HGDL

Release 2.1.3

Marcus Noack, David Perryman, Harinarayan Krishnan, Petrus Zw

Sep 29, 2023

API

class hgdl.hgdl.**HGDL**(*func*, *grad*, *bounds*, *hess=None*, *num_epochs=100000*, *global_optimizer='genetic'*, *local_optimizer='L-BFGS-B'*, *number_of_optima=1000000*, *local_max_iter=1000*, *constraints=()*, *args=()*)

This is HGDL, a class for asynchronous HPC-capable optimization.

H . . . Hybrid

G . . . Global

D . . . Deflated

L . . . Local

The algorithm places a number of walkers inside the domain (the number is determined by the dask client), all of which perform a local optimization in a distributed way in parallel. When the walkers have identified local optima, their positions are communicated back to the host who removes the found optima by deflation, and replaces the fittest walkers by a global optimization step. From here the next epoch begins with distributed local optimizations of the new walkers. The algorithm results in a sorted list of unique optima (only if optima are of the form f'(x) = 0) The method *hgdl.optimize* instantly returns a result object that can be queried for a growing, sorted list of optima. If a Hessian is provided, those optima are classified as minima, maxima or saddle points.

Parameters

- **func** (*Callable*) – The function to be MINIMIZED. A callable that accepts an np.ndarray and optional arguments, and returns a scalar.
- **grad** (*Callable*) – The gradient of the function to be MINIMIZED. A callable that accepts an np.ndarray and optional arguments, and returns a vector (np.ndarray) of shape (D), where D is the dimensionality of the space in which the optimization takes place.
- **bounds** (*np.ndarray*) – The bounds of the domain; an np.ndarray of shape (D x 2), where D is the dimensionality of the space in which the optimization takes place. Here D is the dimension of the input domain.
- **hess** (*Callable, optional*) – The Hessian of the function to be MINIMIZED. A callable that accepts an np.ndarray and optional arguments, and returns a np.ndarray of shape (D x D). The default value is no-op.
- **num_epochs** (*int, optional*) – The number of epochs the algorithm runs through before being terminated. One epoch is the convergence of all local walkers, the deflation of the identified optima, and the global replacement of the walkers. Note, the algorithm is running asynchronously, so a high number of epochs can be chosen without concerns, it will not affect the run time to obtain the optima. Therefore, the default is 100000.
- **global_optimizer** (*Callable or str, optional*) – The function (identified by a string or a Callable) that replaces the fittest walkers after their local convergence. The possible options are *genetic* (default), *random* or a callable that accepts an np.ndarray of shape (U x D) of positions, an np.ndarray of shape (U) of function values, and np.ndarray of shape (D x 2) of bounds, and an integer specifying the number of offspring individuals that should be returned. The callable should return the positions of the offspring individuals as an np.ndarray of shape (number_of_offspring x D).
- **local_optimizer** (*Callable or str, optional*) – The local optimizer that is used. The options are *dNewton* (default), *L-BFGS-B*, *BFGS*, *CG*, *Newton-CG*, *SLSQP*. The above methods have been tested, but most others should work. Visit the *scipy.optimize.minimize* docs (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html) for specifications and limitations of the local methods. The parameter also accepts a callable of the form func(f,grad,hess,bounds,x0,*args), and returns an object equal to the scipy.optimize.minimize methods.

- **number_of_optima** (*int, optional*) – The number of optima that will be stored in the optima list and deflated. The default is 1e6. After that number is reached, worse-performing optima will not be stored or deflated.
- **local_max_iter** (*int, optional*) – The number of iterations before local optimizations are terminated. The default is 1000. It can be lowered when second-order local optimizers are used.
- **args** (*tuple, optional*) – A tuple of arguments that will be communicated to the function, the gradient, and the Hessian callables. Default = ().
- **constraints** (*object, optional*) – An optional n-tuple of constraint objects. The default is no constraints (). Constraints are defined following scipy.optimize.NonlinearConstraint.

optima

Contains the attribute optima.list in which the optima are stored. However, the method 'get_latest()' should be used to access the optima.

