gpCAM

Marcus Michael Noack

May 08, 2023

SETUP

Getting started with gpCAM is simple:

1. *Install the Code*.
2. Decide on the level you want to run gpCAM on and check out the *api* based on that decision.
3. Run your experiment or simulation autonomously!

If you hit any roadblocks, read the documentation carefully, and have a look at *common bugs*.

And if some particularly stubborn problems remain, contact MarcusNoack@lbl.gov

CHAPTER

ONE

INSTALLATION

To install gpCAM do the following:

1. make sure you have Python >=3.7 installed
2. open a terminal
3. create a python environment: e.g. `python3 -m venv test_env`, for conda: `conda create --name my_cool_venv_name python=3.8`
4. activate the environment: `source ./test_env/bin/activate`, conda: `activate my_cool_venv_name` (Windows) and `source activate my_cool_venv_name` (Mac, Linux)
5. type `pip install gpcam`
6. if any problems occur, update pip `pip install --upgrade pip`, setuptools `pip install --upgrade setuptools` and repeat step 5, or try installing from source: `python -m pip install git+https://github.com/lbl-camera/gpCAM.git`

CHAPTER

TWO

EXAMPLES

Available Examples:

- Basic minimal example
- gpCAM advanced-user test
- More examples can be found on the official website

CHAPTER

THREE

COMMON BUGS AND FIXES

Error Message	Solution
Key Error: “Does not support option: ‘fastmath’	A numba error. Please update numba
Matrix Singular	Normally that means that data points are too close and different for the given kernel definition. Try using the exponential kernel, check for duplicates in the data or add more noise to the data
Value Error: Object arrays cannot be loaded when allow_pickle = False	You probably used a hand-made python function that loads a file without specifying allow_pickle = True
General installation issues	update pip, rerun installation, pip install wheel
ERROR: Failed building wheel for psutil	Rerun installation
RuntimeError: An attempt has been made to start a new process before thecurrent process has finished its bootstrapping phase.This probably means that you are not using fork to start yourchild processes and you have forgotten to use the proper idiomin the main module:if __name__ == ‘__main__’: freeze_support() …The “freeze_support()” line can be omitted if the programis not going to be frozen to produce an executable.distributed.nanny - WARNING - Restarting workerTraceback (most recent call last):File “”, line 1, in Traceback (most recent call last):File “/usr/lib/python3.8/multiprocessing/spawn.py”, line 116, in spawn_mainFile “zmq_test.py”, line 75, in exitcode = _main(fd, parent_sentinel)File “/usr/lib/python3.8/multiprocessing/spawn.py”, line 125, in _mainprepare(preparation_data)File “/usr/lib/python3.8/multiprocessing/spawn.py”, line 236, in prepare[and lot more DASK stuff]	Put all your gpCAM code in def main(): # all the gpcam code …if __name__ == “__main__” main()

CHAPTER

FOUR

LOGGING

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

4.1 Configuring logging

To enable logging in gpCAM:

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

You may also want to similarly enable logging for the `fvGP` and `HGDL` packages if these are in use.

To configure the logging level:

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

See Python's reference on levels for more info.

To log to a file:

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

Loguru provides many further options for configuration.

CHAPTER

FIVE

GPCAM API LEVELS

Starting with version 7 of gpCAM, the user has several access points (from high level to low level):

- Using the AutonomousExperimenter functionality
  - AutonomousExperimenterGP: implements an autonomous loop for a single-task GP
  - AutonomousExperimenterfvGP: implements an autonomous loop for a multi-task GP
- The user can use the gpOptimizer (already available in version 6) functionality directly to get full control. The gpOptimizer class is a function optimization wrapper around fvGP, the same is true for the fvgpOptimizer class. Using the gpOptimizer functionality means implementing your own loop
- For GP related work only, the user can use the fvgp package directly (no suggestion capability, no native steering)

For tests and examples, check out the case studies on this very 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

SIX

AUTONOMOUSEXPERIMENTER

class gpcam.autonomous_experimenter.**AutonomousExperimenterGP**(*parameter_bounds*, *hyperparameters*, *hyperparameter_bounds*, *instrument_func=None*, *init_dataset_size=None*, *acq_func='variance'*, *cost_func=None*, *cost_update_func=None*, *cost_func_params={}*, *kernel_func=None*, *prior_mean_func=None*, *run_every_iteration=None*, *x_data=None*, *y_data=None*, *variances=None*, *dataset=None*, *communicate_full_dataset=False*, *compute_device='cpu'*, *use_inv=False*, *multi_task=False*, *training_dask_client=None*, *acq_func_opt_dask_client=None*, *ram_economy=True*, *info=False*, *args=None*)

Executes the autonomous loop for a single-task Gaussian process. Use class AutonomousExperimenterfvGP for multi-task experiments.

