad, hess=hess,
```

```
[1.32121898 0.08591372 0.04376501 0.01      ]

[1.32121898 0.08591372 0.04376501 0.01      ]

[1.32121898 0.08591372 0.04376501 0.01      ]

[1.32121898 0.08591372 0.04376501 0.01      ]

[1.32121898 0.08591372 0.04376501 0.01      ]

[1.32121898 0.08591372 0.04376501 0.01      ]

[1.32121898 0.08591372 0.04376501 0.01      ]

[1.32121898 0.08591372 0.04376501 0.01      ]

[1.32121898 0.08591372 0.04376501 0.01      ]

[1.32121898 0.08591372 0.04376501 0.01      ]
```

```
my_gp1.stop_training(opt_obj)
```

4.5 The Result

```
#let's make a prediction
x_pred = np.linspace(0,1,1000)


%timeit mean1 = my_gp1.posterior_mean(x_pred.reshape(-1,1))["f(x)"]
%timeit var1 =  my_gp1.posterior_covariance(x_pred.reshape(-1,1), variance_only=False,␣
↪add_noise=False)["v(x)"]
%timeit var1 =  my_gp1.posterior_covariance(x_pred.reshape(-1,1), variance_only=True,␣
↪add_noise=False)["v(x)"]

mean1 = my_gp1.posterior_mean(x_pred.reshape(-1,1))["f(x)"]
var1 =  my_gp1.posterior_covariance(x_pred.reshape(-1,1), variance_only=False, add_
↪noise=True)["v(x)"]
plt.figure(figsize = (16,10))
```

(continues on next page)

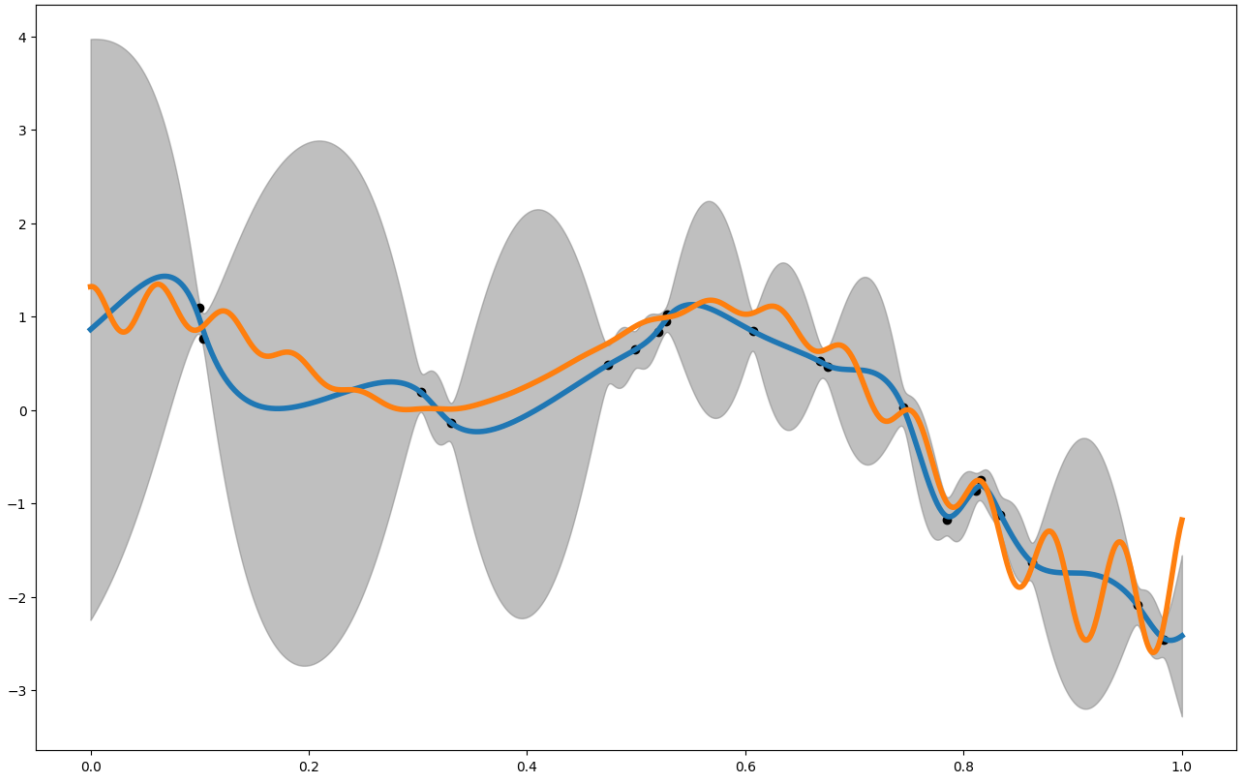
(continued from previous page)