Type object

optimize(*dask_client=None*, *x0=None*, *tolerance=1e-08*)

Function to start the optimization. Note, this function will not return anything. Use the method hgdl.HGDL.get_latest() (non-blocking) or hgdl.HGDL.get_final() (blocking) to query results.

Parameters

- **dask_client** (*distributed.client.Client, optional*) – The client that will be used for the distibuted local optimizations. The default is a local client.
- **x0** (*np.ndarray, optional*) – An np.ndarray of shape (V x D) of points used as starting positions. If V > number of walkers (specified by the dask client) the array will be truncated. If V < number of walkers, random points will be appended. The default is None, meaning only random points will be used.
- **tolerance** (*float, optional*) – The tolerance used by the local optimizers. The default is 1e-6

get_client_info()

Function to receive info about the workers.

get_latest()

Function to request the current result. No inputs

get_final()

Function to request the final result. CAUTION: This function will block the main thread until the end of all epochs is reached. No inputs.

cancel_tasks()

Function to cancel all tasks and therefore the execution. However, this function does not kill the client.

kill_client()

Function to cancel all tasks and kill the dask client, and therefore the execution. If cancel_tasks() is called before, this will throw an error.

CHAPTER

ONE

LOGGING

The HGDL 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.

1.1 Configuring logging

To enable logging in HGDL:

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

To configure the logging level:

```
logger.add(sys.stderr, filter="hgdl", 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

TWO

HGDL CONSTRAINED OPTIMIZATION OF ROSENBROCK'S FUNCTION

In this script, we show how HGDL is used for constrained optimization. Unconstrained optimization is simpler and automatically included.

2.1 First some function to make nice plots

```
%load_ext autoreload
%autoreload 2
import numpy as np
import plotly.graph_objects as go
def plot(x,y,z,data = None, constr = 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] + 50,
                                   mode='markers'))
    if constr is not None:
        fig.add_trace(go.Scatter3d(x=constr[:,0], y=constr[:,1], z=constr[:,2],
                                   mode='markers'))

    fig.update_layout(title='Surface Plot', autosize=True,
                  width=800, height=800, font=dict(
                  family="Courier New, monospace",
                  size=18),
                  margin=dict(l=65, r=50, b=65, t=90))

    fig.show()

def make_plot(data = None, constraint1 = None, constraint2 = None):
    x1,x2 = np.linspace(-500,500,100),np.linspace(-500,500,100)
    x_pred = np.transpose([np.tile(x1, len(x2)), np.repeat(x2, len(x1))])
    r1 = np.sqrt(160000.)
    r2 = np.sqrt(40000.)
    c1,c2 = r1*np.cos(np.linspace(0,2.*np.pi,100)),r1*np.sin(np.linspace(0,2.*np.pi,100))
    d1,d2 = r2*np.cos(np.linspace(0,2.*np.pi,100)),r2*np.sin(np.linspace(0,2.*np.pi,100))
    c3 = np.zeros((len(c2)))
    d3 = np.zeros((len(c2)))
    x1,x2 = np.meshgrid(x1,x2)
```

(continues on next page)

(continued from previous page)

```
    z = np.zeros((10000))
    zc1 = np.zeros((10000))
    zc2 = np.zeros((10000))
    for i in range(10000):
        z[i] = rosen(x_pred[i])
        if constraint1: zc1[i] = constraint1(x_pred[i])
        if constraint2: zc2[i] = constraint2(x_pred[i])
    for i in range(len(c1)):
        c3[i] = rosen(np.array([c1[i],c2[i]]))
        d3[i] = rosen(np.array([d1[i],d2[i]]))
    plot(x1,x2,z.reshape(100,100).T, data = data,
         constr = np.row_stack([np.column_stack([c1,c2,c3]),np.column_stack([d1,d2,
↪d3])]))
```

2.2 Defining the Constraints and some Bounds

Keep in mind that not all local optimizers allow any combination of bounds and constraints Visit https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html for more information on that

