active_subspaces Documentation

Release 0.1

Paul Constantine

October 03, 2016

Contents

Active subspaces are part of an emerging set of tools for discovering low-dimensional structure in a given function of several variables. Interesting applications arise in deterministic computer simulations of complex physical systems, where the function is the map from the physical model's input parameters to its output quantity of interest. The active subspace is the span of particular directions in the input parameter space; perturbing the inputs along these *active* directions changes the output more, on average, than perturbing the inputs orthogonally to the active directions. By focusing on the model's response along active directions and ignoring the relatively inactive directions, we *reduce the dimension* for parameter studies—such as optimization and integration—that are essential to engineering tasks such as design and uncertainty quantification.

For more information on active subspaces, visit http://activesubspaces.org/ or purchase the book Active Subspaces: Emerging Ideas in Dimension Reduction for Parameter Studies published by SIAM.

This library contains Python tools for discovering and exploiting a given model's active subspace. The user may provide a function handle to a complex model or its gradient with respect to the input parameters. Alternatively, the user may provide a set of input/output pairs from a previously executed set of runs (e.g., a Monte Carlo or Latin hypercube study).

CHAPTER 1

Installation

The github repository is https://github.com/paulcon/active_subspaces.git

To install the active subspaces package, open the terminal/command line and clone the repository with the command

```
git clone https://github.com/paulcon/active_subspaces.git
```

Navigate into the `active_subspaces` folder (where the `setup.py` file is located) and run the command

```
python setup.py install
```

You should now be able to import the active subspaces library in Python scripts and interpreters with the command `import active_subspaces`.

CHAPTER 2

Examples

The `tutorials` directory contains several Jupyter notebooks with examples of the code usage. You may also visit the Active Subspaces Data Sets repository for examples of applying active subspaces to real science and engineering models.

For a quickstart, consider a bivariate quadratic function

```
import numpy as np

def fun(x):
    A = np.array([[4., 2.], [2., 1.1]])
    return 0.5*np.dot(x.ravel(), np.dot(A, x.ravel()))
```

with gradient function

```
def dfun(x):
    A = np.array([[4., 2.], [2., 1.1]])
    return np.dot(A, x.ravel()).reshape((2, 1))
```

Draw 50 samples from the function's domain, assumed to be the [-1,1]^2 box equipped with a uniform probability density function,

```
X = np.random.uniform(-1., 1., size=(50, 2))
```

For each sample, compute the function and its gradient using the `SimulationRunner` and `SimulationGradientRunner` classes.

```
import active_subspaces as acs

# evaluate the function
sr = acs.utils.simrunners.SimulationRunner(fun)
f = sr.run(X)

# evaluate the gradient
sgr = acs.utils.simrunners.SimulationGradientRunner(dfun)
df = sgr.run(X)
```

Compute the active subspace with the gradients.

```
ss = acs.subspaces.Subspaces()
ss.compute(df=df, sstype='AS')
```

See the documentation for `Subspaces.compute()` for more details. Use the plotting routines to examine the estimated eigenvalues.

```
acs.utils.plotters.eigenvalues(ss.eigenvals)
```

Make a one-dimensional summary plot

```
y = np.dot(X, ss.W1)
acs.utils.plotters.sufficient_summary(y, f)
```

To exploit the one-dimensional active subspace, first set up the active variable domain and the map between the active variables and the full variables,

```
# set up the active variable domain
avd = acs.domains.BoundedActiveVariableDomain(ss)

# set up the maps between active and full variables
avm = acs.domains.BoundedActiveVariableMap(avd)
```

To estimate an integral,

```
N = 10 # number of active variable quadrature points
mu = acs.integrals.integrate(fun, avm, N)[0]
print 'Estimated integral: {:6.4f}'.format(mu)
```

To train and test a low-dimensional response surface,

```
rs = acs.response_surfaces.ActiveSubspaceResponseSurface(avm)

# train with the interface
N = 10 # number of active variable training points
rs.train_with_interface(fun, N)

# or train with the existing runs
rs.train_with_data(X, f)

# test
XX = np.random.uniform(-1., 1., size=(100, 2))
fXX = rs.predict(XX)
```

Explore the documentation and Jupyter notebooks to see the code's full range of capabilities.

