gpCAM

Marcus Michael Noack

Apr 25, 2024

API

The user has two options to deploy gpCAM: by using the AutonomousExperimenter of the gpOptimizer. Both come in single-task and multi-task versions. The AutonomousExperimenter is simpler to set up and implements a high-level loop, while the GPOptimizer is a thin optimization wrapper around the fvgp package and inherits all of its capabilities, including GPs on non-Euclidean input spaces and world-record-holding scaling of exact GPs.

To get to know gpCAM, check out the examples on this website, download the repository and go to "./tests", or visit the project website.

Quick Links:

- Repository
- *AutonomousExperimenter (GP and fvGP)*
- *gpOptimizer* and *fvgpOptimizer*
- The fvGP Package
- The HGDL Package

Have suggestions for the API or found a bug?

Please submit an issue on GitHub.

CHAPTER

ONE

AUTONOMOUSEXPERIMENTER

class gpcam.autonomous_experimenter.**AutonomousExperimenterGP**(*input_space_bounds*, *hyperparameters=None*, *hyperparameter_bounds=None*, *instrument_function=None*, *init_dataset_size=None*, *acquisition_function='variance'*, *cost_function=None*, *cost_update_function=None*, *cost_function_parameters={}*, *kernel_function=None*, *prior_mean_function=None*, *noise_function=None*, *run_every_iteration=None*, *x_data=None*, *y_data=None*, *noise_variances=None*, *dataset=None*, *communicate_full_dataset=False*, *compute_device='cpu'*, *store_inv=False*, *training_dask_client=None*, *acq_func_opt_dask_client=None*, *gp2Scale=False*, *gp2Scale_dask_client=None*, *gp2Scale_batch_size=10000*, *ram_economy=True*, *info=False*, *args=None*)

Executes the autonomous loop for a single-task Gaussian process. Use class AutonomousExperimenterFvGP for multi-task experiments. The AutonomousExperimenter is a convenience-driven functionality that does not allow as much customization as using the GPOptimizer directly. But it is great option to start with.

Parameters

- **input_space_bounds** (*np.ndarray*) – A numpy array of floats of shape D x 2 describing the input space.
- **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 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 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.
- **instrument_function** (*Callable, optional*) – A function that takes data points (a list of dicts), and returns the same with the measurement data filled in. The function is expected to communicate with the instrument and perform measurements, populating fields of the data input.
- **init_dataset_size** (*int, optional*) – If *x* and *y* are not provided and *dataset* is not provided, *init_dataset_size* must be provided. An initial dataset is constructed randomly with this length. The *instrument_function* is immediately called to measure values at these initial points.
- **acquisition_function** (*Callable, optional*) – The acquisition function accepts as input a numpy array of size V x D (such that V is the number of input points, and D is the parameter space dimensionality) and a *GPOptimizer* object. The return value is 1-D array of length V providing 'scores' for each position, such that the highest scored point will be measured next. Built-in functions can be used by one of the following keys: *ucb*,`lcb`,`maximum`, *minimum*, *variance*,`expected_improvement`, *relative information entropy*,`relative information entropy set`, *probability of improvement*, *gradient*,`total correlation`,`target probability`. If None, the default function *variance*, meaning *fvgp.GP.posterior_covariance* with variance_only = True will be used. The acquisition function can be a callable of the form my_func(x,gpcam.GPOptimizer) which will be maximized (!!!), so make sure desirable new measurement points will be located at maxima. Explanations of the acquisition functions: variance: simply the posterior variance relative information entropy: the KL divergence of the prior over predictions and the posterior relative information entropy set: the KL divergence of the prior defined over predictions and the posterior point-by-point ucb: upper confidence bound, posterior mean + 3. std lcb: lower confidence bound, -(posterior mean - 3. std) maximum: finds the maximum of the current posterior mean minimum: finds the maximum of the current posterior mean gradient: puts focus on high-gradient regions probability of improvement: as the name would suggest expected improvement: as the name would suggest total correlation: extension of mutual information to more than 2 random variables target probability: probability of a target; needs a dictionary GPOptimizer.args = {'a': lower bound, 'b': upper bound} to be defined.
- **cost_function** (*Callable, optional*) – A function encoding the cost of motion through the input space and the cost of a measurements. Its inputs are an *origin* (np.ndarray of size V x D), *x* (np.ndarray of size V x D), and the value of *cost_func_params*; *origin* is the starting position, and *x* is the destination position. The return value is a 1-D array of length V describing the costs as floats. The 'score' from acquisition_function is divided by this returned cost to determine the next measurement point. If None, the default is a uniform cost of 1.
- **cost_update_function** (*Callable, optional*) – A function that updates the *cost_func_params* which are communicated to the *cost_function*. This function accepts as input costs (a list of cost values determined by *instrument_function*), bounds (a V x 2 numpy array) and a parameters object. The default is a no-op.
- **cost_function_parameters** (*Any, optional*) – An object that is communicated to the *cost_function* and *cost_update_function*. The default is *{}*.
- **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 userdefined 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.

- **prior_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.
- **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.
- **run_every_iteration** (*Callable, optional*) – A function that is run at every iteration. It accepts as input a *gpcam.AutonomousExperimenterGP* instance. The default is a no-op.
- **x_data** (*np.ndarray, optional*) – Initial data point positions.
- **y_data** (*np.ndarray, optional*) – Initial data point values.
- **noise_variances** (*np.ndarray, optional*) – Initial data point observation variances.
- **dataset** (*string, optional*) – A filename of a gpcam-generated file that is used to initialize a new instance.
- **communicate_full_dataset** (*bool, optional*) – If True, the full dataset will be communicated to the *instrument_function* on each iteration. If False, only the newly suggested data points will be communicated. The default is False.
- **compute_device** (*str, optional*) – One of "cpu" or "gpu", determines how linear system solves are run. The default is "cpu".
- **store_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. For larger problems (>2000 data points), the use of inversion should be avoided due to computational instability and costs. The default is True. 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.
- **training_dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed training. If None is provided, a new *dask.distributed.Client* instance is constructed.
- **acq_func_opt_dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed *acquisition_function* computation. If None is provided, a new *dask.distributed.Client* instance is constructed.
- **info** (*bool, optional*) – Specifies if info should be displayed. Default = False.

x_data

Data point positions

Type
np.ndarray

y_data

Data point values

Type
np.ndarray

variances

Data point observation variances

Type
np.ndarray

data.dataset

All data

Type
list

hyperparameter_bounds

A 2d array of floats of size J x 2, such that J is the length matching the length of *hyperparameters* defining the bounds for training.

Type
np.ndarray

gp_optimizer

A GPOptimizer instance used for initializing a Gaussian process and performing optimization of the posterior.

Type
gpcam.GPOptimizer

go(*N=10000000000000000.0*, *breaking_error=1e-50*, *retrain_globally_at=(20*, *50*, *100*, *400*, *1000)*, *retrain_locally_at=(20*, *40*, *60*, *80*, *100*, *200*, *400*, *1000)*, *retrain_async_at=()*, *update_cost_func_at=()*, *acq_func_opt_setting=<function AutonomousExperimenterGP.<lambda>>*, *training_opt_max_iter=20*, *training_opt_pop_size=10*, *training_opt_tol=1e-06*, *acq_func_opt_max_iter=20*, *acq_func_opt_pop_size=20*, *acq_func_opt_tol=1e-06*, *acq_func_opt_tol_adjust=0.1*, *number_of_suggested_measurements=1*, *checkpoint_filename=None*, *constraints=()*, *break_condition_callable=<function AutonomousExperimenterGP.<lambda>>*)

Function to start the autonomous-data-acquisition loop.

Parameters

- **N** (*int, optional*) – Run until N points are measured. The default is *1e15*.
- **breaking_error** (*float, optional*) – Run until breaking_error is achieved (or at max N). The default is *1e-50*.
- **retrain_globally_at** (*Iterable [int], optional*) – Retrains the hyperparameters at the given number of measurements using global optimization. The default is *[20,50,100,400,1000]*.
- **retrain_locally_at** (*Iterable[int], optional*) – Retrains the hyperparameters at the given number of measurements using local gradient-based optimization. The default is *[20,40,60,80,100,200,400,1000]*.
- **retrain_async_at** (*Iterable[int], optional*) – Retrains the hyperparameters at the given number of measurements using the HGDL algorithm. This training is asynchronous and can be run in a distributed fashion using *training_dask_client*. The default is *[]*.