```
bounds = np.array([[-4,4],[-4,4]])
def g1(x):
    return (np.linalg.norm(x)**2/10.0) - 2.0
```

2.3 Now we import HGDL and run a constrained optimization

```
from hgdl.hgdl import HGDL as hgdl
from hgdl.support_functions import *
import time
from scipy.optimize import rosen, rosen_der, rosen_hess


#constraint definitions form scipy
from scipy.optimize import NonlinearConstraint
nlc = NonlinearConstraint(g1, -np.inf, 0)


a = hgdl(rosen, rosen_der, bounds,
            hess = rosen_hess, ##if this is None, the Hessian will be approximated if␣
↪the local optimizer needs it
            global_optimizer = "random", #there are a few options to choose from for the␣
↪global optimizer
            #global_optimizer = "genetic",
            local_optimizer = "dNewton", #dNewton is an example and will be changed␣
↪automatically to "SLSQP" because constraints are used
            number_of_optima = 30000, #the number fo optima that will be stored and used␣
↪for deflation
            args = (), num_epochs = 1000, #the number fo total epochs. Since this is an␣
↪asynchronous algorithms, this number can be very high
```

(continues on next page)

(continued from previous page)

```
            constraints = (nlc,) #the constraints
            )

a.optimize(x0=None)
time.sleep(10)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/hgdl/envs/stable/lib/python3.9/site-
↪packages/hgdl/hgdl.py:134: UserWarning: Warning: dNewton will not adhere to bounds. It␣
↪is recommended to formulate your objective function such that it is defined on R^N by␣
↪simple non-linear transformations.
  warnings.warn("Warning: dNewton will not adhere to bounds. It is recommended to␣
↪formulate your objective function such that it is defined on R^N by simple non-linear␣
↪transformations.")
/home/docs/checkouts/readthedocs.org/user_builds/hgdl/envs/stable/lib/python3.9/site-
↪packages/hgdl/hgdl.py:137: UserWarning: Constraints provided, local optimizer changed␣
↪to 'SLSQP'
  warnings.warn("Constraints provided, local optimizer changed to 'SLSQP'")
```

```
/home/docs/checkouts/readthedocs.org/user_builds/hgdl/envs/stable/lib/python3.9/site-
↪packages/scipy/optimize/_minimize.py:565: RuntimeWarning: Method SLSQP does not use␣
↪Hessian information (hess).
  warn('Method %s does not use Hessian information (hess).' % method,
```

```
res = a.get_latest()
for entry in res: print(entry)
```

```
{'x': array([1.00000392, 1.00000722]), 'f(x)': 5.3222399828924006e-11, 'classifier':
↪'minimum', 'Hessian eigvals': array([1.00160711e+03, 3.99407338e-01]), 'df/dx':␣
↪array([ 0.00025398, -0.00012307]), '|df/dx|': 0.00028223142458595323, 'radius': 2.
↪5037096328528436}
```

```
res = a.kill_client()
```

```
2023-09-29 22:50:16,381 - distributed.worker.state_machine - WARNING - Async instruction␣
↪for <Task cancelled name='execute(local_method-29baadea26817556aeccb33572d49afc)' coro=
↪<Worker.execute() done, defined at /home/docs/checkouts/readthedocs.org/user_builds/
↪hgdl/envs/stable/lib/python3.9/site-packages/distributed/worker_state_machine.py:3609>>
↪ ended with CancelledError
2023-09-29 22:50:16,383 - distributed.worker.state_machine - WARNING - Async instruction␣
↪for <Task cancelled name='execute(hgdl-b6d02f5aace9cb386f1fb2d05149742e)' coro=<Worker.
↪execute() done, defined at /home/docs/checkouts/readthedocs.org/user_builds/hgdl/envs/
↪stable/lib/python3.9/site-packages/distributed/worker_state_machine.py:3609>> ended␣
↪with CancelledError
```

2.4 Making a Plot

You should see the constraints and the foudn optima. If everything worked, the found points are in between the two constraints.