CHAPTER 3

Developer documentation

3.1 API

3.1.1 Domains

Utilities for building the domains and maps for active variables.

class active_subspaces.domains.**ActiveVariableDomain**
A base class for the domain of functions of active variables.

subspaces
Subspaces

subspaces that define the domain

m
int

the dimension of the simulation inputs

n
int

the dimension of the active subspace

vertY
ndarray

n-dimensional vertices that define the boundary of the domain when the m-dimensional space is a hypercube

vertX
ndarray

corners of the m-dimensional hypercube that map to the points *vertY*

convhull
scipy.spatial.ConvexHull

the ConvexHull object defined by the vertices *vertY*

constraints
dict

a dictionary of linear inequality constraints conforming to the specifications used in the scipy.optimizer library

Notes

Attributes *vertY*, *vertX*, *convhull*, and *constraints* are None when the m-dimensional parameter space is unbounded.

class active_subspaces.domains.**ActiveVariableMap**(*domain*)
A base class for the map between active/inactive and original variables.

domain
ActiveVariableDomain

an ActiveVariableDomain object

See also:

domains.UnboundedActiveVariableMap, domains.BoundedActiveVariableMap

forward(*X*)
Map full variables to active variables.

Map the points in the original input space to the active and inactive variables.

Parameters **X** (*ndarray*) – an M-by-m matrix, each row of *X* is a point in the original parameter space

Returns

- **Y** (*ndarray*) – M-by-n matrix that contains points in the space of active variables. Each row of *Y* corresponds to a row of *X*.
- **Z** (*ndarray*) – M-by-(m-n) matrix that contains points in the space of inactive variables. Each row of *Z* corresponds to a row of *X*.

inverse(*Y*, *N=1*)
Find points in full space that map to active variable points.

Map the points in the active variable space to the original parameter space.

Parameters

- **Y** (*ndarray*) – M-by-n matrix that contains points in the space of active variables
- **N** (*int, optional*) – the number of points in the original parameter space that are returned that map to the given active variables (default 1)

Returns

- **X** (*ndarray*) – (M*N)-by-m matrix that contains points in the original parameter space
- **ind** (*ndarray*) – (M*N)-by-1 matrix that contains integer indices. These indices identify which rows of *X* map to which rows of *Y*.

Notes

The inverse map depends critically on the *regularize_z* function.

regularize_z(*Y*, *N*)
Pick inactive variables associated active variables.

Find points in the space of inactive variables to complete the inverse map.

Parameters

- **Y** (*ndarray*) – M-by-n matrix that contains points in the space of active variables

- **N** (*int*) – The number of points in the original parameter space that are returned that map to the given active variables

Returns **Z** – (M)-by-(m-n)-by-N matrix that contains values of the inactive variables

Return type ndarray

Notes

The base class does not implement *regularize_z*. Specific implementations depend on whether the original variables are bounded or unbounded. They also depend on what the weight function is on the original parameter space.

class active_subspaces.domains.**BoundedActiveVariableDomain**(*subspaces*)
Domain of functions with bounded domains (uniform on hypercube).

An class for the domain of functions of active variables when the space of simulation parameters is bounded.

Notes

Using this class assumes that the space of simulation inputs is equipped with a uniform weight function. And the space itself is a hypercube.

compute_boundary()
Compute and set the boundary of the domain.

Notes

This function computes the boundary of the active variable range, i.e., the domain of a function of the active variables, and it sets the attributes to the computed components. It is called when the BoundedActiveVariableDomain is initialized. If the dimension of the active subspaces is manually changed, then this function must be called again to recompute the boundary of the domain.

class active_subspaces.domains.**BoundedActiveVariableMap**(*domain*)
Class for mapping between active and bounded full variables.

A class for the map between active/inactive and original variables when the original variables are bounded by a hypercube with a uniform density.

See also:

domains.UnboundedActiveVariableMap

regularize_z(*Y*, *N*)
Pick inactive variables associated active variables.

Find points in the space of inactive variables to complete the inverse map.

Parameters

- **Y** (*ndarray*) – M-by-n matrix that contains points in the space of active variables
- **N** (*int*) – The number of points in the original parameter space that are returned that map to the given active variables

Returns **Z** – (M)-by-(m-n)-by-N matrix that contains values of the inactive variables

Return type ndarray

Notes

This implementation of *regularize_z* uses the function *sample_z* to randomly sample values of the inactive variables to complement the given values of the active variables.

class active_subspaces.domains.**UnboundedActiveVariableDomain**(*subspaces*)
Domain of functions with unbounded domains (Gaussian weight).