- **update_cost_func_at** (*Iterable[int], optional*) – Calls the *update_cost_function* at the given number of measurements. Default = ()
- **acq_func_opt_setting** (*Callable, optional*) – A callable that accepts as input the iteration index and returns either *local*, *global*, *hgdl*. This switches between local gradient-based, global and hybrid optimization for the acquisition function. The default is *lambda number: "global" if number % 2 == 0 else "local"*.
- **training_opt_max_iter** (*int, optional*) – The maximum number of iterations for any training. The default value is 20.
- **training_opt_pop_size** (*int, optional*) – The population size used for any training with a global component (HGDL or standard global optimizers). The default value is 10.
- **training_opt_tol** (*float, optional*) – The optimization tolerance for all training optimization. The default is 1e-6.
- **acq_func_opt_max_iter** (*int, optional*) – The maximum number of iterations for the *acquisition_function* optimization. The default is 20.
- **acq_func_opt_pop_size** (*int, optional*) – The population size used for any *acquisition_function* optimization with a global component (HGDL or standard global optimizers). The default value is 20.
- **acq_func_opt_tol** (*float, optional*) – The optimization tolerance for all *acquisition_function* optimization. The default value is 1e-6
- **acq_func_opt_tol_adjust** (*float, optional*) – The *acquisition_function* optimization tolerance is adjusted at every iteration as a fraction of this value. The default value is 0.1 .
- **number_of_suggested_measurements** (*int, optional*) – The algorithm will try to return this many suggestions for new measurements. This may be limited by how many optima the algorithm may find. If greater than 1, then the *acquisition_function* optimization method is automatically set to use HGDL. The default is 1.
- **checkpoint_filename** (*str, optional*) – When provided, a checkpoint of all the accumulated data will be written to this file on each iteration.
- **constraints** (*tuple, optional*) – If provided, this subjects the acquisition function optimization to constraints. For the definition of the constraints, follow the structure your chosen optimizer requires.
- **break_condition_callable** (*Callable, optional*) – Autonomous loop will stop when this function returns True. The function takes as input a gpcam.AutonomousExperimenterGP instance.

kill_all_clients()

Function to kill both dask.distributed.Client instances. Will be called automatically at the end of go().

kill_training()

Function to stop an asynchronous training. This leaves the dask.distributed.Client alive.

train(*init_hyperparameters=None*, *pop_size=10*, *tol=0.0001*, *max_iter=20*, *method='global'*, *constraints=()*)

This function finds the maximum of the log marginal likelihood and therefore trains the GP (synchronously). The GP prior will automatically be updated with the new hyperparameters after the training.

Parameters

- **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.
- **pop_size** (*int, optional*) – A number of individuals used for any optimizer with a global component. Default = 20.
- **tol** (*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.
- **method** (*str, optional*) – Method to be used for the training. Default is *global* which means a differential evolution algorithm is run with the specified parameters. The options are *global* or *local*, or *mcmc*.
- **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.

train_async(*init_hyperparameters=None*, *max_iter=10000*, *local_method='L-BFGS-B'*, *global_method='genetic'*, *constraints=()*)

Function to train the Gaussian Process asynchronously using the HGDL optimizer. The use is entirely optional; this function will be called as part of the go() command, if so specified. This call starts a highly parallelized optimization process, on an architecture specified by the dask.distributed.Client. The main purpose of this function is to allow for large-scale distributed training.

Parameters

- **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 iterations for the global method. Default = 10000 It is important to remember here that the call is run asynchronously, so this number does not affect run time.
- **local_method** (*str, optional*) – Local method to be used inside HGDL. Many scipy.optimize.minimize methods can be used, or a user-defined callable. Please read the HGDL docs for more information. Default = *L-BFGS-B*.
- **global_method** (*str, optional*) – Local method to be used inside HGDL. Please read the HGDL docs for more information. Default = *genetic*.
- **constraints** (*tuple of object instances, optional*) – Equality and inequality constraints for the optimization. See *hgdl.readthedocs.io* for setting up constraints.

update_hps()

Function to update the hyperparameters if an asynchronous training is running. Will be called during go() as specified.

CHAPTER

TWO

AUTONOMOUSEXPERIMENTERFVGP

class gpcam.autonomous_experimenter.**AutonomousExperimenterFvGP**(*input_space_bounds*, *output_number*, *output_dim=1*, *hyperparameters=None*, *hyperparameter_bounds=None*, *instrument_function=None*, *init_dataset_size=None*, *acquisition_function='variance'*, *cost_function=None*, *cost_update_function=None*, *cost_function_parameters=None*, *kernel_function=None*, *prior_mean_function=None*, *noise_function=None*, *run_every_iteration=None*, *x_data=None*, *y_data=None*, *noise_variances=None*, *vp=None*, *dataset=None*, *communicate_full_dataset=False*, *compute_device='cpu'*, *store_inv=False*, *training_dask_client=None*, *acq_func_opt_dask_client=None*, *gp2Scale=False*, *gp2Scale_dask_client=None*, *gp2Scale_batch_size=10000*, *ram_economy=True*, *info=False*, *args=None*)

Executes the autonomous loop for a multi-task Gaussian process.

Parameters

- **input_space_bounds** (*np.ndarray*) – A numpy array of floats of shape D x 2 describing the input space range
- **output_number** (*int*) – An integer defining how many outputs are created by each measurement.
- **output_dim** (*int*) – Integer specifying the number of dimensions of the output space. Most often 1. This is not the number of outputs/tasks. For instance, a spectrum as output at each input is itself a function over a 1d space but has many outputs.
- **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 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 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.
- **instrument_function** (*Callable, optional*) – A function that takes data points (a list of dicts), and returns the same with the measurement data filled in. The function is expected to communicate with the instrument and perform measurements, populating fields of the data input.
- **init_dataset_size** (*int, optional*) – If *x* and *y* are not provided and *dataset* is not provided, *init_dataset_size* must be provided. An initial dataset is constructed randomly with this length. The *instrument_function* is immediately called to measure values at these initial points.
- **acquisition_function** (*Callable, optional*) – The acquisition function accepts as input a numpy array of size V x D (such that V is the number of input points, and D is the parameter space dimensionality) and a *GPOptimizer* object. The return value is 1d array of length V providing 'scores' for each position, such that the highest scored point will be measured next. Built-in functions can be used by one of the following keys: *variance*, *relative information entropy*, *relative information entropy set*, *total correlation*. See GPOptimizer.ask() for a short explanation of these functions. In the multi-task case, it is highly recommended to deploy a user-defined acquisition function due to the intricate relationship of posterior distributions at different points in the output space. If None, the default function *variance*, meaning *fvgp.GP.posterior_covariance* with variance_only = True will be used. The acquisition function can be a callable of the form my_func(x,gpcam.GPOptimizer) which will be maximized (!!!), so make sure desirable new measurement points will be located at maxima.
- **cost_function** (*Callable, optional*) – A function encoding the cost of motion through the input space and the cost of a measurements. Its inputs are an *origin* (np.ndarray of size V x D), *x* (np.ndarray of size V x D), and the value of *cost_function_parameters*; *origin* is the starting position, and *x* is the destination position. The return value is a 1d array of length V describing the costs as floats. The 'score' from acquisition_function is divided by this returned cost to determine the next measurement point. If None, the default is a uniform cost of 1.
- **cost_update_function** (*Callable, optional*) – A function that updates the *cost_func_params* which are communicated to the *cost_function*. This function accepts as input costs (a list of cost values determined by *instrument_function*), bounds (a V x 2 numpy array) and a parameters object. The default is a no-op.
- **cost_function_parameters** (*Any, optional*) – An object that is communicated to the *cost_function* and *cost_update_function*. The default is *{}*.
- **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.

- **prior_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.
- **run_every_iteration** (*Callable, optional*) – A function that is run at every iteration. It accepts as input a *gpcam.AutonomousExperimenterGP* instance. The default is a no-op.
- **x_data** (*np.ndarray, optional*) – Initial data point positions.
- **y_data** (*np.ndarray, optional*) – Initial data point values.
- **noise_variances** (*np.ndarray, optional*) – Initial data point observation variances.
- **vp** (*np.ndarray, optional*) – A 3d 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]]) for each point in the input space. The default is only permissible if output_dim is 1.
- **communicate_full_dataset** (*bool, optional*) – If True, the full dataset will be communicated to the *instrument_function* on each iteration. If False, only the newly suggested data points will be communicated. The default is False.
- **compute_device** (*str, optional*) – One of "cpu" or "gpu", determines how linear system solves are run. The default is "cpu".
- **store_inv** (*bool, optional*) – If True, the algorithm calculates and stores the inverse of the covariance matrix after each training or update of the dataset, which makes computing the posterior covariance faster. For larger problems (>2000 data points), the use of inversion should be avoided due to computational instability. The default is False. Note, the training will always use a linear solve instead of the inverse for stability reasons.
- **training_dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed training. If None is provided, a new *dask.distributed.Client* instance is constructed.
- **acq_func_opt_dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed *acquisition_function* computation. If None is provided, a new *dask.distributed.Client* instance is constructed.
- **info** (*bool, optional*) – Specifies if info should be displayed. Default = False

x_data

Data point positions

Type
np.ndarray

y_data

Data point values

Type
np.ndarray

variances

Data point observation variances

Type
np.ndarray

data.**dataset**

All data

Type
list

hyperparameter_bounds

A 2d array of floats of size J x 2, such that J is the length matching the length of *hyperparameters* defining the bounds for training.

Type
np.ndarray

gp_optimizer

A GPOptimizer instance used for initializing a Gaussian process and performing optimization of the posterior.

Type
gpcam.GPOptimizer

CHAPTER

THREE

GPOPTIMIZER

class gpcam.gp_optimizer.**GPOptimizer**(*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*, *store_inv=True*, *ram_economy=False*, *args=None*, *info=False*, *cost_function=None*, *cost_function_parameters=None*, *cost_update_function=None*)

This class is an optimization wrapper around the fvgp package for single-task (scalar-valued) Gaussian Processes. Gaussian Processes can be initialized, trained, and conditioned; also the posterior can be evaluated and used via acquisition functions, and plugged into optimizers to find its maxima. This class inherits many methods from the `fvgp.GP` class.