```
plt.plot(x_pred,mean1, label = "posterior mean", linewidth = 4)
plt.plot(x_pred1D,f1(x_pred1D), label = "latent function", linewidth = 4)
plt.fill_between(x_pred, mean1 - 3. * np.sqrt(var1), mean1 + 3. * np.sqrt(var1), alpha =␣
↪0.5, color = "grey", label = "var")
plt.scatter(x_data,y_data, color = 'black')


##looking at some validation metrics
print(my_gp1.rmse(x_pred1D,f1(x_pred1D)))
print(my_gp1.crps(x_pred1D,f1(x_pred1D)))

#1.58 ms ± 120 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
#87 ms ± 7.16 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
1.53 ms ± 189 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
83.2 ms ± 6.8 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
69 ms ± 8.17 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
45.50482130674711
```

```
/home/marcus/Coding/fvGP/fvgp/gp_posterior.py:159: UserWarning: Noise could not be added,
↪ you did not provide a noise callable at initialization
  warnings.warn("Noise could not be added, you did not provide a noise callable at␣
↪initialization")
```

```
(1.028113817193425, 0.9067912229063404)
```

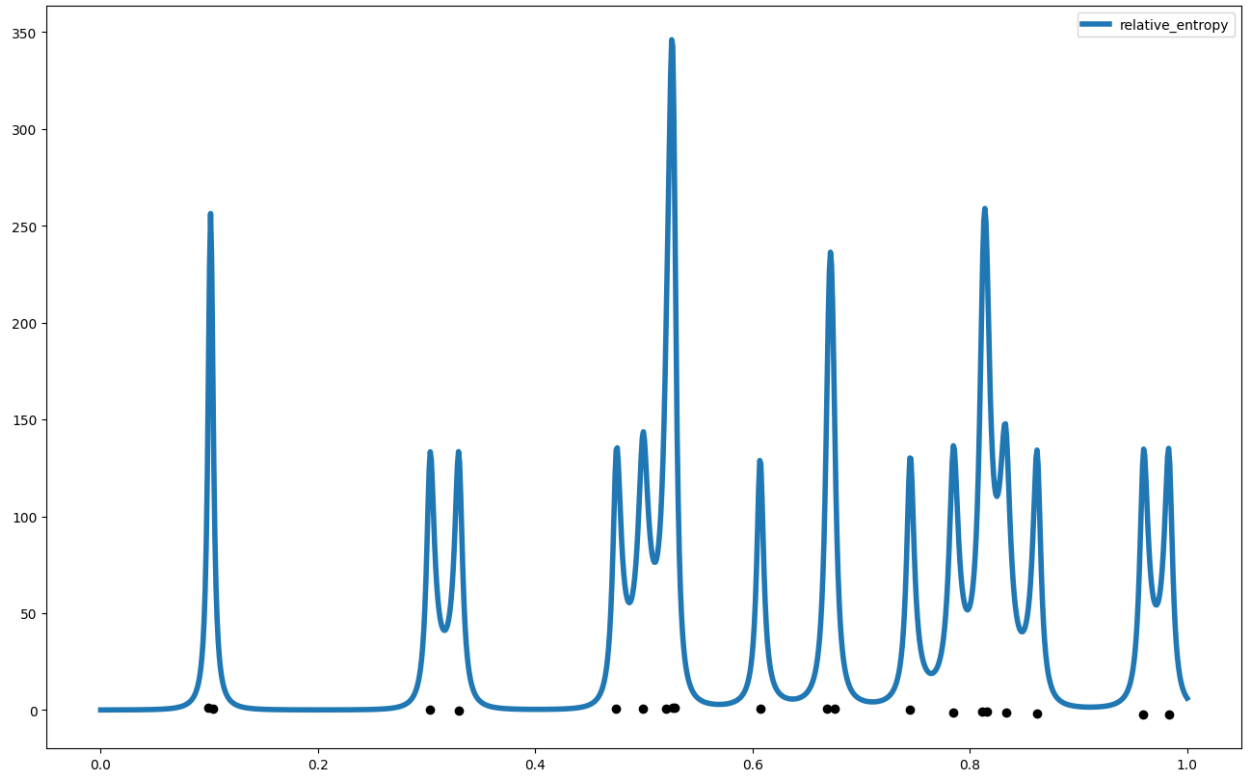


4.5.1 And just for fun, we can plot how much information we are predicted to gain if we measured points across the domain