```
data = [np.append(entry["x"],entry["f(x)"]) for entry in res]
make_plot(data = np.array(data), constraint1 = g1)
```

CHAPTER

THREE

HGDL CONSTRAINED OPTIMIZATION OF SCHWEFEL'S FUNCTION

In this script, we show how HGDL is used for constrained optimization. Unconstrained optimization is simpler and automatically included.

3.1 First some function to make nice plots

```
%load_ext autoreload
%autoreload 2
import numpy as np
import plotly.graph_objects as go
def plot(x,y,z,data = None, constr = 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] + 50,
                                   mode='markers'))
    if constr is not None:
        fig.add_trace(go.Scatter3d(x=constr[:,0], y=constr[:,1], z=constr[:,2],
                                   mode='markers'))

    fig.update_layout(title='Surface Plot', autosize=True,
                  width=800, height=800, font=dict(
                  family="Courier New, monospace",
                  size=18),
                  margin=dict(l=65, r=50, b=65, t=90))

    fig.show()

def make_plot(data = None, constraint1 = None, constraint2 = None):
    x1,x2 = np.linspace(-500,500,100),np.linspace(-500,500,100)
    x_pred = np.transpose([np.tile(x1, len(x2)), np.repeat(x2, len(x1))])
    r1 = np.sqrt(160000.)
    r2 = np.sqrt(40000.)
    c1,c2 = r1*np.cos(np.linspace(0,2.*np.pi,100)),r1*np.sin(np.linspace(0,2.*np.pi,100))
    d1,d2 = r2*np.cos(np.linspace(0,2.*np.pi,100)),r2*np.sin(np.linspace(0,2.*np.pi,100))
    c3 = np.zeros((len(c2)))
    d3 = np.zeros((len(c2)))
    x1,x2 = np.meshgrid(x1,x2)
    z = np.zeros((10000))
```

(continues on next page)

(continued from previous page)

```
    zc1 = np.zeros((10000))
    zc2 = np.zeros((10000))
    for i in range(10000):
        z[i] = schwefel(x_pred[i], 1, 1)
        if constraint1: zc1[i] = constraint1(x_pred[i])
        if constraint2: zc2[i] = constraint2(x_pred[i])
    for i in range(len(c1)):
        c3[i] = schwefel(np.array([c1[i],c2[i]]))
        d3[i] = schwefel(np.array([d1[i],d2[i]]))
    plot(x1,x2,z.reshape(100,100).T, data = data,
         constr = np.row_stack([np.column_stack([c1,c2,c3]),np.column_stack([d1,d2,
↪d3])]))
```

3.2 Defining the Constraints and some Bounds

Keep in mind that not all local optimizers allow any combination of bounds and constraints Visit https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html for more information on that

```
bounds = np.array([[-500,500],[-500,500]])
def g1(x):
    return (np.linalg.norm(x)**2/10.0) - 16000.0
def g2(x):
    return (np.linalg.norm(x)**2/10.0) - 4000.0
```

3.3 Now we import HGDL and run a constrained optimization

```
from hgdl.hgdl import HGDL as hgdl
from hgdl.support_functions import *
import time

#example arguments that will be passed to the function, the gradient and the Hessian␣
↪function
a  = 5
b  = 6

#constraint definitions form scipy
from scipy.optimize import NonlinearConstraint
nlc = NonlinearConstraint(g1, -np.inf, 0)
nlc = NonlinearConstraint(g2, 0,np.inf)


a = hgdl(schwefel, schwefel_gradient, bounds,
           hess = None, ##if this is None, the Hessian will be approximated if the␣
↪local optimizer needs it
           global_optimizer = "random", #there are a few options to choose from for the␣
↪global optimizer
           #global_optimizer = "genetic",
           local_optimizer = "dNewton", #dNewton is an example and will be changed␣
↪automatically to "SLSQP" because constraints are used
```

(continues on next page)

(continued from previous page)