An class for the domain of functions of active variables when the space of simulation parameters is unbounded.

Notes

Using this class assumes that the space of simulation inputs is equipped with a Gaussian weight function.

class active_subspaces.domains.**UnboundedActiveVariableMap**(*domain*)
Class for mapping between active and unbounded full variables.

A class for the map between active/inactive and original variables when the original variables are ubbounded and the space is equipped with a standard Gaussian density.

See also:

domains.BoundedActiveVariableMap

regularize_z(*Y*, *N*)
Pick inactive variables associated active variables.

Find points in the space of inactive variables to complete the inverse map.

Parameters

- **Y** (*ndarray*) – M-by-n matrix that contains points in the space of active variables
- **N** (*int*) – The number of points in the original parameter space that are returned that map to the given active variables

Returns **Z** – (M)-by-(m-n)-by-N matrix that contains values of the inactive variables

Return type ndarray

Notes

This implementation of *regularize_z* samples the inactive variables from a standard (m-n)-variate Gaussian distribution.

active_subspaces.domains.**hit_and_run_z**(*N*, *y*, *W1*, *W2*)
A hit and run method for sampling the inactive variables from a polytope.

Parameters

- **N** (*int*) – the number of inactive variable samples
- **y** (*ndarray*) – the value of the active variables
- **W1** (*ndarray*) – m-by-n matrix that contains the eigenvector bases of the n-dimensional active subspace
- **W2** (*ndarray*) – m-by-(m-n) matrix that contains the eigenvector bases of the (m-n)-dimensional inactive subspace

Returns **Z** – N-by-(m-n) matrix that contains values of the inactive variable that correspond to the given *y*

Return type ndarray

See also:

`domains.sample_z()`

Notes

The interface for this implementation is written specifically for *domains.sample_z*.

`active_subspaces.domains.`**`interval_endpoints`**`(W1)`
Compute the range of a 1d active variable.

Parameters **W1** (*ndarray*) – m-by-1 matrix that contains the eigenvector that defines the first active variable

Returns

- **Y** (*ndarray*) – 2-by-1 matrix that contains the endpoints of the interval defining the range of the 1d active variable
- **X** (*ndarray*) – 2-by-m matrix that contains the corners of the m-dimensional hypercube that map to the active variable endpoints

`active_subspaces.domains.`**`nzv`**`(m, n)`
Number of zonotope vertices.

Compute the number of zonotope vertices for a linear map from R^m to R^n.

Parameters

- **m** (*int*) – the dimension of the hypercube
- **n** (*int*) – the dimension of the low-dimesional subspace

Returns **N** – the number of vertices defining the zonotope

Return type int

`active_subspaces.domains.`**`random_walk_z`**`(N, y, W1, W2)`
A random walk method for sampling from a polytope.

Parameters

- **N** (*int*) – the number of inactive variable samples
- **y** (*ndarray*) – the value of the active variables
- **W1** (*ndarray*) – m-by-n matrix that contains the eigenvector bases of the n-dimensional active subspace
- **W2** (*ndarray*) – m-by-(m-n) matrix that contains the eigenvector bases of the (m-n)-dimensional inactive subspace

Returns **Z** – N-by-(m-n) matrix that contains values of the inactive variable that correspond to the given *y*

Return type ndarray

See also:

`domains.sample_z()`

Notes

The interface for this implementation is written specifically for *domains.sample_z*.

active_subspaces.domains.**rejection_sampling_z**(*N*, *y*, *W1*, *W2*)
A rejection sampling method for sampling the from a polytope.

Parameters

- **N** (*int*) – the number of inactive variable samples
- **y** (*ndarray*) – the value of the active variables
- **W1** (*ndarray*) – m-by-n matrix that contains the eigenvector bases of the n-dimensional active subspace
- **W2** (*ndarray*) – m-by-(m-n) matrix that contains the eigenvector bases of the (m-n)-dimensional inactive subspace

Returns **Z** – N-by-(m-n) matrix that contains values of the inactive variable that correspond to the given *y*

Return type ndarray

See also:

domains.sample_z()

Notes

The interface for this implementation is written specifically for *domains.sample_z*.

active_subspaces.domains.**sample_z**(*N*, *y*, *W1*, *W2*)
Sample inactive variables.