Parameters

- **parameter_bounds** (*np.ndarray*) – A numpy array of floats of shape D x 2 describing the input space.
- **hyperparameters** (*np.ndarray*) – A 1-D numpy array of floats. The default kernel function expects a length of D+1, where the first value is a signal variance, followed by a length scale in each direction of the input space. If a kernel function is provided, then the expected length is determined by that function.
- **hyperparameter_bounds** (*np.ndarray*) – A 2-D array of floats of size J x 2, such that J is the length matching the length of *hyperparameters* defining the bounds for training.
- **instrument_func** (*Callable, optional*) – A function that takes data points (a list of dicts), and returns a similar structure with 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_func* is immediately called to measure values at these initial points.

- **acq_func** (*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: *'shannon_ig'*, *'shannon_ig_vec'*, *'ucb'*, *'maximum'*, *'minimum'*, *'covariance'*, *'variance'*, and *'gradient'*. If None, the default function is the *'variance'*, meaning *fvgp.gp.GP.posterior_covariance* with variance_only = True.
- **cost_func** (*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 1-D array of length V describing the costs as floats. The 'score' from acq_func is divided by this returned cost to determine the next measurement point. If None, the default is a uniform cost of 1.
- **cost_update_func** (*Callable, optional*) – A function that updates the *cost_func_params* which are communicated to the *cost_func*. This accepts as input costs (a list of cost values determined by *instrument_func*), bounds (a V x 2 numpy array) and parameters object. The default is a no-op.
- **cost_func_params** (*Any, optional*) – An object that is communicated to the *cost_func* and *cost_update_func*. The default is *{}*.
- **kernel_func** (*Callable, optional*) – A function that calculates the covariance between data points. It accepts as input x1 (a V x D array of positions), x2 (a U x D array of positions), hyperparameters (a 1-D array of length D+1 for the default kernel), and a *gpcam.gp_optimizer.GPOptimizer* instance. The default is a stationary anisotropic kernel (*fvgp.gp.GP.default_kernel*).
- **prior_mean_func** (*Callable, optional*) – A function that evaluates the prior mean at an input position. It accepts as input a *gpcam.gp_optimizer.GPOptimizer* instance, an array of positions (of size V x D), and hyperparameters (a 1-D array of length D+1 for the default kernel). The return value is a 1-D array of length V. If None is provided, *fvgp.gp.GP.default_mean_function* is used.
- **run_every_iteration** (*Callable, optional*) – A function that is run at every iteration. It accepts as input this *gpcam.autonomous_experimenter.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
- **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_func* 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".
- **use_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_func* computation. If None is provided, a new *dask.distributed.Client* instance is constructed.
- **info** (*bool, optional*) – bool specifying if the should be extensive std out. 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 2-D 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.gp_optimizer.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=()*, *ask_args=None*, *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 deafult 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_func* 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 *acq_func* optimization. The default is 20.
- **acq_func_opt_pop_size** (*int, optional*) – The population size used for any *acq_func* 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 *acq_func* optimization. The default value is 1e-6
- **acq_func_opt_tol_adjust** (*float, optional*) – The *acq_func* 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 *acq_func* 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.autonomous_experimenter instance
- **ask_args** (*dict, optional*) – For now, only required for acquisiton function "target probability". In this case it should be defined as {"a": some lower bound, "b":some upper bound}, example: "ask_args = {"a": 1.0,"b": 3.0}".

kill_all_clients()

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

kill_training()

Function to kill an active asynchronous training

train(*pop_size=10*, *tol=1e-06*, *max_iter=20*, *method='global'*)

Function to train the Gaussian Process. The use is entirely optional; this function will be called as part of the go() command.

Parameters

- **pop_size** (*int, optional*) – The number of individuals in case method=``global``. Default = 10
- **tol** (*float, optional*) – Convergence tolerance for the local optimizer (if method = *'local'*) Default = 1e-6
- **max_iter** (*int, optional*) – Maximum number of iterations for the global method. Default = 20
- **method** (*str, optional*) – Method to be used for the training. Default is *'global'* which means a differetnial evolution algorithm is run with the specified parameters. The options are *'global'* or *'local'*, or *'mcmc'*.