```
relative_entropy =  my_gp1.gp_relative_information_entropy_set(x_pred.reshape(-1,1))["RIE
↪"]
```

```
plt.figure(figsize = (16,10))
plt.plot(x_pred,relative_entropy, label = "relative_entropy", linewidth = 4)
plt.scatter(x_data,y_data, color = 'black')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fc886739540>
```




```
#We can ask mutual information and total correlation there is given some test data
x_test = np.array([[0.45],[0.45]])
print("MI: ",my_gp1.gp_mutual_information(x_test))
print("TC: ",my_gp1.gp_total_correlation(x_test))
my_gp1.gp_entropy(x_test)
my_gp1.gp_entropy_grad(x_test, 0)
my_gp1.gp_kl_div(x_test, np.ones((len(x_test))), np.identity((len(x_test))))
my_gp1.gp_kl_div_grad(x_test, np.ones((len(x_test))), np.identity((len(x_test))), 0)
my_gp1.gp_relative_information_entropy(x_test)
my_gp1.gp_relative_information_entropy_set(x_test)
my_gp1.posterior_covariance(x_test)
my_gp1.posterior_covariance_grad(x_test)
my_gp1.posterior_mean(x_test)
my_gp1.posterior_mean_grad(x_test)
```

(continues on next page)

(continued from previous page)

```
my_gp1.posterior_probability(x_test, np.ones((len(x_test))), np.identity((len(x_test))))
my_gp1.posterior_probability_grad(x_test, np.ones((len(x_test))), np.identity((len(x_
↪test))),0)
```

```
MI:  {'x': array([[0.45],
       [0.45]]), 'mutual information': 1.1188091321335776}
TC:  {'x': array([[0.45],
       [0.45]]), 'total correlation': 11.273145755711699}
```

```
{'probability grad': 0.9795836351314535}
```

CHAPTER

FIVE

MULTI-TASK TEST

At first we have to install the newest version of fvGP

```
##first install the newest version of fvgp
#!pip install fvgp==4.2.0
```

5.1 Setup

```
import numpy as np
import matplotlib.pyplot as plt
from fvgp import GP
import plotly.graph_objects as go
from itertools import product

%load_ext autoreload
%autoreload 2
```

5.2 Data

```
data = np.load("./data/sim_variable_mod.npy")
sparsification = 32

x_data3 = data[:,5:][::sparsification]
y_data3 = data[:,0:3][::sparsification]

#it is good practice to check the format of the data
print(x_data3.shape)
print(y_data3.shape)
```

```
x = np.linspace(30,100,100)
y = np.linspace(40,130,100)
x_pred3D = np.asarray(list(product(x, y)))
x_pred3D = np.column_stack([x_pred3D, np.zeros((len(x_pred3D),1)) + 300.])
```

5.3 Plotting

```
def scatter(x,y,z,size=3, color = 1):
    #if not color: color = z
    fig = go.Figure()
    fig.add_trace(go.Scatter3d(x=x, y=y, z=z,mode='markers',marker=dict(color=color,␣
↪size = size)))


    fig.update_layout(autosize=False,
                      width=800, height=800,
                      font=dict(size=18,),
                      margin=dict(l=0, r=0, b=0, t=0))
    fig.show()
```