Sample values of the inactive variables for a fixed value of the active variables when the original variables are bounded by a hypercube.

Parameters

- **N** (*int*) – the number of inactive variable samples
- **y** (*ndarray*) – the value of the active variables
- **W1** (*ndarray*) – m-by-n matrix that contains the eigenvector bases of the n-dimensional active subspace
- **W2** (*ndarray*) – m-by-(m-n) matrix that contains the eigenvector bases of the (m-n)-dimensional inactive subspace

Returns **Z** – N-by-(m-n) matrix that contains values of the inactive variable that correspond to the given *y*

Return type ndarray

Notes

The trick here is to sample the inactive variables z so that -1 <= W1*y + W2*z <= 1, where y is the given value of the active variables. In other words, we need to sample z such that it respects the linear equalities W2*z <= 1 - W1*y, -W2*z <= 1 + W1*y. These inequalities define a polytope in R^(m-n). We want to sample *N* points uniformly from the polytope.

This function first tries a simple rejection sampling scheme, which (i) finds a bounding hyperbox for the polytope, (ii) draws points uniformly from the bounding hyperbox, and (iii) rejects points outside the polytope.

If that method does not return enough samples, the method tries a "hit and run" method for sampling from the polytope.

If that doesn't work, it returns an array with *N* copies of a feasible point computed as the Chebyshev center of the polytope. Thanks to David Gleich for showing me Chebyshev centers.

active_subspaces.domains.**unique_rows**(*S*)
Return the unique rows from ndarray

Parameters **S** (*ndarray*) – array with rows to reduces

Returns **T** – version of *S* with unique rows

Return type ndarray

Notes

http://stackoverflow.com/questions/16970982/find-unique-rows-in-numpy-array

active_subspaces.domains.**zonotope_vertices**(*W1*, *Nsamples=10000*, *maxcount=100000*)
Compute the vertices of the zonotope.

Parameters

- **W1** (*ndarray*) – m-by-n matrix that contains the eigenvector bases of the n-dimensional active subspace
- **Nsamples** (*int, optional*) – number of samples per iteration to check (default 1e4)
- **maxcount** (*int, optional*) – maximum number of iterations (default 1e5)

Returns

- **Y** (*ndarray*) – nzv-by-n matrix that contains the zonotope vertices
- **X** (*ndarray*) – nzv-by-m matrix that contains the corners of the m-dimensional hypercube that map to the zonotope vertices

3.1.2 Gradients

Utilities for approximating gradients.

active_subspaces.gradients.**finite_difference_gradients**(*X*, *fun*, *h=1e-06*)
Compute finite difference gradients with a given interface.

Parameters

- **X** (*ndarray*) – M-by-m matrix that contains the points to estimate the gradients with finite differences
- **fun** (*function*) – function that returns the simulation's quantity of interest given inputs
- **h** (*float, optional*) – the finite difference step size (default 1e-6)

Returns **df** – M-by-m matrix that contains estimated partial derivatives approximated by finite differences

Return type ndarray

active_subspaces.gradients.**local_linear_gradients**(*X*, *f*, *p=None*, *weights=None*)
Estimate a collection of gradients from input/output pairs.

Given a set of input/output pairs, choose subsets of neighboring points and build a local linear model for each subset. The gradients of these local linear models comprise estimates of sampled gradients.

Parameters

- **X** (*ndarray*) – M-by-m matrix that contains the m-dimensional inputs
- **f** (*ndarray*) – M-by-1 matrix that contains scalar outputs
- **p** (*int, optional*) – how many nearest neighbors to use when constructing the local linear model (default 1)
- **weights** (*ndarray, optional*) – M-by-1 matrix that contains the weights for each observation (default None)

Returns **df** – M-by-m matrix that contains estimated partial derivatives approximated by the local linear models

Return type ndarray

Notes

If *p* is not specified, the default value is floor(1.7*m).

3.1.3 Integrals

Utilities for exploiting active subspaces when estimating integrals.

active_subspaces.integrals.**av_integrate**(*avfun*, *avmap*, *N*)
Approximate the integral of a function of active variables.