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

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

- **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'*.

update_hps()

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

CHAPTER

SEVEN

GPOPTIMIZER

class gpcam.gp_optimizer.**GPOptimizer**(*input_space_dimension*, *input_space_bounds*, *args=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 plugged into optimizers to find its maxima.

Parameters

- **input_space_dimension** (*int*) – Integer specifying the number of dimensions of the input space.
- **input_space_bounds** (*np.ndarray*) – A numpy array of floats of shape D x 2 describing the input space range
- **args** (*any, optional*) – args will be available as class attributes in kernel and prior-mean functions

x_data

Datapoint positions

Type np.ndarray

y_data

Datapoint values

Type np.ndarray

variances

Datapoint observation variances

Type np.ndarray

input_dim

Dimensionality of the input space

Type int

input_space_bounds

Bounds of the input space

Type np.ndarray

gp_initialized

A check whether the object instance has an initialized Gaussian Process.

Type bool

hyperparameters

Only available after training is executed.

Type np.ndarray

ask(*position=None*, *n=1*, *acquisition_function='variance'*, *bounds=None*, *method='global'*, *pop_size=20*, *max_iter=20*, *tol=1e-06*, *constraints=()*, *x0=None*, *vectorized=True*, *args={}*, *multi_task=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

- **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 this many suggestions for new measurements. This may be limited by how many optima the algorithm may find. If greater than 1, then the *acq_func* optimization method is automatically set to use HGDL. The default is 1.
- **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: *'shannon_ig'*, *'shannon_ig_multi'*, *'shannon_ig_vec'*, *'ucb'*, *'maximum'*, *'minimum'*, *'covariance'*, *'variance'*, and *'gradient'*. If None, the default function is the *'variance'*, meaning *fvgp.gp.GP.posterior_covariance* with variance_only = True.
- **bounds** (*np.ndarray, optional*) – A numpy array of floats of shape D x 2 describing the search range. The default is the entire input space.
- **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 acquisiiton function vecotrized to return the solution to an array of inquiries as an array, this optionmakes the optimization faster if method = 'global' is used. The default is True but will bes et to False if method is not global.
- **constraints** (*tuple of object instances, optional*) – Either a tuple of hgdl.constraints.NonLinearConstraint or scipy constraints instances, depending on the used optimizer.

- **args** (*dict, optional*) – Provides arguments for certain acquisition functions, such as, "target_probability". In this case it should be defined as {"a": some lower bound, "b":some upper bound}, example: "args = {"a": 1.0,"b": 3.0}".
- **dask_client** (*distributed.client.Client, optional*) – A Dask Distributed Client instance for distributed *acquisition_func* computation. If None is provided, a new *dask.distributed.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*, *acquisition_function='variance'*, *origin=None*)

Function to evaluate the acquisition function.

Parameters

- **x** (*np.ndarray*) – Point positions at which the acquisition function is evaluated.
- **acquisition_function** (*Callable, optional*) – Acquisiiton functio 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 1-D array of length V. The default is *variance*.
- **origin** (*np.ndarray, optional*) – If a cost function is provided this 1-D 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

init_cost(*cost_function*, *cost_function_parameters=None*, *cost_update_function=None*)

This function initializes the cost function and its parameters. If used, the acquisition function will be augmented by the costs which leads to different suggestions.

Parameters

- **cost_function** (*Callable*) – 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 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.
- **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 *gpcam.gp_optimizer.GPOptimizer.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.

Return type No return, cost function will automatically be used by GPOptimizer.ask()

init_gp(*init_hyperparameters*, *compute_device='cpu'*, *gp_kernel_function=None*, *gp_mean_function=None*, *gp_kernel_function_grad=None*, *gp_mean_function_grad=None*, *normalize_y=False*, *use_inv=False*, *ram_economy=True*)

Function to initialize the GP.