```
scatter(x_data3[:,0],x_data3[:,1],x_data3[:,2], size = 5, color = y_data3[:,0])
scatter(x_data3[:,0],x_data3[:,1],x_data3[:,2], size = 5, color = y_data3[:,1])
scatter(x_data3[:,0],x_data3[:,1],x_data3[:,2], size = 5, color = y_data3[:,2])
```

5.4 A simple kernel definition

It is vital in the multi-task case to think hard about kernel design. The kernel is now a function over $\mathcal{X} \times \mathcal{X} \times T \times T$, where $\mathcal{X}$ is the input and T is the output space. Print the input into kernel, it will have the dimensionality of this cartesian product space. The default kernel in fvgp is a deep kernel, which can be good, but there is no guarantee. To use the default kernel, pytorch has to be installed manually (pip install torch).

```
#A simple kernel, that won't lead to good performance because it's stationary
from fvgp.gp_kernels import *
def mkernel(x1,x2,hps,obj):
    d = get_distance_matrix(x1,x2)
    return hps[0] * matern_kernel_diff1(d,hps[1])
```

5.5 Initialization

```
from fvgp import fvGP
#This is where things get a little complicated. What's with all those numbers there?
#This input space is 3-dimensional, and the output space has 3 tasks but is still 1-
↪dimensional
#in the fvgp world. Therefore fvGP(3,1,3, ...), for 3 dim input, 1 dim output, with 3␣
↪outputs
#for the default (deep) kernel to work you have to install PyTorch manually

my_gp2 = fvGP(x_data3,y_data3,init_hyperparameters=np.ones((2)), info = False,
              #gp_kernel_function=mkernel#what happens if we comment this in/out?
             )
print("Global Training in progress")
#use the next two lines if kernel `mkernel` is used
```

(continues on next page)

(continued from previous page)

```
#if not a default deep kernel will be used that will set initi hyperparameters and bounds
#hps_bounds = np.array([[0.001,10000.],[1.,1000.]])
#my_gp2.train(hyperparameter_bounds = hps_bounds, max_iter = 2)

#use this next line if the default (deep) kernel is used (no bounds required)
my_gp2.train(max_iter = 2)
```

5.6 Prediction

```
#first task
mean1 = my_gp2.posterior_mean(x_pred3D, x_out = np.zeros((1)))["f(x)"]
var1 =  my_gp2.posterior_covariance(x_pred3D, x_out = np.zeros((1)))["v(x)"]

#second task
mean2 = my_gp2.posterior_mean(x_pred3D, x_out = np.zeros((1)) + 1)["f(x)"]
var2 =  my_gp2.posterior_covariance(x_pred3D, x_out = np.zeros((1))+1)["v(x)"]

#third task
mean3 = my_gp2.posterior_mean(x_pred3D, x_out = np.zeros((1)) + 2)["f(x)"]
var3 =  my_gp2.posterior_covariance(x_pred3D, x_out = np.zeros((1))+2)["v(x)"]
```

```
#extract data point to compare to:
index300 = np.where(x_data3[:,2]==300.)
imageX_data = x_data3[index300]
imageY_data = y_data3[index300]
#print(y_data3)
```

```
fig = go.Figure()
fig.add_trace(go.Scatter3d(x=x_pred3D[:,0],y=x_pred3D[:,1], z=mean1,
                           mode='markers',marker=dict(color=mean1, size = 5)))
fig.add_trace(go.Scatter3d(x=imageX_data[:,0], y=imageX_data[:,1] , z=imageY_data[:,0],
                           mode='markers',marker=dict(color=imageY_data[:,0], size = 5)))
fig.update_layout(autosize=False,
                  width=800, height=800,
                  font=dict(size=18,),
                  margin=dict(l=0, r=0, b=0, t=0))
fig.show()



fig = go.Figure()
fig.add_trace(go.Scatter3d(x=x_pred3D[:,0], y=x_pred3D[:,1], z=mean2,
                           mode='markers',marker=dict(color=mean2, size = 5)))
fig.add_trace(go.Scatter3d(x=imageX_data[:,0], y=imageX_data[:,1], z=imageY_data[:,1],
                           mode='markers',marker=dict(color=imageY_data[:,1], size = 5)))
fig.update_layout(autosize=False,
                  width=800, height=800,
                  font=dict(size=18,),
                  margin=dict(l=0, r=0, b=0, t=0))
```

(continues on next page)

(continued from previous page)