Parameters

- **avfun** (*function*) – a function of the active variables
- **avmap** (ActiveVariableMap) – a domains.ActiveVariableMap
- **N** (*int*) – the number of points in the quadrature rule

Returns **mu** – an estimate of the integral

Return type float

Notes

This function is usually used when one has already constructed a response surface on the active variables and wants to estimate its integral.

active_subspaces.integrals.**av_quadrature_rule**(*avmap*, *N*)
Get a quadrature rule on the space of active variables.

Parameters

- **avmap** (ActiveVariableMap) – a domains.ActiveVariableMap
- **N** (*int*) – the number of quadrature nodes in the active variables

Returns

- **Yp** (*ndarray*) – quadrature nodes on the active variables
- **Yw** (*ndarray*) – quadrature weights on the active variables

See also:

`integrals.quadrature_rule()`

`active_subspaces.integrals.`**`integrate`**(*fun*, *avmap*, *N*, *NMC=10*)

Approximate the integral of a function of m variables.

Parameters

- **`fun`** (*`function`*) – an interface to the simulation that returns the quantity of interest given inputs as an 1-by-m ndarray
- **`avmap`** (`ActiveVariableMap`) – a domains.ActiveVariableMap
- **`N`** (*`int`*) – the number of points in the quadrature rule
- **`NMC`** (*`int, optional`*) – the number of points in the Monte Carlo estimates of the conditional expectation and conditional variance (default 10)

Returns

- **mu** (*float*) – an estimate of the integral of the function computed against the weight function on the simulation inputs
- **lb** (*float*) – a central-limit-theorem 95% lower confidence from the Monte Carlo part of the integration
- **ub** (*float*) – a central-limit-theorem 95% upper confidence from the Monte Carlo part of the integration

See also:

`integrals.quadrature_rule()`

Notes

The CLT-based bounds *lb* and *ub* are likely poor estimators of the error. They only account for the variance from the Monte Carlo portion. They do not include any error from the integration rule on the active variables.

`active_subspaces.integrals.`**`interval_quadrature_rule`**(*avmap*, *N*, *NX=10000*)

Quadrature rule on a one-dimensional interval.

Quadrature when the dimension of the active subspace is 1 and the simulation parameter space is bounded.

Parameters

- **`avmap`** (`ActiveVariableMap`) – a domains.ActiveVariableMap
- **`N`** (*`int`*) – the number of quadrature nodes in the active variables
- **`NX`** (*`int, optional`*) – the number of samples to use to estimate the quadrature weights (default 10000)

Returns

- **Yp** (*ndarray*) – quadrature nodes on the active variables
- **Yw** (*ndarray*) – quadrature weights on the active variables

See also:

`integrals.quadrature_rule()`

active_subspaces.integrals.**quadrature_rule**(*avmap*, *N*, *NMC=10*)

Get a quadrature rule on the space of simulation inputs.

Parameters

- **avmap** (ActiveVariableMap) – a domains.ActiveVariableMap
- **N** (*int*) – the number of quadrature nodes in the active variables
- **NMC** (*int, optional*) – the number of samples in the simple Monte Carlo over the inactive variables (default 10)

Returns

- **Xp** (*ndarray*) – (N*NMC)-by-m matrix containing the quadrature nodes on the simulation input space
- **Xw** (*ndarray*) – (N*NMC)-by-1 matrix containing the quadrature weights on the simulation input space
- **ind** (*ndarray*) – array of indices identifies which rows of *Xp* correspond to the same fixed value of the active variables

See also:

integrals.av_quadrature_rule()

Notes

This quadrature rule uses an integration rule on the active variables and simple Monte Carlo on the inactive variables.

If the simulation inputs are bounded, then the quadrature nodes on the active variables is constructed with a Delaunay triangulation of a maximin design. The weights are computed by sampling the original variables, mapping them to the active variables, and determining which triangle the active variables fall in. These samples are used to estimate quadrature weights. Note that when the dimension of the active subspace is one-dimensional, this reduces to operations on an interval.

If the simulation inputs are unbounded, the quadrature rule on the active variables is given by a tensor product Gauss-Hermite quadrature rule.

active_subspaces.integrals.**zonotope_quadrature_rule**(*avmap*, *N*, *NX=10000*)

Quadrature rule on a zonotope.

Quadrature when the dimension of the active subspace is greater than 1 and the simulation parameter space is bounded.