```
            number_of_optima = 30000, #the number fo optima that will be stored and used␣
↪for deflation
            args = (a,b), num_epochs = 1000, #the number of total epochs. Since this is␣
↪an asynchronous algorithms, this number can be very high
            constraints = (nlc,) #the constraints
            )

a.optimize(x0=None)
time.sleep(10)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/hgdl/envs/stable/lib/python3.9/site-
↪packages/hgdl/hgdl.py:134: UserWarning: Warning: dNewton will not adhere to bounds. It␣
↪is recommended to formulate your objective function such that it is defined on R^N by␣
↪simple non-linear transformations.
  warnings.warn("Warning: dNewton will not adhere to bounds. It is recommended to␣
↪formulate your objective function such that it is defined on R^N by simple non-linear␣
↪transformations.")
/home/docs/checkouts/readthedocs.org/user_builds/hgdl/envs/stable/lib/python3.9/site-
↪packages/hgdl/hgdl.py:137: UserWarning: Constraints provided, local optimizer changed␣
↪to 'SLSQP'
  warnings.warn("Constraints provided, local optimizer changed to 'SLSQP'")
```

```
/home/docs/checkouts/readthedocs.org/user_builds/hgdl/envs/stable/lib/python3.9/site-
↪packages/scipy/optimize/_minimize.py:565: RuntimeWarning: Method SLSQP does not use␣
↪Hessian information (hess).
  warn('Method %s does not use Hessian information (hess).' % method,
```

```
res = a.get_latest()
for entry in res: print(entry)
```

```
{'x': array([420.96874751, 420.96874742]), 'f(x)': 2.5455132799834246e-05, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25236706, 0.25236706]), 'df/dx': array([2.
↪90366101e-07, 2.66268203e-07]), '|df/dx|': 3.9396856247078785e-07, 'radius': 3.
↪9624822751869258}
{'x': array([ 420.9687584 , -302.52494415]), 'f(x)': 118.43836006959793, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25236707, 0.25328927]), 'df/dx': array([ 3.
↪03755323e-06, -2.16373546e-06]), '|df/dx|': 3.729407564262412e-06, 'radius': 3.
↪9624821758250595}
{'x': array([-302.52499091,  420.96880724]), 'f(x)': 118.43836007042546, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25328931, 0.25236706]), 'df/dx': array([-1.
↪40067597e-05,  1.53629991e-05]), '|df/dx|': 2.0789686354264693e-05, 'radius': 3.
↪9624822943620233}
{'x': array([420.96874191, 203.81425321]), 'f(x)': 217.13969484570794, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25236706, 0.25487078]), 'df/dx': array([ 4.
↪70988725e-05, -4.01084458e-05]), '|df/dx|': 6.186268027021028e-05, 'radius': 3.
↪962482327482647}
{'x': array([203.81424493, 420.96876898]), 'f(x)': 217.13969484577774, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25487078, 0.25236707]), 'df/dx': array([-1.
↪96714689e-06,  5.70976703e-06]), '|df/dx|': 6.039131260946407e-06, 'radius': 3.
↪9624820799495795}
{'x': array([-302.52494555, -302.52505257]), 'f(x)': 236.87669468575336, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25328927, 0.25328933]), 'df/dx': array([-2.
↪517277e-06, -2.962543e-05]), '|df/dx|': 2.9732184301132748e-05, 'radius': 3.
↪9480551401704873}
```

(continues on next page)

(continued from previous page)