V ... number of input points

D ... input space dimensionality

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

All posterior evaluation functions are inherited from `fvgp.GP` class. Check there for a full list of capabilities. In addition, other methods like kernel definitions and methods for validation are available. These include, but are not limited to:

`fvgp.GP.posterior_mean()`

`fvgp.GP.posterior_covariance()`

`fvgp.GP.posterior_mean_grad()`

`fvgp.GP.posterior_covariance_grad()`

`fvgp.GP.joint_gp_prior()`

`fvgp.GP.joint_gp_prior_grad()`

`fvgp.GP.gp_entropy()`

`fvgp.GP.gp_entropy_grad()`

`fvgp.GP.gp_kl_div()`

`fvgp.GP.gp_kl_div_grad()`

`fvgp.GP.gp_mutual_information()`

`fvgp.GP.gp_total_correlation()`

fvgp.GP.gp_relative_information_entropy()

fvgp.GP.gp_relative_information_entropy_set()

fvgp.GP.posterior_probability()

fvgp.GP.posterior_probability_grad()

Kernel functions:

fvgp.GP.squared_exponential_kernel()

fvgp.GP.squared_exponential_kernel_robust()

fvgp.GP.exponential_kernel()

fvgp.GP.exponential_kernel_robust()

fvgp.GP.matern_kernel_diff1()

fvgp.GP.matern_kernel_diff1_robust()

fvgp.GP.matern_kernel_diff2()

fvgp.GP.matern_kernel_diff2_robust()

fvgp.GP.sparse_kernel()

fvgp.GP.periodic_kernel()

fvgp.GP.linear_kernel()

fvgp.GP.dot_product_kernel()

fvgp.GP.polynomial_kernel()

fvgp.GP.wendland_anisotropic()

fvgp.GP.non_stat_kernel()

fvgp.GP.non_stat_kernel_gradient()

fvgp.GP.get_distance_matrix()

Other methods:

fvgp.GP.crps()

fvgp.GP.rmse()

fvgp.GP.make_2d_x_pred()

fvgp.GP.make_1d_x_pred()

fvgp.GP.log_likelihood()

fvgp.GP.neg_log_likelihood()

fvgp.GP.neg_log_likelihood_gradient()

fvgp.GP.neg_log_likelihood_hessian()

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,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 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 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 :py:class: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 3d 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** (*distributed.client.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.
- **store_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. For larger problems (>2000 data points), the use of inversion should be avoided due to computational instability and costs. The default is True. 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.
- **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

- **cost_function** (*Callable, optional*) – A function encoding the cost of motion through the input space and the cost of a measurement. Its inputs are an *origin* (np.ndarray of size V x D), *x* (np.ndarray of size V x D), and the value of *cost_func_params*; *origin* is the starting position, and *x* is the destination position. The return value is a 1d array of length V describing the costs as floats. The 'score' from acquisition_function is divided by this returned cost to determine the next measurement point. The default in no-op.
- **cost_function_parameters** (*object, optional*) – This object is transmitted to the cost function; it can be of any type. The default is None.
- **cost_update_function** (*Callable, optional*) – If provided this function will be used when *update_cost_function()* is called. The function *cost_update_function* accepts as input costs (a list of cost values usually determined by *instrument_func*) and a parameter object. The default is a no-op.

x_data

Datapoint positions

Type
np.ndarray

y_data

Datapoint values

Type
np.ndarray

noise_variances

Datapoint observation (co)variances

Type
np.ndarray

hyperparameters

Current hyperparameters in use.

Type
np.ndarray

K

Current prior covariance matrix of the GP

Type
np.ndarray

KVinv

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

Type
np.ndarray

KVlogdet

logdet(K+V)

Type
float

V

the noise covariance matrix

Type
np.ndarray

ask(*bounds=None*, *candidates=None*, *position=None*, *n=1*, *acquisition_function='variance'*, *method='global'*, *pop_size=20*, *max_iter=20*, *tol=1e-06*, *constraints=()*, *x0=None*, *vectorized=True*, *info=False*, *dask_client=None*)

Given that the acquisition device is at "position", this function `ask()`s for "n" new optimal points within certain "bounds" and using the optimization setup: "method", "acquisition_function_pop_size", "max_iter", "tol", "constraints", and "x0". This function can also choose the best candidate of a candidate set for Bayesian optimization on non-Euclidean input spaces.

Parameters

- **bounds** (*np.ndarray, optional*) – A numpy array of floats of shape D x 2 describing the search range. While this is optional, bounds or a candidate set has to be provided.
- **candidates** (*list or np.ndarray, optional*) – If provided, ask will statistically choose the best candidate from the set. This is usually desirable for non-Euclidean inputs but can be used either way. If candidates are Euclidean, they should be provided as 2d numpy array. Bounds or candidates have to be specified, not both. If N optimal solutions are requested (n=N), then a maximum of 100*N candidates are being considered randomly. If fewer candidates are provided, all will be considered.
- **position** (*np.ndarray, optional*) – Current position in the input space. If a cost function is provided this position will be taken into account to guarantee a cost-efficient new suggestion. The default is None.
- **n** (*int, optional*) – The algorithm will try to return n suggestions for new measurements. This is either done by method = 'hgdl', or otherwise by maximizing the collective information gain (default).
- **acquisition_function** (*Callable, optional*) – The acquisition function accepts as input a numpy array of size V x D (such that V is the number of input points, and D is the parameter space dimensionality) and a *GPOptimizer* object. The return value is 1d array of length V providing 'scores' for each position, such that the highest scored point will be measured next. Built-in functions can be used by one of the following keys: *ucb*,`lcb`,`maximum`, *minimum*, *variance*,`expected_improvement`, *relative information entropy*,`relative information entropy set`, *probability of improvement*, *gradient*,`total correlation`,`target probability`. If None, the default function *variance*, meaning `fvgp.GP.posterior_covariance()` with variance_only = True will be used. The acquisition function can be a callable of the form my_func(x,gpcam.GPOptimizer) which will be maximized (!!!), so make sure desirable new measurement points will be located at maxima. Explanations of the acquisition functions: variance: simply the posterior variance relative information entropy: the KL divergence of the prior over predictions and the posterior relative information entropy set: the KL divergence of the prior defined over predictions and the posterior point-by-point ucb: upper confidence bound, posterior mean + 3. std lcb: lower confidence bound, -(posterior mean - 3. std) maximum: finds the maximum of the current posterior mean minimum: finds the maximum of the current posterior mean gradient: puts focus on high-gradient regions probability of improvement: as the name would suggest expected improvement: as the name would suggest total correlation: extension of mutual information to more than 2 random variables target probability: probability of a target; needs a dictionary GPOptimizer.args = {'a': lower bound, 'b': upper bound} to be defined.

- **method** (*str, optional*) – A string defining the method used to find the maximum of the acquisition function. Choose from *global*, *local*, *hgdl*. HGDL is an in-house hybrid optimizer that is comfortable on HPC hardware. The default is *global*.
- **pop_size** (*int, optional*) – An integer defining the number of individuals if *global* is chosen as method. The default is 20. For `hgdl` this will be overwritten by the *dask_client* definition.
- **max_iter** (*int, optional*) – This number defined the number of iterations before the optimizer is terminated. The default is 20.
- **tol** (*float, optional*) – Termination criterion for the local optimizer. The default is 1e-6.
- **x0** (*np.ndarray, optional*) – A set of points as numpy array of shape N x D, used as starting location(s) for the optimization algorithms. The default is None.
- **vectorized** (*bool, optional*) – If your acquisition function is vectorized to return the solution to an array of inquiries as an array, this option makes the optimization faster if method = 'global' is used. The default is True but will be set to False if method is not global.
- **info** (*bool, optional*) – Print optimization information. The default is False.
- **constraints** (*tuple of object instances, optional*) – scipy constraints instances, depending on the used optimizer.
- **dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed *acquisition_function* optimization. If None is provided, a new `distributed.client.Client` instance is constructed for hgdl.

Returns

Solution – Found maxima of the acquisition function, the associated function values and optimization object that, only in case of *method = hgdl* can be queried for solutions.

Return type

{'x': np.array(maxima), "f(x)" : np.array(func_evals), "opt_obj" : opt_obj}

evaluate_acquisition_function(*x*, *acquisition_function='variance'*, *origin=None*)

Function to evaluate the acquisition function.

Parameters

- **x** (*np.ndarray*) – Point positions at which the acquisition function is evaluated. Shape (N x D).
- **acquisition_function** (*Callable, optional*) – Acquisition function to execute. Callable with inputs (x,gpcam.GPOptimizer), where x is a V x D array of input x position. The return value is a 1d array of length V. The default is *variance*.
- **origin** (*np.ndarray, optional*) – If a cost function is provided this 1d numpy array of length D is used as the origin of motion.

Returns

Evaluation – The acquisition function evaluations at all points x.

Return type

np.ndarray

get_data()

Function that provides a way to access the class attributes.

Returns
dictionary of class attributes

Return type
dict

kill_training(*opt_obj*)

Function to kill an asynchronous training. This shuts down the associated distributed.client.Client.

Parameters
opt_obj (*object instance*) – Object created by *train_async()*.

stop_training(*opt_obj*)

Function to stop an asynchronous training. This leaves the distributed.client.Client alive.

Parameters
opt_obj (*object instance*) – Object created by *train_async()*.

tell(*x*, *y*, *noise_variances=None*, *overwrite=True*)

This function can tell() the gp_optimizer class the data that was collected. The data will instantly be used to update the gp data. IMPORTANT: This call does not append data. The entire dataset, including the updates, has to be provided.