```
fig.show()

fig = go.Figure()
fig.add_trace(go.Scatter3d(x=x_pred3D[:,0], y=x_pred3D[:,1], z=mean3,
                           mode='markers',marker=dict(color=mean3, size = 5)))
fig.add_trace(go.Scatter3d(x=imageX_data[:,0], y=imageX_data[:,1], z=imageY_data[:,2],
                           mode='markers',marker=dict(color=imageY_data[:,2], size = 5)))
fig.update_layout(autosize=False,
                  width=800, height=800,
                  font=dict(size=18,),
                  margin=dict(l=0, r=0, b=0, t=0))
fig.show()
```

CHAPTER

SIX

GP2SCALE

gp2Scale is a special setting in fvgp that combines non-stationary, compactly-supported kernels, HPC distributed computing, and sparse linear algebra to allow scale-up of exact GPs to millions of data points. gp2Scale holds the world record in this category! Here we run a moderately-sized GP, just because we assume you might run this locally.

I hope it is clear how cool it is what is happening here. If you have a dask client that points to a remote cluster with 500 GPUs, you will distribute the covariance matrix computation across those. The full matrix is sparse and will be fast to work with in downstream operations. The algorithm only makes use of naturally-occuring sparsity, so the result is exact in contrast to Vecchia or inducing-point methods.

```
##first install the newest version of fvgp
#!pip install fvgp==4.2.0
```

6.1 Setup

```
import numpy as np
import matplotlib.pyplot as plt
from fvgp import GP
from dask.distributed import Client
%load_ext autoreload
%autoreload 2

client = Client() ##this is the client you can make locally like this or
#your HPC team can provide a script to get it. We included an example to get gp2Scale␣
↪going
#on Perlmutter


#It's good practice to make sure to wait for all the workers to be ready
client.wait_for_workers(4)
```

6.2 Preparing the data and some other inputs

```
def f1(x):
    return ((np.sin(5. * x) + np.cos(10. * x) + (2.* (x-0.4)**2) * np.cos(100. * x)))

input_dim = 1
N = 10000
x_data = np.random.rand(N,input_dim)
y_data = f1(x_data).reshape(len(x_data))
hps_n = 2

hps_bounds = np.array([[0.1,10.],      ##signal var of Wendland kernel
                       [0.001,0.02]])  ##length scale for Wendland kernel
```

```
init_hps = np.random.uniform(size = len(hps_bounds), low = hps_bounds[:,0], high = hps_
↪bounds[:,1])

my_gp2S = GP(x_data,y_data,init_hps,  #compute_device = 'gpu', #you can use gpus here
            gp2Scale = True, gp2Scale_batch_size= 1000, gp2Scale_dask_client = client,␣
↪info = True
            )

my_gp2S.train(hyperparameter_bounds = hps_bounds, max_iter = 2)
```

```
/home/marcus/Coding/fvGP/fvgp/gp.py:261: UserWarning: hyperparameter_bounds not provided.
↪ They will have to be provided in the training call.
  warnings.warn("hyperparameter_bounds not provided. "
```

```
KV sparsity =  0.03029374
CG solve in progress ...
CG compute time: 1.1873481273651123 seconds, exit status  0 (0:=successful)
Random logdet() in progress ...  1.1878197193145752 seconds.
Random logdet() compute time:  69.58024549484253 seconds.
KV sparsity =  0.03029374
CG solve in progress ...
CG compute time: 1.1581900119781494 seconds, exit status  0 (0:=successful)
Random logdet() in progress ...  1.1586105823516846 seconds.
Random logdet() compute time:  68.79514074325562 seconds.
KV sparsity =  0.02742228
CG solve in progress ...
CG compute time: 1.0823793411254883 seconds, exit status  0 (0:=successful)
Random logdet() in progress ...  1.0828258991241455 seconds.
Random logdet() compute time:  65.2050678730011 seconds.
KV sparsity =  0.02742228
CG solve in progress ...
CG compute time: 1.0921993255615234 seconds, exit status  0 (0:=successful)
Random logdet() in progress ...  1.0926308631896973 seconds.
Random logdet() compute time:  62.44978094100952 seconds.
```