Parameters

- **init_hyperparameters** (*np.ndarray*) – Initial hyperparameters as 1-D numpy array.
- **compute_device** (*str, optional*) – One of "cpu" or "gpu", determines how linear system solves are run. The default is "cpu".
- **gp_kernel_function** (*Callable, optional*) – A function that calculates the covariance between datapoints. It accepts as input x1 (a V x D array of positions), x2 (a U x D array of positions), hyperparameters (a 1-D array of length D+1 for the default kernel), and a *gpcam.gp_optimizer.GPOptimizer* instance. The default is a stationary anisotropic kernel (*fvgp.gp.GP.default_kernel*).
- **gp_mean_function** (*Callable, optional*) – A function that evaluates the prior mean at an input position. It accepts as input a *gpcam.gp_optimizer.GPOptimizer* instance, an array of positions (of size V x D), and hyperparameters (a 1-D array of length D+1 for the default kernel). The return value is a 1-D array of length V. If None is provided, *fvgp.gp.GP.default_mean_function* is used.
- **use_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.
- **ram_economy** (*bool, optional*) – Offers a ram-efficient way to compute marginal-log-likelihood derivatives for training.

kill_async_train(*opt_obj*)

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

Parameters **opt_obj** (*object instance*) – Object instance created by gpcam.gp_optimizer.GPOptimizer.train_gp_async()

stop_async_train(*opt_obj*)

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

Parameters **opt_obj** (*object instance*) – Object instance created by gpcam.gp_optimizer.GPOptimizer.train_gp_async()

tell(*x*, *y*, *variances=None*)

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

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.
- **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.

train_gp(*hyperparameter_bounds*, *method='global'*, *pop_size=20*, *tolerance=1e-06*, *max_iter=120*, *constraints=()*, *deflation_radius=None*, *dask_client=None*)

Function to train a Gaussian Process.

Parameters

- **hyperparameters_bounds** (*np.ndarray*) – Bounds for the optimization of the hyperparameters of shape (V x 2)
- **max_iter** (*int, optional*) – Number of iterations before the optimization algorithm is terminated. The default is 120
- **method** (*str or callable, optional*) – Optimization method. Choose from *'local'* or *'global'*, or *'mcmc'*. The default is *global*. The argument also accepts a callable that accepts as input a *fvgp.gp.GP* instance and returns a new vector of hyperparameters.
- **pop_size** (*int, optional*) – The number of individuals used if *global* is chosen as method.
- **tolerance** (*float, optional*) – Tolerance to be used to define a termination criterion for the optimizer.
- **constraints** (*tuple of object instances, optional*) – Scipy constraints instances, depending on the used optimizer.

Returns **hyperparameters** – This is just informative, the Gaussian Process is automatically updated.

Return type np.ndarray

train_gp_async(*hyperparameter_bounds*, *max_iter=10000*, *dask_client=None*, *deflation_radius=None*, *constraints=()*, *local_method='L-BFGS-B'*, *global_method='genetic'*)

Function to train a Gaussian Process asynchronously on distributed HPC compute architecture using the *HGDL* software package.

Parameters

- **hyperparameters_bounds** (*np.ndarray*) – Bounds for the optimization of the hyperparameters of shape (V x 2)
- **max_iter** (*int, optional*) – Number of iterations before the optimization algorithm is terminated. Since the algorithm works asynchronously, this number can be high. The default is 10000
- **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 training. If None is provided, a local *dask.distributed.Client* instance is constructed.
- **local_method** (*str, optional*) – Controls the local optimizer running in *HGDL*. Many scipy.minimize optimizers can be used, in addition, "dNewton". *L-BFGS-B* is the default.
- **global_method** (*str, optional*) – Global optimization step running in *HGDL*. Choose from *genetic* or 'random'. The default is *genetic*

Returns

- *This function return an optimization object (opt_obj) that can be used to stop(opt_obj) or kill(opt_obj)*

- *the optimization.*
- Call *gpcam.gp_optimizer.GPOptimizer.update_hyperparameters(opt_obj)* to update the
- *current Gaussian Process with the new hyperparameters. This allows to start several optimization procedures*
- *and selectively use or stop them.*

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 how they are used by the cost function. :param measurement_costs: 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.

Return type No return, the cost function parameters will automatically be updated.

update_hyperparameters(*opt_obj*)

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

Parameters **opt_obj** (*object instance*) – Object instance created by gpcam.gp_optimizer.GPOptimizer.train_gp_async()

Returns **hyperparameters**

Return type np.ndarray

CHAPTER

EIGHT

FVGPOPTIMIZER

class gpcam.gp_optimizer.**fvGPOptimizer**(*input_space_dimension*, *output_space_dimension*, *output_number*, *input_space_bounds*, *args=None*)

This class is an optimization wrapper around the fvgp package for multi-task (multi-variate) Gaussian Processes. Gaussian Processes can be initialized, trained, and conditioned; also the posterior can be evaluated and plugged into optimizers to find its maxima.