Parameters

- **x** (*np.ndarray*) – Point positions (of shape U x D) to be communicated to the Gaussian Process.
- **y** (*np.ndarray*) – Point values (of shape U x 1 or U) to be communicated to the Gaussian Process.
- **noise_variances** (*np.ndarray, optional*) – Point value variances (of shape U x 1 or U) to be communicated to the Gaussian Process. If not provided, the GP will 1% of the y values as variances.
- **overwrite** (*bool, optional*) – The default is True. Indicates if all previous data should be overwritten.

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 gpcam.GPOptimizer.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. The default is an array of bounds of the length of the initial hyperparameters with all bounds defined practically as [0.00001, inf]. The initial hyperparameters are either defined by the user at initialization, or in this function call, or are defined as np.ones((input_space_dim + 1)). This choice is only recommended in very basic scenarios and can lead to suboptimal results. It is better to provide hyperparameter bounds.
- **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 fvgp.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 gpcam.GPOptimizer.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.opimize.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 the [hgdl documentation](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 `distributed.client.Client` instance is constructed.

Returns

hyperparameters – Returned are the hyperparameters, however, the GP is automatically updated.

Return type

np.ndarray

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 `gpcam.GPOptimizer.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. The default is an array of bounds for the default kernel D = input_space_dim + 1 with all bounds defined practically as [0.00001, inf]. This choice is only recommended in very basic scenarios.
- **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.opimize.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.
- **dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed training if HGDL is used. If None is provided, a new `distributed.client.Client` instance is constructed.

Returns

opt_obj – Optimization object that can be given to `gpcam.GPOptimizer.update_hyperparameters()` to update the prior GP.

Return type

object instance

update_cost_function(*measurement_costs*)

This function updates the parameters for the user-defined cost function It essentially calls the user-given cost_update_function which should return the new parameters.

Parameters

measurement_costs (*object*) – An arbitrary object that describes the costs when moving in the parameter space. It can be arbitrary because the cost function using the parameters and

the cost_update_function updating the parameters are both user-defined and this object has to be in accordance with those definitions.

update_hyperparameters(*opt_obj*)

Function to update the Gaussian Process hyperparameters if an asynchronous training is running.

Parameters
opt_obj (*object instance*) – Object created by *train_async()*.

Returns
hyperparameters

Return type
np.ndarray

CHAPTER

FOUR

FVGPOPTIMIZER

class gpcam.gp_optimizer.**fvGPOptimizer**(*x_data*, *y_data*, *output_space_dimension=1*, *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*, *store_inv=True*, *ram_economy=False*, *args=None*, *info=False*, *cost_function=None*, *cost_function_parameters=None*, *cost_update_function=None*)

This class is an optimization wrapper around the fvgp package for multitask (scalar-valued) Gaussian Processes. Gaussian Processes can be initialized, trained, and conditioned; also the posterior can be evaluated and used via acquisition functions, and plugged into optimizers to find its maxima. This class inherits many methods from the `fvgp.GP` class. Check fvgp.readthedocs.io for a full list of capabilities. Please check `gpcam.GPOptimizer` for a list of capabilities.

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).
- **output_space_dimension** (*int*) – Integer specifying the number of dimensions of the output space. Most often 1. This is not the number of outputs/tasks. For instance, a spectrum as output at each input is itself a function over a 1d space but has many outputs.
- **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 3d 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]]) for each point in the input space. The default is only permissible if output_dim is 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, **:py:method:`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 3d 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** (*distributed.client.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.
- **store_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. For larger problems (>2000 data points), the use of inversion should be avoided due to computational instability and costs. The default is True. 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. If gp2Scale is used, store_inv will be set to False.
- **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
- **cost_function** (*Callable, optional*) – A function encoding the cost of motion through the input space and the cost of a measurement. Its inputs are an *origin* (np.ndarray of size V x D), *x* (np.ndarray of size V x D), and the value of *cost_func_params*; *origin* is the starting position, and *x* is the destination position. The return value is a 1d array of length V describing the costs as floats. The 'score' from acquisition_function is divided by this returned cost to determine the next measurement point. The default in no-op.
- **cost_function_parameters** (*object, optional*) – This object is transmitted to the cost function; it can be of any type. The default is None.
- **cost_update_function** (*Callable, optional*) – If provided this function will be used when *update_cost_function()* is called. The function *cost_update_function* accepts as input costs (a list of cost values usually determined by *instrument_func*) and a parameter object. The default is a no-op.

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

hyperparameters

Current hyperparameters in use.

Type
np.ndarray

K

Current prior covariance matrix of the GP

Type
np.ndarray

KVinv

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

Type
np.ndarray

KVlogdet

logdet(K+V)

Type
float

ask(*bounds*, *x_out*, *acquisition_function='variance'*, *position=None*, *n=1*, *method='global'*, *pop_size=20*, *max_iter=20*, *tol=1e-06*, *constraints=()*, *x0=None*, *vectorized=True*, *candidates=None*, *info=False*, *dask_client=None*)

Given that the acquisition device is at *position*, the function ask()s for "n" new optimal points within certain "bounds" and using the optimization setup: "acquisition_function_pop_size", *max_iter* and *tol*.

Parameters

- **bounds** (*np.ndarray*) – A numpy array of floats of shape D x 2 describing the search range.
- **candidates** (*list, optional*) – Not implemented yet for multitask fvgp.
- **x_out** (*np.ndarray*) – The position indicating where in the output space the acquisition function should be evaluated. This array is of shape (No, Do).
- **position** (*np.ndarray, optional*) – Current position in the input space. If a cost function is provided this position will be taken into account to guarantee a cost-efficient new suggestion. The default is None.
- **n** (*int, optional*) – The algorithm will try to return n suggestions for new measurements. This is either done by method = 'hgdl', or otherwise by maximizing the collective information gain (default).
- **acquisition_function** (*Callable, optional*) – The acquisition function accepts as input a numpy array of size V x D (such that V is the number of input points, and D is the parameter space dimensionality) and a *GPOptimizer* object. The return value is 1d array of length V providing 'scores' for each position, such that the highest scored point will be measured next. Built-in functions can be used by one of the following keys: *variance*, *relative information entropy*, *relative information entropy set*, *total correlation*, *ucb*, and *expected improvement*. See *GPOptimizer.ask()* for a short explanation of these functions. In the multitask case, it is highly recommended to deploy a user-defined acquisition function due to the intricate relationship of posterior distributions at different points in the output space. If None, the default function *variance*, meaning `fvgp.GP.posterior_covariance()` with variance_only = True will be used. The acquisition function can be a callable of the form my_func(x,gpcam.GPOptimizer) which will be maximized (!!!), so make sure desirable new measurement points will be located at maxima.
- **method** (*str, optional*) – A string defining the method used to find the maximum of the acquisition function. Choose from *global*, *local*, *hgdl*. The default is *global*.

- **pop_size** (*int, optional*) – An integer defining the number of individuals if *global* is chosen as method. The default is 20. For *hgdl* this will be overwritten by the *dask_client* definition.
- **max_iter** (*int, optional*) – This number defined the number of iterations before the optimizer is terminated. The default is 20.
- **tol** (*float, optional*) – Termination criterion for the local optimizer. The default is 1e-6.
- **x0** (*np.ndarray, optional*) – A set of points as numpy array of shape V x D, used as starting location(s) for the local and hgdl optimization algorithm. The default is None.
- **vectorized** (*bool, optional*) – If your acquisition function vectorized to return the solution to an array of inquiries as an array, this option makes the optimization faster if method = 'global' is used. The default is True but will be set to False if method is not global.
- **info** (*bool, optional*) – Print optimization information. The default is False.
- **constraints** (*tuple of object instances, optional*) – Either a tuple of hgdl.constraints.NonLinearConstraint or scipy constraints instances, depending on the used optimizer.
- **dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed *acquisition_func* computation. If None is provided, a new distributed.client.Client instance is constructed.

Returns

dictionary – Found maxima of the acquisition function, the associated function values and optimization object that, only in case of *method = hgdl* can be queried for solutions.

Return type

{'x': np.array(maxima), 'f(x)' : np.array(func_evals), 'opt_obj' : opt_obj}

evaluate_acquisition_function(*x*, *x_out*, *acquisition_function='variance'*, *origin=None*)

Function to evaluate the acquisition function.

Parameters

- **x** (*np.ndarray*) – Point positions at which the acquisition function is evaluated. This is a point in the input space.
- **x_out** (*np.ndarray*) – Point positions in the output space.
- **acquisition_function** (*Callable, optional*) – Acquisition function to execute. Callable with inputs (x,gpcam.gp_optimizer.GPOptimizer), where x is a V x D array of input x_data. The return value is a 1d array of length V. The default is *variance*.
- **origin** (*np.ndarray, optional*) – If a cost function is provided this 1d numpy array of length D is used as the origin of motion.

Returns

The acquisition function evaluations at all points x

Return type

np.ndarray

get_data()

Function that provides a way to access the class attributes.

Returns

dictionary of class attributes

Return type
dict

kill_training(*opt_obj*)

Function to kill an asynchronous training. This shuts down the associated distributed.client.Client.

Parameters
opt_obj (*object instance*) – Object created by *train_async()*.

stop_training(*opt_obj*)

Function to stop an asynchronous training. This leaves the distributed.client.Client alive.

Parameters
opt_obj (*object instance*) – Object created by *train_async()*.

tell(*x*, *y*, *noise_variances=None*, *output_positions=None*, *overwrite=True*)

This function can tell() the gp_optimizer class the data that was collected. The data will instantly be used to update the GP data.