Parameters

- **avmap** (ActiveVariableMap) – a domains.ActiveVariableMap
- **N** (*int*) – the number of quadrature nodes in the active variables
- **NX** (*int, optional*) – the number of samples to use to estimate the quadrature weights (default 10000)

Returns

- **Yp** (*ndarray*) – quadrature nodes on the active variables
- **Yw** (*ndarray*) – quadrature weights on the active variables

See also:

integrals.quadrature_rule()

3.1.4 Optimizers

Utilities for exploiting active subspaces when optimizing.

class active_subspaces.optimizers.**BoundedMinVariableMap**(*domain*)
This subclass is a MinVariableMap for bounded simulation inputs.

See also:

optimizers.MinVariableMap, optimizers.UnboundedMinVariableMap

regularize_z(*Y*, *N=1*)
Train the global quadratic for the regularization.

Parameters

- **Y** (*ndarray*) – N-by-n matrix of points in the space of active variables
- **N** (*int, optional*) – merely there satisfy the interface of *regularize_z*. It should not be anything other than 1

Returns **Z** – N-by-(m-n)-by-1 matrix that contains a value of the inactive variables for each value of the inactive variables

Return type ndarray

Notes

In contrast to the *regularize_z* in BoundedActiveVariableMap and UnboundedActiveVariableMap, this implementation of *regularize_z* uses a quadratic program to find a single value of the inactive variables for each value of the active variables.

class active_subspaces.optimizers.**MinVariableMap**(*domain*)
ActiveVariableMap for optimization

This subclass is an domains.ActiveVariableMap specifically for optimization.

See also:

optimizers.BoundedMinVariableMap, optimizers.UnboundedMinVariableMap

Notes

This class's train function fits a global quadratic surrogate model to the n+2 active variables—two more than the dimension of the active subspace. This quadratic surrogate is used to map points in the space of active variables back to the simulation parameter space for minimization.

train(*X*, *f*)
Train the global quadratic for the regularization.

Parameters

- **X** (*ndarray*) – input points used to train a global quadratic used in the *regularize_z* function
- **f** (*ndarray*) – simulation outputs used to train a global quadratic in the *regularize_z* function

class active_subspaces.optimizers.**UnboundedMinVariableMap**(*domain*)
This subclass is a MinVariableMap for unbounded simulation inputs.

See also:

optimizers.MinVariableMap, optimizers.BoundedMinVariableMap

regularize_z(*Y*, *N=1*)

Train the global quadratic for the regularization.

Parameters

- **Y** (*ndarray*) – N-by-n matrix of points in the space of active variables
- **N** (*int, optional*) – merely there satisfy the interface of *regularize_z*. It should not be anything other than 1

Returns **Z** – N-by-(m-n)-by-1 matrix that contains a value of the inactive variables for each value of the inactive variables

Return type ndarray

Notes

In contrast to the *regularize_z* in BoundedActiveVariableMap and UnboundedActiveVariableMap, this implementation of *regularize_z* uses a quadratic program to find a single value of the inactive variables for each value of the active variables.

active_subspaces.optimizers.**av_minimize**(*avfun*, *avdom*, *avdfun=None*)

Minimize a response surface on the active variables.

Parameters

- **avfun** (*function*) – a function of the active variables
- **avdom** (ActiveVariableDomain) – information about the domain of *avfun*
- **avdfun** (*function*) – returns the gradient of *avfun*

Returns

- **ystar** (*ndarray*) – the estimated minimizer of *avfun*
- **fstar** (*float*) – the estimated minimum of *avfun*

See also:

optimizers.interval_minimize(), optimizers.zonotope_minimize(), optimizers.unbounded_minimize()

active_subspaces.optimizers.**interval_minimize**(*avfun*, *avdom*)

Minimize a response surface defined on an interval.

Parameters

- **avfun** (*function*) – a function of the active variables
- **avdom** (ActiveVariableDomain) – contains information about the domain of *avfun*

Returns

- **ystar** (*ndarray*) – the estimated minimizer of *avfun*
- **fstar** (*float*) – the estimated minimum of *avfun*

See also:

optimizers.av_minimize()

Notes

This function wraps the scipy.optimize function fminbound.

active_subspaces.optimizers.**minimize**(*asrs*, *X*, *f*)

Minimize a response surface constructed with the active subspace.