```
{'x': array([ 420.96874628, -124.8293565 ]), 'f(x)': 296.10673921365026, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25236706, 0.25791668]), 'df/dx': array([-1.
↪93766073e-08, -2.15171760e-08]), '|df/dx|': 2.895585909548802e-08, 'radius': 3.
↪962482283902879}
{'x': array([-124.82936586,  420.96878856]), 'f(x)': 296.1067392138866, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25791668, 0.25236705]), 'df/dx': array([-2.
↪43462183e-06,  1.06505996e-05]), '|df/dx|': 1.0925321753875174e-05, 'radius': 3.
↪9624824704242916}
{'x': array([ 203.81424801, -302.52494998]), 'f(x)': 335.5780294601722, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25487078, 0.25328927]), 'df/dx': array([-1.
↪18279553e-06, -3.63943755e-06]), '|df/dx|': 3.826814727190929e-06, 'radius': 3.
↪9480550813328046}
{'x': array([-302.52494271,  203.81420496]), 'f(x)': 335.57802946043955, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25328927, 0.25487076]), 'df/dx': array([-1.
↪79866389e-06, -1.21552982e-05]), '|df/dx|': 1.2287654979809248e-05, 'radius': 3.
↪9480551782419298}
{'x': array([420.96874593,  65.54786442]), 'f(x)': 355.3479307760119, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25236706, 0.26492046]), 'df/dx': array([-1.
↪08324132e-07, -1.77591726e-07]), '|df/dx|': 2.0802148601417215e-07, 'radius': 3.
↪9624822873892604}
{'x': array([ 65.5478516 , 420.96873867]), 'f(x)': 355.34793077604337, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.26492044, 0.25236705]), 'df/dx': array([-3.
↪57506205e-06, -1.94052802e-06]), '|df/dx|': 4.0677656828632715e-06, 'radius': 3.
↪962482353630508}
{'x': array([420.96874631, -25.87741685]), 'f(x)': 394.89995250449795, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25236706, 0.28660987]), 'df/dx': array([-6.
↪88793362e-05,  1.28562205e-05]), '|df/dx|': 7.006886155216386e-05, 'radius': 3.
↪962482290875642}
{'x': array([-25.87741831, 420.96874798]), 'f(x)': 394.89995250449834, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.28660987, 0.25236706]), 'df/dx': array([-2.
↪75807492e-07,  4.09007625e-07]), '|df/dx|': 4.933122843795251e-07, 'radius': 3.
↪962482268214163}
{'x': array([-302.5249348 , -124.82935385]), 'f(x)': 414.5450738280892, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25328926, 0.25791668]), 'df/dx': array([2.
↪04869882e-07, 6.62012064e-07]), '|df/dx|': 6.929874752492491e-07, 'radius': 3.
↪94805527688158}
{'x': array([-124.82934494, -302.52495752]), 'f(x)': 414.545073828166, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25791666, 0.25328928]), 'df/dx': array([ 2.
↪96155717e-06, -5.54891390e-06]), '|df/dx|': 6.289774748183183e-06, 'radius': 3.
↪948054982693164}
{'x': array([  5.23919934, 420.96874633]), 'f(x)': 415.03761110228186, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.40385527, 0.25236706]), 'df/dx': array([ 1.
↪69728841e-08, -6.53743171e-09]), '|df/dx|': 1.8188370128610815e-08, 'radius': 3.
↪96248228564607}
{'x': array([203.81444034, 203.81418418]), 'f(x)': 434.27936424136533, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25487089, 0.25487075]), 'df/dx': array([ 4.
↪78382028e-05, -1.74512966e-05]), '|df/dx|': 5.092191471661888e-05, 'radius': 3.
↪9235573469420832}
{'x': array([-302.52493401,  65.54782194]), 'f(x)': 473.7862653906966, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25328926, 0.2649204 ]), 'df/dx': array([ 4.
↪04967791e-07, -1.14309055e-05]), '|df/dx|': 1.1438076783196258e-05, 'radius': 3.
↪948055287264701 5}
```

(continues on next page)

(continued from previous page)