Parameters

- **x** (*np.ndarray*) – Point positions (of shape U x D) to be communicated to the Gaussian Process.
- **y** (*np.ndarray*) – Point values (of shape U x 1 or U) to be communicated to the Gaussian Process.
- **noise_variances** (*np.ndarray, optional*) – Point value variances (of shape U x 1 or U) to be communicated to the Gaussian Process. If not provided, the GP will 1% of the y values as variances.
- **output_positions** (*np.ndarray, optional*) – A 3d 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]]) for each point in the input space. The default is only permissible if output_dim is 1.
- **overwrite** (*bool, optional*) – The default is True. Indicates if all previous data should be overwritten.

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 *py:meth:`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. The default is an array of bounds of the length of the initial hyperparameters with all bounds defined practically as [0.00001, inf]. The initial hyperparameters are either defined by the user at initialization, or in this function call, or are defined as np.ones((input_space_dim + 1)). This choice is only recommended in very basic scenarios and can lead to suboptimal results. It is better to provide hyperparameter bounds.
- **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 *fvgp.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 gpcam.GPOptimizer.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.opimize.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. 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 distributed.client.Client instance is constructed.

Returns

hyperparameters – Returned are the hyperparameters, however, the GP is automatically updated.

Return type

np.ndarray

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 `gpcam.GPOptimizer.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. The default is an array of bounds for the default kernel D = input_space_dim + 1 with all bounds defined practically as [0.00001, inf]. This choice is only recommended in very basic scenarios.
- **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
- **dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed training if HGDL is used. If None is provided, a new `distributed.client.Client` instance is constructed.

Returns

opt_obj – Optimization object that can be given to `gpcam.GPOptimizer.update_hyperparameters()` to update the prior GP

Return type

object instance

update_cost_function(*measurement_costs*)

This function updates the parameters for the user-defined cost function. It essentially calls the user-given cost_update_function which should return the new parameters.

Parameters

measurement_costs (*object*) – An arbitrary object that describes the costs when moving in the parameter space. It can be arbitrary because the cost function using the parameters and

the cost_update_function updating the parameters are both user-defined and this object has to be in accordance with those definitions.

update_hyperparameters(*opt_obj*)

Function to update the Gaussian Process hyperparameters if an asynchronous training is running.

Parameters
opt_obj (*object instance*) – Object created by *train_async()*.

Returns
hyperparameters – Hyperparameter are returned but are also automatically used to update the GP.

Return type
np.ndarray

CHAPTER

FIVE

AUTONOMOUSEXPERIMENTER BASIC

```
#!pip install gpcam==8.0.3
```

```
from gpcam import AutonomousExperimenterGP
import numpy as np

def instrument(data):
    for entry in data:
        print("I want to know the y_data at: ", entry["x_data"])
        entry["y_data"] = np.sin(np.linalg.norm(entry["x_data"]))
        print("I received ",entry["y_data"])
        print("")
    return data

##set up your parameter space
parameters = np.array([[3.0,45.8],
                       [4.0,47.0]])

##set up some hyperparameters, if you have no idea, set them to 1 and make the training␣
↪bounds large
init_hyperparameters = np.array([1,1,1])
hyperparameter_bounds =  np.array([[0.01,100],[0.01,100.0],[0.01,100]])

##let's initialize the autonomous experimenter ...
my_ae = AutonomousExperimenterGP(parameters, init_hyperparameters,
                                 hyperparameter_bounds,instrument_function = instrument,
                                 init_dataset_size=10, info=False)
#...train...
my_ae.train()

#...and run. That's it. You successfully executed an autonomous experiment.
my_ae.go(N = 100)
```

CHAPTER

SIX

GPCAM ADVANCED APPLICATION

In this notebook, we will go through many features of gpCAM. Work through it and you are ready for your own autonomous experiment. This notebook uses gpCAM version 8.0.3!

```
####install gpcam here if you do not have already done so
#!pip install gpcam==8.0.3
```

6.1 Preparations

```
%load_ext autoreload
%autoreload 2
```

```
import plotly.graph_objects as go
import numpy as np
def plot(x,y,z,data = None):
    fig = go.Figure()
    fig.add_trace(go.Surface(x = x, y = y,z=z))
    if data is not None:
        fig.add_trace(go.Scatter3d(x=data[:,0], y=data[:,1], z=data[:,2],
                                   mode='markers'))

    fig.update_layout(title='Posterior Mean', autosize=True,
                  width=800, height=800,
                  margin=dict(l=65, r=50, b=65, t=90))


    fig.show()
```

6.2 Defining Prediction Points

```
x_pred = np.zeros((10000,2))
x = np.linspace(0,10,100)
y = np.linspace(0,10,100)
x,y = np.meshgrid(x,y)
counter = 0
for i in  range(100):
```

(continues on next page)

(continued from previous page)

```
    for j in range(100):
        x_pred[counter] = np.array([x[i,j],y[i,j]])
        counter += 1
```

6.3 Definition of Optional Customization Functions

```
def optional_acq_func(x,obj):
    #this acquisition function makes the autonomous experiment a Bayesian optimization
    #but is just here as an example. 'acq_funciton="ucb"' will give you the same result
    a = 3.0 #3.0 for 95 percent confidence interval
    mean = obj.posterior_mean(x)["f(x)"]
    cov = obj.posterior_covariance(x)["v(x)"]
    return mean + a * np.sqrt(cov)

def optional_mean_func(x,hyperparameters,gp_obj):
    #the prior mean function should return a vector: a mean function evaluation for␣
↪every x
    return np.zeros((len(x)))

def optional_cost_function(origin,x,arguments = None):
    #cost pf l1 motion in the input space
    offset = arguments["offset"]
    slope = arguments["slope"]
    d = np.abs(np.subtract(origin,x))
    c = (d * slope) + offset
    n = np.sum(c)
    return n

def optional_cost_update_function(costs, parameters):
    ###defining a cost update function might look tricky but just needs a bit
    ###of tenacity. And remember, this is optional, if you have a great guess for your␣
↪costs you
    ###don't need to update the costs. Also, if you don't account for costs, this␣
↪function is not needed.
    #In this example we just return the old parameters, but print the costs.
    #I hope it is clear how the parameters can be fitted to the recorded costs.
    print("recorded costs (from,to,costs): ", costs)

    return parameters
```

6.4 AutonomousExperimenter Initialization

```
import time
from gpcam import AutonomousExperimenterGP

def instrument(data):
    print("Suggested by gpCAM: ")
    for entry in data:
        print("suggested:", entry["x_data"])
        entry["y_data"] = np.sin(np.linalg.norm(entry["x_data"]))
        entry["cost"]  = [np.array([0,0]),entry["x_data"],np.sum(entry["x_data"])]
        print("received: ", entry["y_data"])
    print("")
    return data

#initialization
#feel free to try different acquisition functions, e.g. optional_acq_func, "covariance",
↪"shannon_ig"
#note how costs are defined in for the autonomous experimenter
my_ae = AutonomousExperimenterGP(np.array([[0,10],[0,10]]),
                                 np.ones((3)), np.array([[0.001,100.],[0.001,100.],[0.
↪001,100.]]),
                                 init_dataset_size= 20, instrument_function = instrument,
                                 acquisition_function = optional_acq_func,
                                 cost_function = optional_cost_function,
                                 cost_update_function = optional_cost_update_function,
                                 cost_function_parameters={"offset": 5.0,"slope":10.0},
                                 kernel_function = None, store_inv = True,
                                 prior_mean_function = optional_mean_func,
                                 communicate_full_dataset = False, ram_economy = True)#,␣
↪info = False, prior_mean_func = optional_mean_func)


print("length of the dataset: ",len(my_ae.x_data))


#my_ae.train_async()                 #train asynchronously
my_ae.train(method = "global")       #or not, or both, choose between "global","local"␣
↪and "hgdl"
```

```
#update hyperparameters in case they are optimized asynchronously
my_ae.update_hps()
print(my_ae.gp_optimizer.hyperparameters)
```

```
#training and client can be killed if desired and in case they are optimized␣
↪asynchronously
my_ae.kill_training()
```

6.5 Initial Model Vizualization

```
f = my_ae.gp_optimizer.posterior_mean(x_pred)["f(x)"]
f_re = f.reshape(100,100)

plot(x,y,f_re, data = np.column_stack([my_ae.x_data,my_ae.y_data]))
```

6.6 The go() Command

```
#run the autonomous loop
my_ae.go(N = 100,
            retrain_async_at=[30,40,50,60,70,80,90],
            retrain_globally_at = [20,22,24,26,28,30,40,50,60,70],
            retrain_locally_at = [21,22,56,78],
            acq_func_opt_setting = lambda obj: "global" if len(obj.data.dataset) % 2 ==␣
↪0 else "local",
            update_cost_func_at = (50,),
            training_opt_max_iter = 20,
            training_opt_pop_size = 10,
            training_opt_tol      = 1e-6,
            acq_func_opt_max_iter = 20,
            acq_func_opt_pop_size = 20,
            acq_func_opt_tol      = 1e-6,
            number_of_suggested_measurements = 1,
            acq_func_opt_tol_adjust = 0.1)
```

6.7 Visualization of the Resulting Model

```
res = my_ae.gp_optimizer.posterior_mean(x_pred)
f = res["f(x)"]
f = f.reshape(100,100)

plot(x,y,f, data = np.column_stack([my_ae.x_data,my_ae.y_data]))
```