Parameters

- **input_space_dimension** (*int*) – Integer specifying the number of dimensions of the input space.
- **output_space_dimension** (*int*) – Integer specifying the number of dimensions of the output space. Most often 1.
- **output_number** (*int*) – Number of output values.
- **input_space_bounds** (*np.ndarray*) – A numpy array of floats of shape D x 2 describing the input space range
- **args** (*any, optional*) – args will be available as class attributes in kernel and prior-mean functions

x_data

Datapoint positions

Type np.ndarray

y_data

Datapoint values

Type np.ndarray

variances

Datapoint observation variances

Type np.ndarray

input_dim

Dimensionality of the input space

Type int

input_space_bounds

Bounds of the input space

Type np.ndarray

gp_initialized

A check whether the object instance has an initialized Gaussian Process.

Type bool

hyperparameters

Only available after the training is executed.

Type np.ndarray

get_data_fvGP()

Function that provides a way to access the class attributes.

Returns **dictionary of class attributes**

Return type dict

init_fvgp(*init_hyperparameters*, *compute_device='cpu'*, *gp_kernel_function=None*, *gp_mean_function=None*, *use_inv=False*, *ram_economy=True*)

Function to initialize the multi-task GP.

Parameters

- **init_hyperparameters** (*np.ndarray*) – Initial hyperparameters as 1-D numpy array.
- **compute_device** (*str, optional*) – One of "cpu" or "gpu", determines how linear system solves are run. The default is "cpu".
- **gp_kernel_function** (*Callable, optional*) – A function that calculates the covariance between datapoints. It accepts as input x1 (a V x D array of positions), x2 (a U x D array of positions), hyperparameters (a 1-D array of length D+1 for the default kernel), and a *gpcam.gp_optimizer.GPOptimizer* instance. The default is a stationary anisotropic kernel (*fvgp.gp.GP.default_kernel*).
- **gp_mean_function** (*Callable, optional*) – A function that evaluates the prior mean at an input position. It accepts as input a *gpcam.gp_optimizer.GPOptimizer* instance, an array of positions (of size V x D), and hyperparameters (a 1-D array of length D+1 for the default kernel). The return value is a 1-D array of length V. If None is provided, *fvgp.gp.GP.default_mean_function* is used.
- **use_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.
- **ram_economy** (*bool, optional*) – Offers a ram-efficient way to compute marginal-log-likelihood derivatives for training.

tell(*x*, *y*, *variances=None*, *value_positions=None*)

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

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.

- **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.
- **value_positions** (*np.ndarray, optional*) – A 3-D numpy array of shape (U x output_number x output_dim), so that for each measurement position, the outputs are clearly defined by their positions in the output space. The default is np.array([[0],[1],[2],[3],...,[output_number - 1]]) for each point in the input space. The default is only permissible if output_dim is 1.

update_fvgp()

This function updates the data in the fvGP, tell(...) will call this function automatically if GP is already intialized

CHAPTER

NINE

ADVANCED USE OF GPCAM

The advanced use of gpCAM is about communicating domain knowledge in the form of kernel, acquisition and mean functions, and optimization constraints.

9.1 Prior-Mean Functions to Communicate Trends

Often times an overall trend of the model is known in absolute terms or in parametric form. In that case, the user may define their own prior mean function following the example below.

```
def himmel_blau(x,hyperparameters, gp_obj):
    return (x[:,0] ** 2 + x[:,1] - 11.0) ** 2 + (x[:,0] + x[:,1] ** 2 - 7.0) ** 2
```

9.2 Tailored Acquisition Functions for Feature Finding

The acquisition function uses the output of a Gaussian process to steer the experiment or simulation to high-value regions of the search space. You can find an example below.

```
def upper_confidence_bounds(x,obj):
    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)   ##which is 1-d numpy array
```

9.3 Tailored Kernel Functions for Hard Constraints on the Posterior Mean

Kernel functions are a tremendously powerful tool to communicate hard constraints to the Gaussian process. Examples include the order of differentiability, periodicity, and symmetry of the model function. The kernel can be defined in the way presented below.

```
def kernel_l2_single_task(x1,x2,hyperparameters,obj):
    hps = hyperparameters
    distance_matrix = np.zeros((len(x1),len(x2)))

    for i in range(len(x1[0])-1):
```