```
{'x': array([  65.54784372, -302.52489368]), 'f(x)': 473.7862653907328, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.26492043, 0.25328926]), 'df/dx': array([-5.
↪66054650e-06,  1.06210921e-05]), '|df/dx|': 1.2035338988579099e-05, 'radius': 3.
↪9480553149530255}
{'x': array([-124.82934378,  203.81425277]), 'f(x)': 513.2464086042439, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25791667, 0.25487078]), 'df/dx': array([3.
↪26121394e-06, 3.13624282e-08]), '|df/dx|': 3.2613647414422305e-06, 'radius': 3.
↪9235568530091007}
{'x': array([ 203.81425394, -124.82936922]), 'f(x)': 513.2464086042446, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25487078, 0.25791669]), 'df/dx': array([ 3.
↪28795789e-07, -3.30084884e-06]), '|df/dx|': 3.3171840071152417e-06, 'radius': 3.
↪923556834208886}
{'x': array([-302.52493667,  -25.87742463]), 'f(x)': 513.3382871189435, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.25328926, 0.28660989]), 'df/dx': array([-2.
↪67481062e-07, -2.08783586e-06]), '|df/dx|': 2.1049001648845596e-06, 'radius': 3.
↪948055250923777}
{'x': array([ -25.87738751, -302.52492037]), 'f(x)': 513.3382871190928, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.28660977, 0.25328925]), 'df/dx': array([8.
↪55158995e-06, 3.85988854e-06]), '|df/dx|': 9.382346729080099e-06, 'radius': 3.
↪9480554603167324}
{'x': array([   5.23919384, -302.52492805]), 'f(x)': 533.475945716733, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.40385528, 0.25328925]), 'df/dx': array([-2.
↪20542113e-06,  1.91451026e-06]), '|df/dx|': 2.920484874800241e-06, 'radius': 3.
↪948055361677073}
{'x': array([ 65.54778404, 203.81423688]), 'f(x)': 572.4876001674866, 'classifier':
↪'minimum', 'Hessian eigvals': array([0.26492033, 0.25487077]), 'df/dx': array([-2.
↪14712665e-05, -4.01908979e-06]), '|df/dx|': 2.1844183808925003e-05, 'radius': 3.
↪92355695384662}
```

```
res = a.kill_client()
```

```
2023-09-29 22:50:35,008 - distributed.worker.state_machine - WARNING - Async instruction␣
↪for <Task cancelled name='execute(local_method-35a52325a8f0b8cbafffe74244dc15e6)' coro=
↪<Worker.execute() done, defined at /home/docs/checkouts/readthedocs.org/user_builds/
↪hgdl/envs/stable/lib/python3.9/site-packages/distributed/worker_state_machine.py:3609>>
↪ ended with CancelledError
2023-09-29 22:50:35,008 - distributed.worker.state_machine - WARNING - Async instruction␣
↪for <Task cancelled name='execute(hgdl-b6d02f5aace9cb386f1fb2d05149742e)' coro=<Worker.
↪execute() done, defined at /home/docs/checkouts/readthedocs.org/user_builds/hgdl/envs/
↪stable/lib/python3.9/site-packages/distributed/worker_state_machine.py:3609>> ended␣
↪with CancelledError
```

3.4 Making a Plot

You should see the constraints and the foudn optima. If everything worked, the found points are in between the two constraints.

```
data = [np.append(entry["x"],entry["f(x)"]) for entry in res]
make_plot(data = np.array(data), constraint1 = g1, constraint2 = g2)
```

CHAPTER

FOUR

HGDL

4.1 HGDL

Welcome to the documentation of the HGDL API. HGDL is an optimization algorithm specialized in finding not only one but a diverse set of optima, alleviating challenges of non-uniqueness that are common in modern applications such as inversion problems and training of machine learning models.

HGDL is customized for distributed HP computing; all workers can be distributed across as many nodes or cores. All local optimizations will then be executed in parallel. As solutions are found, they are deflated which effectively removes those optima from the function, so that they cannot be reidentified by subsequent local searches. For more information please have a look at the content below.

4.2 See Also

- Recent Paper
- HGDN
- gpCAM
- fvGP

INDEX