CHAPTER

SEVEN

RUNNING A MULTI-TASK GP AUTONOMOUS DATA ACQUISITION

This example uses 21 (!) dim robot data and 7 tasks, which you can all use or pick a subset of them

```
##prepare some data
import numpy as np
from scipy.interpolate import griddata
data = np.load("./data/sarcos.npy")
print(data.shape)
x = data[:,0:21]
y = data[:,21:23]
```

```
from gpcam import AutonomousExperimenterFvGP

def instrument(data, instrument_dict = {}):
    for entry in data:
        print("Suggested by gpCAM: ", entry["x_data"])
        entry["y_data"] = griddata(x,y,entry["x_data"],method = "nearest", fill_value =␣
↪0)[0]
        entry["output positions"] = np.array([[0],[1]])
        print("received: ", entry["y_data"])
    print("")
    return data

def acq_func(x,obj):
    #multi-tast autonomous experiments should make use of a user-defined acquisition␣
↪function to
    #make full use of the surrogate and the uncertainty in all tasks.
    a = 3.0 #3.0 for ~95 percent confidence interval
    x = np.block([[x,np.zeros((len(x))).reshape(-1,1)],[x,np.ones((len(x))).reshape(-1,
↪1)]]) #for task 0 and 1
    mean = obj.posterior_mean(x)["f(x)"]
    cov = obj.posterior_covariance(x)["v(x)"]
    #it takes a little bit of wiggling to get the tasks seperated and then merged again..
↪.
    task0index = np.where(x[:,21] == 0.)[0]
    task1index = np.where(x[:,21] == 1.)[0]
    mean_task0 = mean[task0index]
    mean_task1 = mean[task1index]
    cov_task0 = cov[task0index]
    cov_task1 = cov[task1index]
    mean = np.column_stack([mean_task0,mean_task1])
```

(continues on next page)

(continued from previous page)

```
    cov  = np.column_stack([cov_task0 ,cov_task1 ])
    #and now we are interested in the l2 norm of the mean and variance at each input␣
↪location.
    return np.linalg.norm(mean, axis = 1) + a * np.linalg.norm(cov,axis = 1)


input_s = np.array([np.array([np.min(x[:,i]),np.max(x[:,i])]) for i in range(len(x[0]))])
print("index set (input space) bounds:")
print(input_s)
print("hps bounds:")
hps_bounds = np.empty((23,2))
hps_bounds[:,0] = 0.0001
hps_bounds[:,1] = 100.0
hps_bounds[0] = np.array([0.0001, 10000])
print(hps_bounds)
print("shape of y: ")
print(y.shape)

my_fvae = AutonomousExperimenterFvGP(input_s,2,1,
                                     init_dataset_size= 10, instrument_function =␣
↪instrument, \
                                     acquisition_function=acq_func)


my_fvae.train()
my_fvae.go(N = 50, retrain_async_at=(22,), retrain_globally_at=(50,90,120), retrain_
↪locally_at=(25,))
```

7.1 Plotting the 0th task in a 2d slice

```
x_pred = np.zeros((10000,22))
x = np.linspace(input_s[0,0],input_s[0,1],100)
y = np.linspace(input_s[1,0],input_s[1,1],100)
x,y = np.meshgrid(x,y)
counter = 0
for i in  range(100):
    for j in range(100):
        x_pred[counter] = np.zeros((22))
        x_pred[counter,[0,1,-1]] = np.array([x[i,j],y[i,j],0.])
        counter += 1
res = my_fvae.gp_optimizer.posterior_mean(x_pred)
f = res["f(x)"]
f = f.reshape(100,100)

print(f)

plot(x,y,f)
```

CHAPTER

EIGHT

GPOPTIMIZER: SINGLE-TASK ACQUISITION FUNCTIONS

```
#!pip install gpcam==8.0.3
```

8.1 Setup

```
import numpy as np
import matplotlib.pyplot as plt
from gpcam import GPOptimizer
import time
from loguru import logger

#do this for printouts, otherwise disable
#logger.enable("gpcam")
#logger.enable("fvgp")
#logger.enable("hgdl")

%load_ext autoreload
%autoreload 2
```

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

8.2 Data Preparation

```
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(50).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)
```

(continues on next page)

(continued from previous page)

```
plt.plot(x_pred1D,f1(x_pred1D), color = 'orange', linewidth = 4)
plt.scatter(x_data[:,0],y_data, color = 'black')
```

8.3 Customizing the Gaussian Process

```
def my_noise(x,hps,obj):
    #This is a simple noise function but can be made 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 = obj.get_distance_matrix(x1,x2)
    return hps[0] * obj.matern_kernel_diff1(d,hps[1])


def meanf(x, hps, obj):
    #This is 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')
```

8.4 Initialization and Different Training Options

```
my_gpo = GPOptimizer(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, #providing noise variances and␣
↪a noise function will raise a warning
            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,
            store_inv=False,
            ram_economy=False,
            args=None,
            )
```

(continues on next page)

(continued from previous page)

```
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
                       [0.001,1.]  #mean
                      ])
my_gpo.tell(x_data, y_data, noise_variances=np.ones(y_data.shape) * 0.01,␣
↪overwrite=False)
my_gpo.tell(x_data, y_data, noise_variances=np.ones(y_data.shape) * 0.01, overwrite=True)
my_gpo.tell(x_data, y_data, noise_variances=np.ones(y_data.shape) * 0.01)
print("Standard Training")
my_gpo.train(hyperparameter_bounds=hps_bounds)
print("Global Training")
my_gpo.train(hyperparameter_bounds=hps_bounds, method='global')
print("hps: ", my_gpo.get_hyperparameters())
print("Local Training")
my_gpo.train(hyperparameter_bounds=hps_bounds, method='local')
print(my_gpo.get_hyperparameters())
print("MCMC Training")
my_gpo.train(hyperparameter_bounds=hps_bounds, method='mcmc', max_iter=1000)
print("HGDL Training")
my_gpo.train(hyperparameter_bounds=hps_bounds, method='hgdl', max_iter=10)
```

8.5 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_gpo.train_async(hyperparameter_bounds=hps_bounds)
for i in range(10):
    my_gpo.update_hyperparameters(opt_obj)
    time.sleep(2)
    print(my_gpo.hyperparameters)
    print("")

my_gpo.stop_training(opt_obj)
```

8.6 Calculating on Vizualizing the Results

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

mean1 = my_gpo.posterior_mean(x_pred.reshape(-1,1))["f(x)"]
var1 =  my_gpo.posterior_covariance(x_pred.reshape(-1,1), variance_only=False, add_
↪noise=True)["v(x)"]
plt.figure(figsize = (16,10))
plt.plot(x_pred,mean1, label = "posterior mean", linewidth = 4)
plt.plot(x_pred1D,f1(x_pred1D), label = "latent function", linewidth = 4)
```

(continues on next page)

(continued from previous page)

```
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_gpo.rmse(x_pred1D,f1(x_pred1D)))
print(my_gpo.crps(x_pred1D,f1(x_pred1D)))
```

```
#available acquisition function:
acquisition_functions = ["variance","relative information entropy","relative information␣
↪entropy set",
                         "ucb","lcb","maximum","minimum","gradient","expected improvement
↪",
                          "probability of improvement", "target probability", "total␣
↪correlation"]
```

```
plt.figure(figsize=(16,10))
for acq_func in acquisition_functions:
    print("Acquisition function ",acq_func)
    my_gpo.args = args={'a': 1.5, 'b':2.}
    res = my_gpo.evaluate_acquisition_function(x_pred, acquisition_function=acq_func)
    print(" creates an output of shape",res.shape)
    if len(res)==len(x_pred):
        res = res - np.min(res)
        res = res/np.max(res)
        plt.plot(x_pred,res, label = acq_func, linewidth = 2)
    else: print("Some acquisition function return a scalar score for the entirety of␣
↪points. Here: ", acq_func)
plt.legend()
plt.show()
```

8.7 ask()ing for Optimal Evaluations

with several optimization methods and acquisition functions

```
#let's test the asks:
bounds = np.array([[0.0,1.0]])
for acq_func in acquisition_functions:
    for method in ["global","local","hgdl"]:
        print("Acquisition function ", acq_func," and method ",method)
        new_suggestion = my_gpo.ask(bounds, acquisition_function=acq_func,
                                    method=method, max_iter = 2,)
        print("led to new suggestion: \n", new_suggestion)
        print("")
```

```
#here we can test other options of the ask() command
bounds = np.array([[0.0,1.0]])
new_suggestion = my_gpo.ask(bounds, acquisition_function="total_correlation", method=
```

(continues on next page)

(continued from previous page)

```
↪"global",
                              max_iter=10, n = 5, info = True)
my_gpo.ask(bounds, n = 5, acquisition_function="variance", vectorized=True, method =
↪'global')
my_gpo.ask(bounds, n = 1, acquisition_function="relative information entropy",␣
↪vectorized=True, method = 'global')
my_gpo.ask(bounds, n = 2, acquisition_function="expected improvement", vectorized=True,␣
↪method = 'global')
my_gpo.ask(bounds, n = 1, acquisition_function="variance", vectorized=True, method =
↪'global')
my_gpo.ask(bounds, n = 3, acquisition_function="variance", vectorized=True, method =
↪'hgdl')
print(new_suggestion)
```

```
#We can also ask for the best subset of a candidate set
```

```
my_gpo.ask(candidates = np.array([[1.],[2.]]), n = 3, acquisition_function="variance",␣
↪vectorized=True, method = 'hgdl')
```

```
bounds = np.array([[0.0,1.0]])

#You can even start an ask() search asynchronously and check back later what was found
new_suggestion = my_gpo.ask(bounds, acquisition_function=acquisition_functions[0],␣
↪method="hgdlAsync")
time.sleep(10)
print(new_suggestion["opt_obj"])
#And we have to cancel that trainig and possibly kill the client
new_suggestion["opt_obj"].kill_client()
```