(continues on next page)

(continued from previous page)

```
            distance_matrix += abs(np.subtract.outer(x1[:,i],x2[:,i])/hps[1+i])**2

    distance_matrix = np.sqrt(distance_matrix)

    return   hps[0] *  obj.matern_kernel_diff1(distance_matrix,1)
```

9.4 Tailored Cost Functions for Optimizing Data Acquisition when Costs are Present

Cost functions are very useful when the main effort of exploration does not come from the data acquisition itself but from the motion through the search space. gpCAM can use cost and cost update functions. You can find examples for both below. If costs are recorded during data acquisition, gpCAM can use them to update the cost function repeatedly.

```
def l2_cost(origin,x,arguments = None):
    offset = arguments["offset"]
    slope = arguments["slope"]
    return slope*np.linalg.norm(np.abs(np.subtract(origin,x)), axis = 1)+offset
```

```
def update_l2_cost_function(costs, bounds, parameters):
    print("Cost adjustment in progress...")
    print("old cost parameters: ",parameters)
    ###remove outliers:
    origins = []
    points = []
    motions = []
    c   = []
    cost_per_motion = []

    for i in range(len(costs)):
        origins.append(costs[i]["origin"])
        points.append(costs[i]["point"])
        motions.append(abs(costs[i]["origin"] - costs[i]["point"]))
        c.append(costs[i]["cost"])
        cost_per_motion.append(costs[i]["cost"]/l2_cost(costs[i]["origin"],costs[i][
↪"point"], parameters))

    mean_costs_per_distance = np.mean(np.asarray(cost_per_motion))
    sd = np.std(np.asarray(cost_per_motion))

    for element in cost_per_motion:
        if (
            element >= mean_costs_per_distance - 2.0 * sd
            and element <= mean_costs_per_distance + 2.0 * sd
        ):
            continue
        else:
            motions.pop(cost_per_motion.index(element))
            c.pop(cost_per_motion.index(element))
            origins.pop(cost_per_motion.index(element))
```

(continues on next page)

(continued from previous page)

```
            points.pop(cost_per_motion.index(element))
            cost_per_motion.pop(cost_per_motion.index(element))

    res = devo(compute_l2_cost_misfit, bounds, args = (origins, points,c), tol=1e-6,␣
↪disp=True, maxiter=300, popsize=20, polish=False)
    arguments = {"offset": res["x"][0],"slope": res["x"][1]}
    print("New cost parameters: ", arguments)
    return arguments
```

9.5 Constrained Optimization

It is now possible to create hgdl.constraints.NonLinearConstraint object instances and communicate them to gp_optimizer.train and gp_optimizer.train_async(). Setting this up is a little tricky but potentially very beneficial. Have a look at the HGDL documentation.