Parameters

- **asrs** (ActiveSubspaceResponseSurface) – a trained response_surfaces.ActiveSubspaceResponseSurface
- **X** (*ndarray*) – input points used to train the MinVariableMap
- **f** (*ndarray*) – simulation outputs used to train the MinVariableMap

Returns

- **xstar** (*ndarray*) – the estimated minimizer of the function modeled by the ActiveSubspaceResponseSurface *asrs*
- **fstar** (*float*) – the estimated minimum of the function modeled by *asrs*

Notes

This function has two stages. First it uses the scipy.optimize package to minimize the response surface of the active variables. Then it trains a MinVariableMap with the given input/output pairs, which it uses to map the minimizer back to the space of simulation inputs.

This is very heuristic.

active_subspaces.optimizers.**unbounded_minimize**(*avfun*, *avdom*, *avdfun*)

Minimize a response surface defined on an unbounded domain.

Parameters

- **avfun** (*function*) – a function of the active variables
- **avdom** (ActiveVariableDomain) – contains information about the domain of *avfun*
- **avdfun** (*function*) – returns the gradient of *avfun*

Returns

- **ystar** (*ndarray*) – the estimated minimizer of *avfun*
- **fstar** (*float*) – the estimated minimum of *avfun*

See also:

optimizers.av_minimize()

Notes

If the gradient *avdfun* is None, this function wraps the scipy.optimize implementation of SLSQP. Otherwise, it wraps BFGS.

active_subspaces.optimizers.**zonotope_minimize**(*avfun*, *avdom*, *avdfun*)

Minimize a response surface defined on a zonotope.

Parameters

- **avfun** (*function*) – a function of the active variables

- **avdom** (ActiveVariableDomain) – contains information about the domain of *avfun*
- **avdfun** (*function*) – returns the gradient of *avfun*

Returns

- **ystar** (*ndarray*) – the estimated minimizer of *avfun*
- **fstar** (*float*) – the estimated minimum of *avfun*

See also:

`optimizers.av_minimize()`

Notes

This function wraps the scipy.optimize implementation of SLSQP with linear inequality constraints derived from the zonotope.

3.1.5 Response Surfaces

Utilities for exploiting active subspaces in response surfaces.

class `active_subspaces.response_surfaces.`**ActiveSubspaceResponseSurface**(*avmap*, *respsurf=None*)

A class for using active subspace with response surfaces.

respsurf
ResponseSurface

respsurf is a utils.response_surfaces.ResponseSurface

avmap
ActiveVariableMap

a domains.ActiveVariableMap

Notes

This class has several convenient functions for training and using a response surface with active subspaces. Note that the *avmap* must be given. This means that the active subspace must be computed already.

gradient(*X*)
Gradient of the response surface.

A convenience function for computing the gradient of the response surface with respect to the simulation inputs.

Parameters **X** (*ndarray*) – M-by-m matrix containing points in the space of simulation inputs

Returns **df** – contains the response surface gradient at the given *X*

Return type ndarray

gradient_av(*Y*)
Compute the gradient with respect to the active variables.

A convenience function for computing the gradient of the response surface with respect to the active variables.

Parameters **Y** (*ndarray*) – M-by-n matrix containing points in the range of active variables to evaluate the response surface gradient

Returns **df** – contains the response surface gradient at the given *Y*

Return type ndarray

predict(*X*, *compgrad=False*)

Evaluate the response surface at full space points.

Compute the value of the response surface given values of the simulation variables.

Parameters

- **X** (*ndarray*) – M-by-m matrix containing points in simulation's parameter space
- **compgrad** (*bool, optional*) – determines if the gradient of the response surface is computed and returned (default False)

Returns

- **f** (*ndarray*) – contains the response surface values at the given *X*
- **dfdx** (*ndarray*) – an ndarray of shape M-by-m that contains the estimated gradient at the given *X*. If *compgrad* is False, then *dfdx* is None.

predict_av(*Y*, *compgrad=False*)

Evaluate response surface at active variable.

Compute the value of the response surface given values of the active variables.

Parameters

- **Y** (*ndarray*) – M-by-n matrix containing points in the range of active variables to evaluate the response surface
- **compgrad** (*bool, optional*) – determines if the gradient of the response surface with respect to the active variables is computed and returned (default False)

Returns

- **f** (*ndarray*) – contains the response surface values at the given *Y*
- **df** (*ndarray*) – contains the response surface gradients at the given *Y*. If *compgrad* is False, then *df* is None.

train_with_data(*X*, *f*, *v=None*)

Train the response surface with input/output pairs.