```
x_pred = np.linspace(0,1,100) ##for big GPs, this is usually not a good idea, but in 1d,␣
```

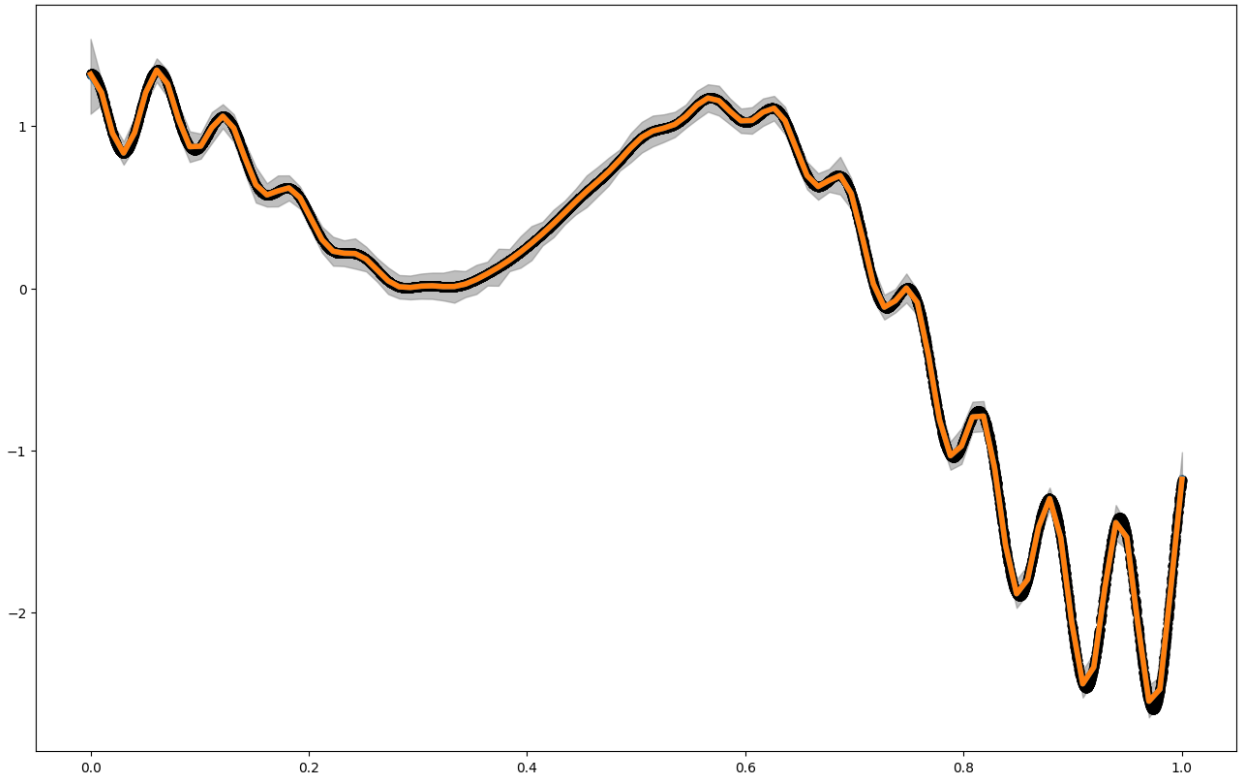
(continues on next page)

(continued from previous page)

```
↪we can still do it
                              ##It's better to do predicitons only for a handful of␣
↪points.

mean1 = my_gp2S.posterior_mean(x_pred.reshape(-1,1))["f(x)"]
var1 =  my_gp2S.posterior_covariance(x_pred.reshape(-1,1), variance_only=False)["v(x)"]

print(my_gp2S.get_hyperparameters())

plt.figure(figsize = (16,10))
plt.plot(x_pred,mean1, label = "posterior mean", linewidth = 4)
plt.plot(x_pred,f1(x_pred), label = "latent function", linewidth = 4)
plt.fill_between(x_pred, mean1 - 3. * np.sqrt(var1), mean1 + 3. * np.sqrt(var1), alpha =␣
↪0.5, color = "grey", label = "var")
plt.scatter(x_data,y_data, color = 'black')
```