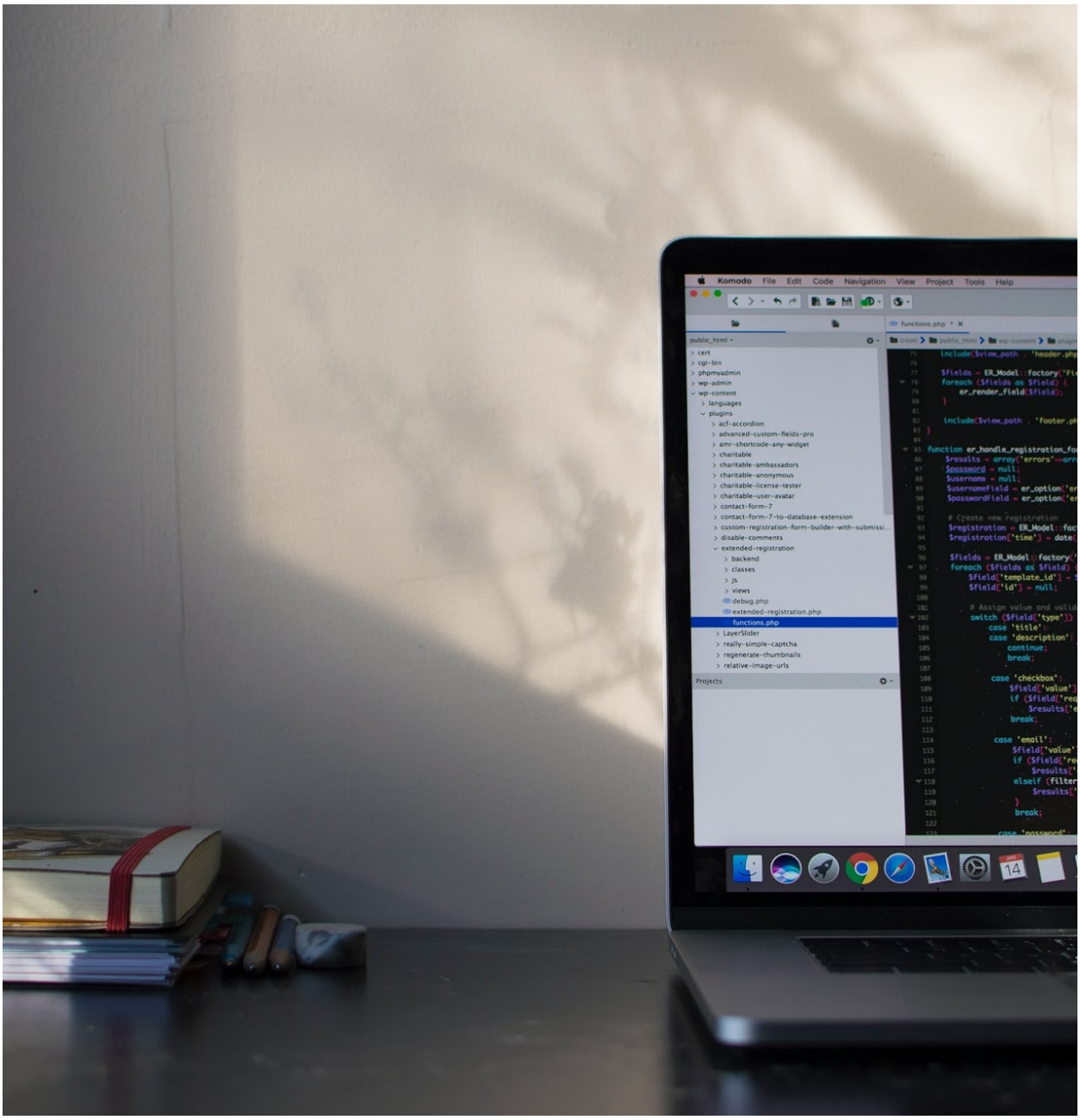
CHAPTER

TEN

GPCAM

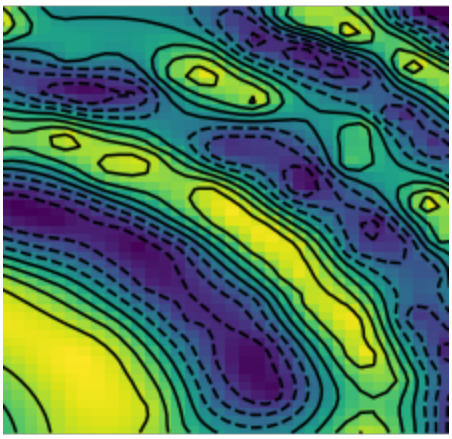
Mission of the project

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.