CHAPTER

NINE

GPOPTIMIZER: SINGLE-TASK

This is the new test for gpCAM version 8.0.4 and later.

```
##first install the newest version of fvgp
#!pip install gpcam==8.0.3
```

9.1 Setup

```
import numpy as np
import matplotlib.pyplot as plt
from gpcam import GPOptimizer
import time


%load_ext autoreload
%autoreload 2
```

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

9.2 Data Prep

```
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')
```

9.3 Customizing the Gaussian Process

```
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 = obj.get_distance_matrix(x1,x2)
    return hps[0] * obj.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')
```

9.4 Initialization and Different Training Options

```
my_gp1 = GPOptimizer(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, # providing noise variances␣
↪and a noise function will raise a warning
            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,
            store_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
                      ])
```

(continues on next page)

(continued from previous page)

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

9.5 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)
for i in range(10):
    time.sleep(1)
    my_gp1.update_hyperparameters(opt_obj)
    print(my_gp1.hyperparameters)
    print("")
my_gp1.stop_training(opt_obj)
```

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

9.6 Plotting the Result

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

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))
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')
```

(continues on next page)

(continued from previous page)

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

```
#We can ask mutual information and total correlation there is given some test data
x_test = np.array([[0.45],[0.45]])
print("Mutual Information: ",my_gp1.gp_mutual_information(x_test))
print("Total Correlation : ",my_gp1.gp_total_correlation(x_test))
```

```
next_point = my_gp1.ask(bounds=np.array([[0.,1.]]))
print(next_point)
```

CHAPTER

TEN

GPS ON NON-EUCLIDEAN INPUT SPACES

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

In this example, we run a small GP on words.

```
#install the newest version of gpcam
#!pip install gpcam==8.0.3
```

```
import numpy as np
import matplotlib.pyplot as plt
from gpcam import GPOptimizer
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'),('gpcam')]
y_data = np.array([2.,1.9,1.8,3.0,5.])


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
        for string2 in x2:
```

(continues on next page)

(continued from previous page)

```
            d[count1,count2] = string_distance(string1,string2)
            count2 += 1
        count1 += 1
    return hps[0] * obj.matern_kernel_diff1(d,hps[1])



my_gp = GPOptimizer(x_data,y_data, init_hyperparameters=np.ones((2)),
                    gp_kernel_function=kernel, info = True)
bounds = np.array([[0.001,100.],[0.001,100]])
my_gp.train(hyperparameter_bounds=bounds)

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

```
##which one should I measure next?
print(my_gp.input_space_dim)
my_gp.ask(candidates=[('hello'),('world'),("it"),("is"),("me")], n = 4)
```

CHAPTER

ELEVEN

GP2SCALE VIA THE GPOPTIMIZER

gp2Scale is a special setting in gpCAM that combines non-stationary, compactly-supported kernels, HPC distributed computing, and sparse random linear algebra to allow scale-up of exact GPs to millions of data points. 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 gpcam==8.0.3
```

11.1 Setup

```
import numpy as np
import matplotlib.pyplot as plt
from gpcam import GPOptimizer
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 NERSC's Perlmutter


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

11.2 Preparing the Data

```
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)
```

11.3 Setting up the GPOptimizer with gp2Scale

```
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 = GPOptimizer(x_data,y_data,init_hyperparameters=init_hps,
            gp2Scale = True, gp2Scale_batch_size= 1000, gp2Scale_dask_client = client,␣
↪info = True
            )

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

11.4 Vizualizing the Result

```
x_pred = np.linspace(0,1,100) ##for big GPs, this is usually not a good idea, but in 1d,␣
↪we can still do it
                              ##It's better to do predictions 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.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')
```

```
my_gp2S.ask(np.array([[0,1]]))
```

CHAPTER

TWELVE

GPOPTIMIZER MULTI-TASK TEST

At first we have to install the newest version of gpCAM

```
##first install the newest version of fvgp
#!pip install gpCAM==8.0.3
```

12.1 Setup

```
import numpy as np
import matplotlib.pyplot as plt
from gpcam import fvGPOptimizer
import plotly.graph_objects as go
from itertools import product

%load_ext autoreload
%autoreload 2
```

12.2 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.])
```

12.3 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])
```

12.4 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).

```
#As imple kernel, that won't lead to good performance because its stationary
def mkernel(x1,x2,hps,obj):
    d = obj.get_distance_matrix(x1,x2)
    return hps[0] * obj.matern_kernel_diff1(d,hps[1])
```

12.5 Initialization

```
#This is where things get a littlecomplicated. What's with all those numbers there.
#This input space is 3-dimensional, 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

my_gp2 = fvGPOptimizer(x_data3,y_data3, 1,init_hyperparameters=np.ones((2)), info = True,
            #gp_kernel_function=mkernel #what happens if comment this out? (adjust␣
↪bounds below)
            )
#change this based on kernel choice
hps_bounds = my_gp2.hps_bounds
#hps_bounds = np.array([[0.001,10000.],[1.,1000.]])
```

(continues on next page)

(continued from previous page)

```
#my_gp2.update_gp_data(x_data3,y_data3)
print("Global Training in progress")
#my_gp2.train(hyperparameter_bounds=hps_bounds, max_iter = 2)
```

12.6 Prediction

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

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

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

fig = go.Figure()
```

(continues on next page)

(continued from previous page)

```
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()
```

```
my_gp2.ask(bounds=np.array([[0,1],[0,1],[0,1]]), n = 1, acquisition_function = 'relative␣
↪information entropy set', x_out = np.array([[0],[1],[2]]), vectorized = True)
my_gp2.ask(bounds=np.array([[0,1],[0,1],[0,1]]), n = 1, acquisition_function = 'relative␣
↪information entropy', x_out = np.array([[0],[1],[2]]), vectorized = True)
my_gp2.ask(bounds=np.array([[0,1],[0,1],[0,1]]), n = 1, acquisition_function = 'variance
↪', x_out = np.array([[0],[1],[2]]), vectorized = True)
my_gp2.ask(bounds=np.array([[0,1],[0,1],[0,1]]), n = 1, acquisition_function = 'total␣
↪correlation', x_out = np.array([[0],[1],[2]]), vectorized = True)