```
[2.55431933 0.01375789]
```

```
<matplotlib.collections.PathCollection at 0x7feba86da9e0>
```



CHAPTER

SEVEN

GPS ON NON-EUCLIDEAN INPUT SPACES

GPs on non-Euclidean input spaces have become more and more relevant in recent years. fvgp can be used for that purpose as long as a cvalid kernel is provided. Of course, if mean functions and noise functions are also provided, they have to operate on these non-Euclidean spaces.

In this example, we run a small GP on words. It's a proof of concept, the results are not super relevant

```
#install the newest version of fvgp
#!pip install fvgp==4.2.0
```

```
import numpy as np
import matplotlib.pyplot as plt
from fvgp import GP
from dask.distributed import Client
%load_ext autoreload
%autoreload 2
```

```
#making the x_data a set will allow us to put any objects or structures into it.
x_data = [('hello'),('world'),('this'),('is'),('fvgp')]
y_data = np.array([2.,1.9,1.8,3.0,5.])


from fvgp.gp_kernels import *
def string_distance(string1, string2):
    difference = abs(len(string1) - len(string2))
    common_length = min(len(string1),len(string2))
    string1 = string1[0:common_length]
    string2 = string2[0:common_length]

    for i in range(len(string1)):
        if string1[i] != string2[i]:
            difference += 1.

    return difference


def kernel(x1,x2,hps,obj):
    d = np.zeros((len(x1),len(x2)))
    count1 = 0
    for string1 in x1:
        count2 = 0
```

(continues on next page)

(continued from previous page)

```
        for string2 in x2:
            d[count1,count2] = string_distance(string1,string2)
            count2 += 1
        count1 += 1
    return hps[0] * matern_kernel_diff1(d,hps[1])




my_gp = GP(x_data,y_data,init_hyperparameters=np.ones((2)), gp_kernel_function=kernel,␣
↪info = False)

bounds = np.array([[0.001,100.],[0.001,100]])
my_gp.train(hyperparameter_bounds=bounds)

print("hyperparameters: ", my_gp.get_hyperparameters())
print("prediction : ",my_gp.posterior_mean(['full'])["f(x)"])
print("uncertainty: ",np.sqrt(my_gp.posterior_covariance(['full'])["v(x)"]))
```

```
hyperparameters:  [1.43431191 0.40010055]
prediction :  [2.74004694]
uncertainty:  [1.19762762]
```

```
/home/marcus/Coding/fvGP/fvgp/gp.py:261: UserWarning: hyperparameter_bounds not provided.
↪ They will have to be provided in the training call.
  warnings.warn("hyperparameter_bounds not provided. "
/home/marcus/Coding/fvGP/fvgp/gp.py:301: UserWarning: No noise function or measurement␣
↪noise provided. Noise variances will be set to 1% of mean(y_data).
  self.likelihood = GPlikelihood(self.data.x_data,
```

CHAPTER

EIGHT

FVGP - A FLEXIBLE MULTI-TASK GP ENGINE

8.1 fvGP

Welcome to the documentation of the fvGP API.

fvGP is a next-generation Gaussian (and Gaussian-related) process engine for flexible, domain-informed and HPC-ready stochastic function approximation. It is the backbone of the gpCAM API. The objective of fvGP is to take care of the mathematics behind GP training and predictions but allow the user to have maximum flexibility in defining GPs. The fv in fvGP stands for function valued, an extension of multi-task GPs by the notion of an output space with it's own topology. In this framwork, the output space is assumed to have a non-constant (accoss input and output space) metric for the norm that can be learned via hyperparameter optimization. HGDL provides distributed multi-node asynchronous constrained function optimization for the training.

The fvGP package holds the world record for scaling up exact GPs!

8.2 See Also

- gpCAM
- HGDL

INDEX

U

V

X

Y