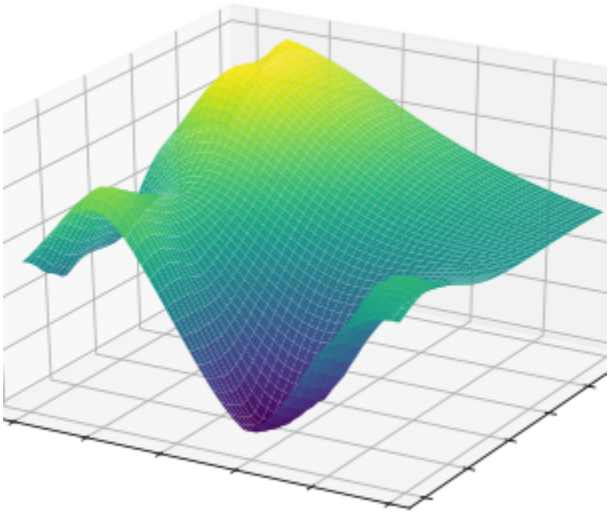
Simple API

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



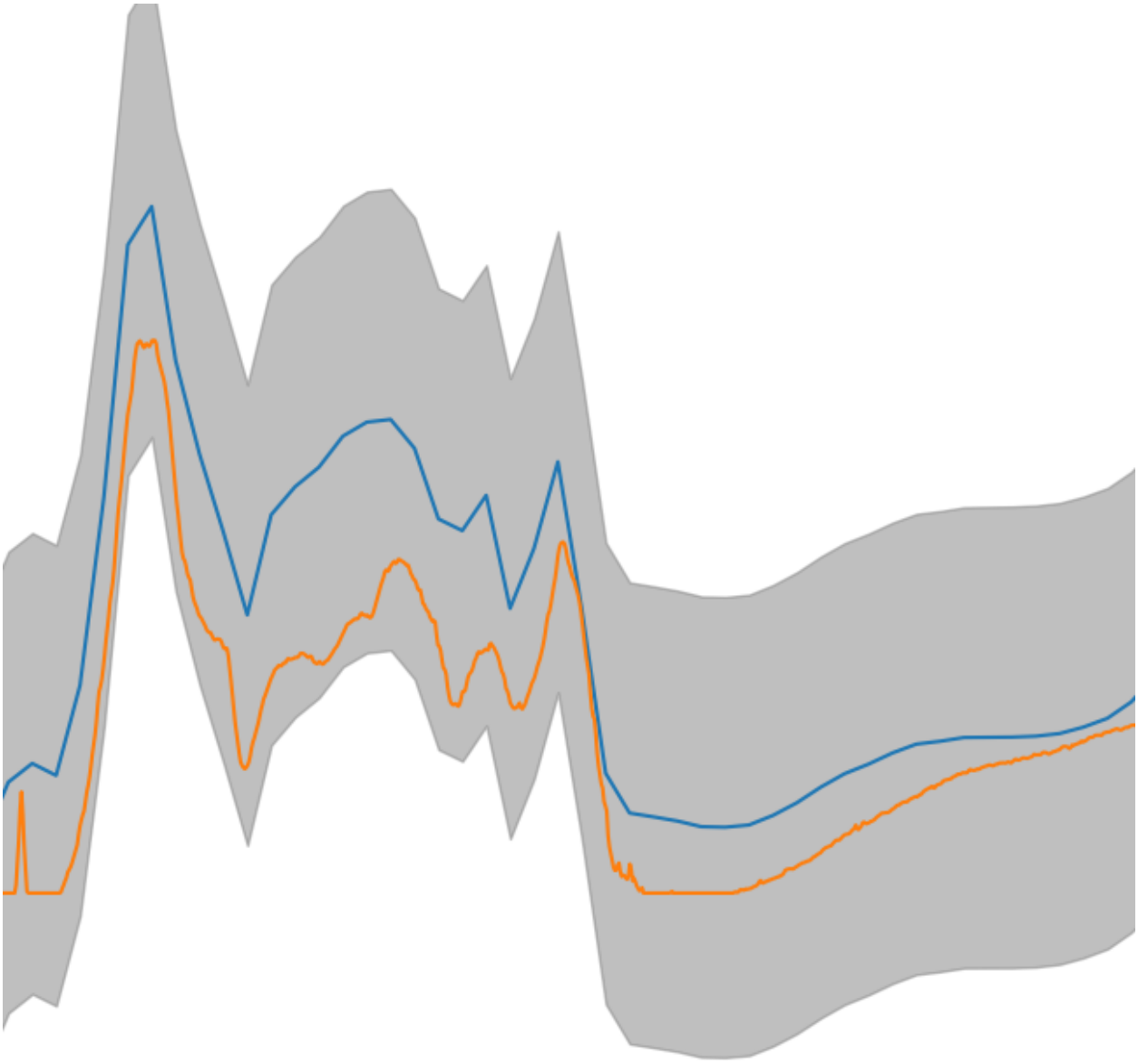
Powerful Computing

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

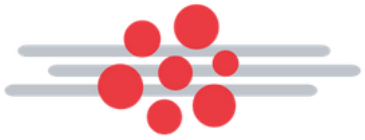

BERKELEY LAB


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


BSISB
Berkeley Synchrotron Infrared
Structural Biology Imaging Program


ALS
ADVANCED LIGHT SOURCE


UNIVERSITÄT
BAYREUTH

U.S. DEPARTMENT OF ENERGY | Office of Science

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

U

V

X

Y