my_gp2.ask(bounds=np.array([[0,1],[0,1],[0,1]]), n = 4, acquisition_function = 'relative␣
↪information entropy set', x_out = np.array([[0],[1],[2]]), vectorized = True)
my_gp2.ask(bounds=np.array([[0,1],[0,1],[0,1]]), n = 5, acquisition_function = 'relative␣
↪information entropy', x_out = np.array([[0],[1],[2]]), vectorized = True)
my_gp2.ask(bounds=np.array([[0,1],[0,1],[0,1]]), n = 6, acquisition_function = 'variance
↪', x_out = np.array([[0],[1],[2]]), vectorized = True)
my_gp2.ask(bounds=np.array([[0,1],[0,1],[0,1]]), n = 2, acquisition_function = 'total␣
↪correlation', x_out = np.array([[0],[1],[2]]), vectorized = True)
```

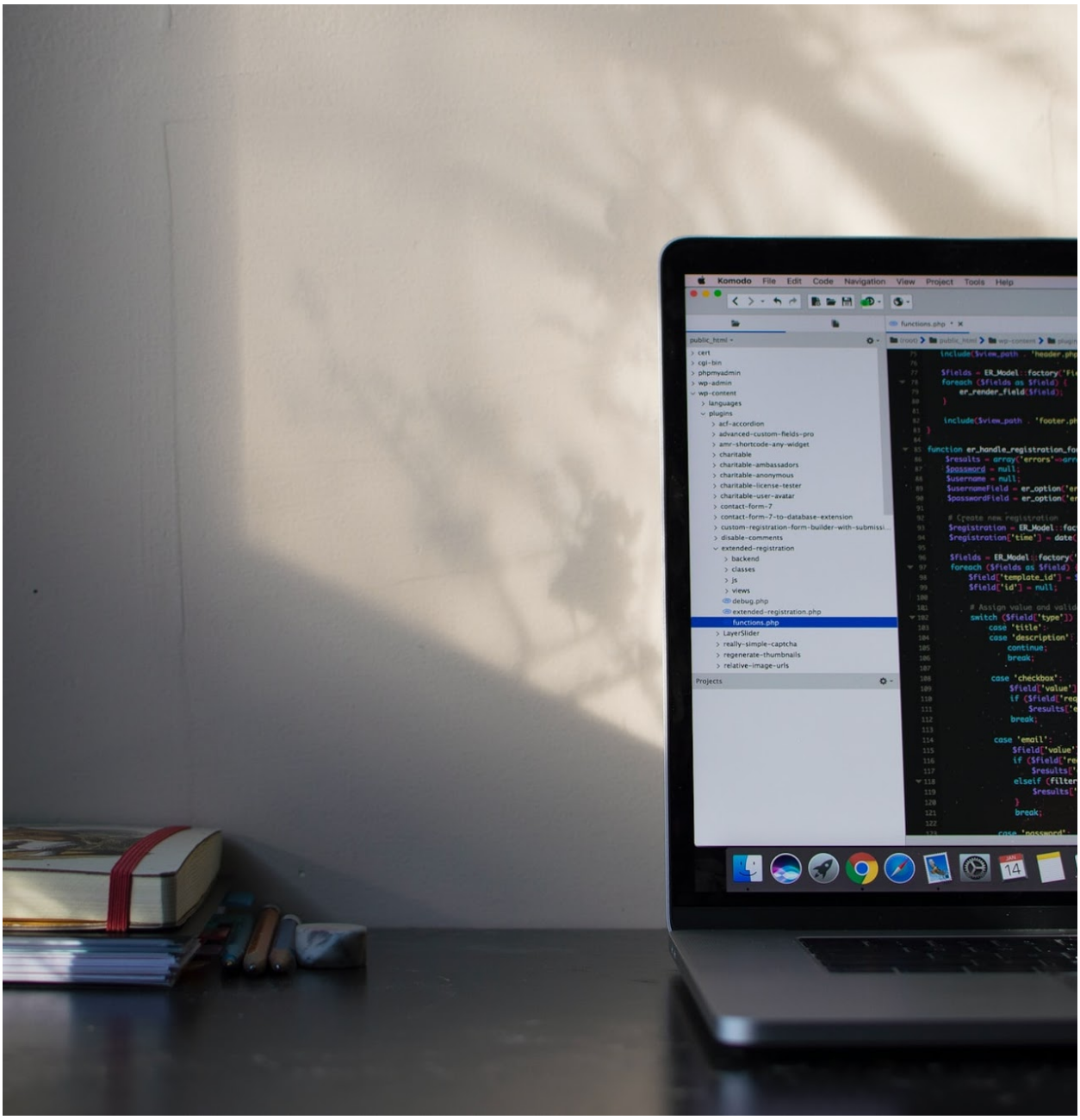
CHAPTER

THIRTEEN

GPCAM

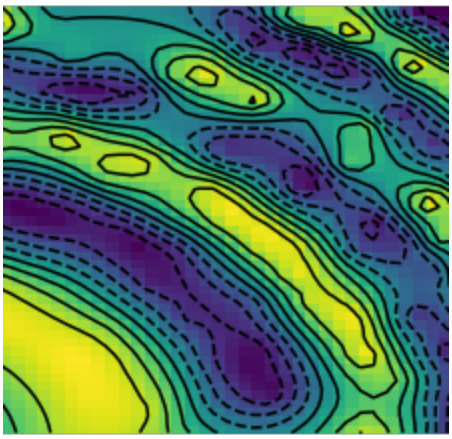
Mission of the project

gpCAM is an API and software designed to make autonomous data acquisition and analysis for experiments and simulations faster, simpler and more widely available. The tool is based on a flexible and powerful Gaussian process regression at the core. The flexibility stems from the modular design of gpCAM which allows the user to implement and import their own Python functions to customize and control almost every aspect of the software. That makes it possible to easily tune the algorithm to account for various kinds of physics and other domain knowledge, and to identify and find interesting features. A specialized function optimizer in gpCAM can take advantage of HPC architectures for fast analysis time and reactive autonomous data acquisition.



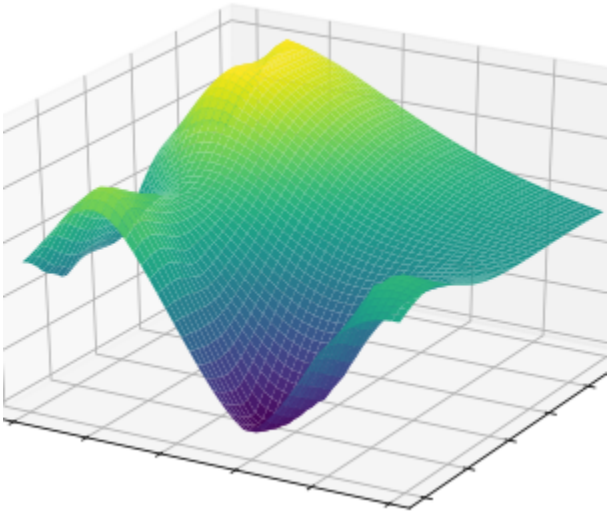
Simple API

The API is designed in a way that makes it easy to be used



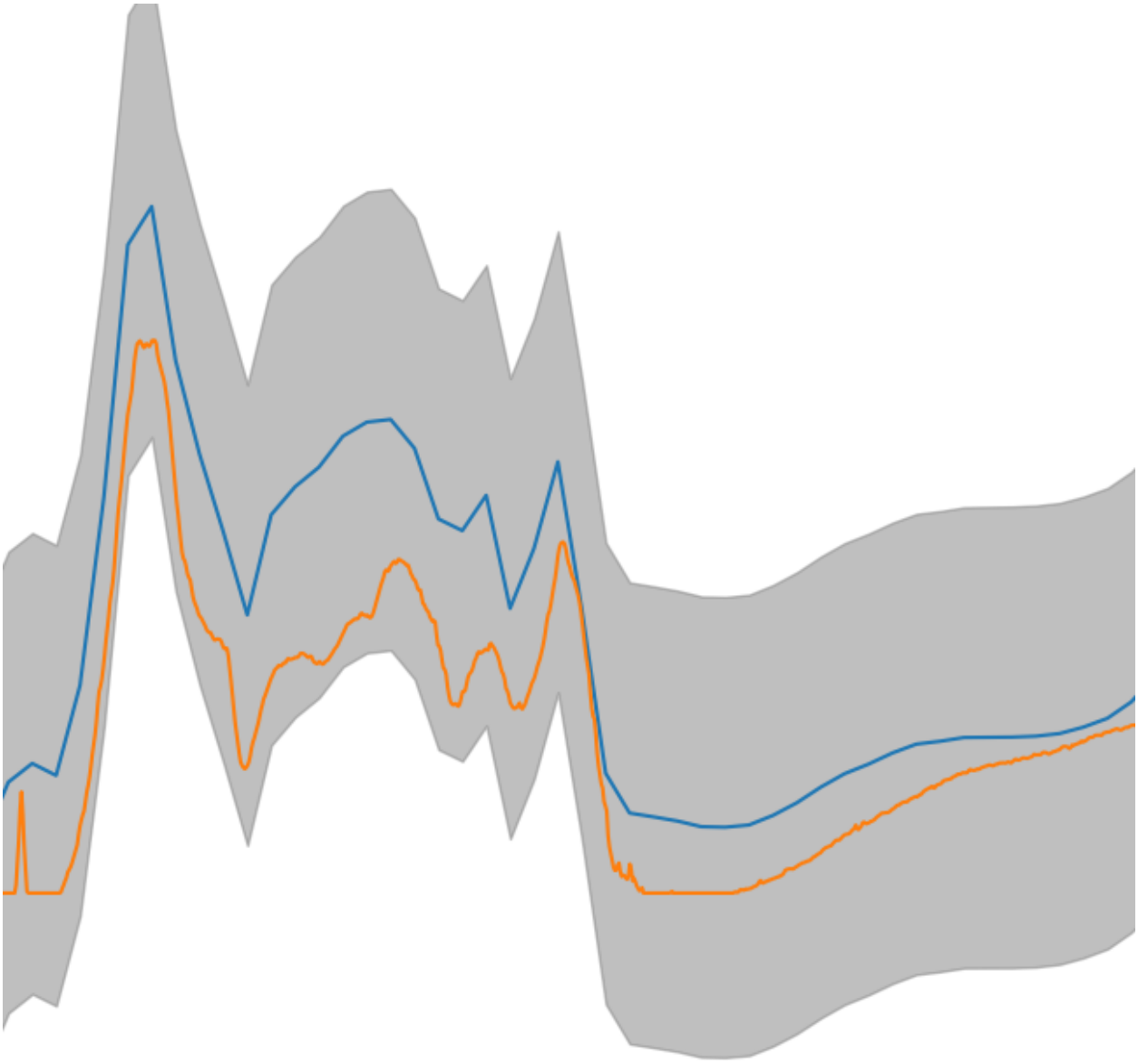
Powerful Computing

gpCAM is implemented using torch and DASK for fast training and predictions



Advanced Mathematics for Increased Flexibility

gpCAM allows the advanced user to import their own Python functions to control the training and prediction



Software for the Novice and the Expert

Simple approximation and autonomous-experimentation problems can be set up in minutes; the options for customization are endless

---

Questions?

Contact MarcusNoack@lbl.gov to get more information on the project. We also encourage you to join the SLACK channel.

Want to transform your science with autonomous data acquisition?

Take action

---

gpCAM is a software tool created by CAMERA

The Center for Advanced Mathematics for Energy Research Application

Partners

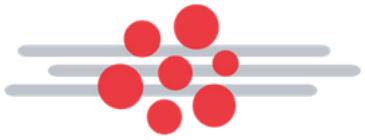

BERKELEY LAB


NEUTRONS
FOR SCIENCE


cam


UNIVERSITY OF
BIRMINGHAM


SLAC




Center for Functional Nanomaterials
Brookhaven National Laboratory


NSLS-II
National Synchrotron Light Source II


BROOKHAVEN
NATIONAL LABORATORY


UNIVERSITY OF
ALBERTA

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK


NIST


UNIVERSITÄT
BAYREUTH


BSISB
Berkeley Synchrotron Infrared
Structural Biology Imaging Program


ALS
ADVANCED LIGHT SOURCE

Supported by the US Department of Energy Office of Science
Advanced Scientific Computing Research (steven.lee@science.doe.gov)
Basic Energy Sciences (Peter.Lee@science.doe.gov)

INDEX