Parameters

- **X** (*ndarray*) – M-by-m matrix with evaluations of the simulation inputs
- **f** (*ndarray*) – M-by-1 matrix with corresponding simulation quantities of interest
- **v** (*ndarray, optional*) – M-by-1 matrix that contains the regularization (i.e., errors) associated with *f* (default None)

Notes

The training methods exploit the eigenvalues from the active subspace analysis to determine length scales for each variable when tuning the parameters of the radial bases.

The method sets attributes of the object for further use.

train_with_interface (*fun*, *N*, *NMC=10*)
Train the response surface with input/output pairs.

Parameters

- **fun** (*function*) – a function that returns the simulation quantity of interest given a point in the input space as an 1-by-m ndarray
- **N** (*int*) – the number of points used in the design-of-experiments for constructing the response surface
- **NMC** (*int, optional*) – the number of points used to estimate the conditional expectation and conditional variance of the function given a value of the active variables

Notes

The training methods exploit the eigenvalues from the active subspace analysis to determine length scales for each variable when tuning the parameters of the radial bases.

The method sets attributes of the object for further use.

The method uses the response_surfaces.av_design function to get the design for the appropriate *avmap*.

active_subspaces.response_surfaces.**av_design** (*avmap*, *N*, *NMC=10*)
Design on active variable space.

A wrapper that returns the design for the response surface in the space of the active variables.

Parameters

- **avmap** (ActiveVariableMap) – a domains.ActiveVariable map that includes the active variable domain, which includes the active and inactive subspaces
- **N** (*int*) – the number of points used in the design-of-experiments for constructing the response surface
- **NMC** (*int, optional*) – the number of points used to estimate the conditional expectation and conditional variance of the function given a value of the active variables (Default is 10)

Returns

- **Y** (*ndarray*) – N-by-n matrix that contains the design points in the space of active variables
- **X** (*ndarray*) – (N*NMC)-by-m matrix that contains points in the simulation input space to run the simulation
- **ind** (*ndarray*) – indices that map points in *X* to points in *Y*

See also:

utils.designs.gauss_hermite_design(), utils.designs.interval_design(), utils.designs.maximin_design()

3.1.6 Subspaces

Utilities for computing active and inactive subspaces.

class active_subspaces.subspaces.**Subspaces**
A class for computing active and inactive subspaces.

eigenvals
ndarray

m-by-1 matrix of eigenvalues

eigenvecs
ndarray

m-by-m matrix, eigenvectors oriented column-wise

W1
ndarray

m-by-n matrix, basis for the active subspace

W2
ndarray

m-by-(m-n) matrix, basis for the inactive subspace

e_br
ndarray

m-by-2 matrix, bootstrap ranges for the eigenvalues

sub_br
ndarray

m-by-3 matrix, bootstrap ranges (first and third column) and the mean (second column) of the error in the estimated active subspace approximated by bootstrap

Notes

The attributes *W1* and *W2* are convenience variables. They are identical to the first n and last (m-n) columns of *eigenvecs*, respectively.

compute (*X=None*, *f=None*, *df=None*, *weights=None*, *sstype='AS'*, *ptype='EVG'*, *nboot=0*)
Compute the active and inactive subspaces.

Given input points and corresponding outputs, or given samples of the gradients, estimate an active subspace. This method has four different algorithms for estimating the active subspace: 'AS' is the standard active subspace that requires gradients, 'OLS' uses a global linear model to estimate a one-dimensional active subspace, 'QPHD' uses a global quadratic model to estimate subspaces, and 'OPG' uses a set of local linear models computed from subsets of give input/output pairs.

The function also sets the dimension of the active subspace (and, consequently, the dimenison of the inactive subspace). There are three heuristic choices for the dimension of the active subspace. The default is the largest gap in the eigenvalue spectrum, which is 'EVG'. The other two choices are 'RS', which estimates the error in a low-dimensional response surface using the eigenvalues and the estimated subspace errors, and 'LI' which is a heuristic from Bing Li on order determination.

Note that either *df* or *X* and *f* must be given, although formally all are optional.

Parameters

- **X** (*ndarray, optional*) – M-by-m matrix of samples of inputs points, arranged as rows (default None)
- **f** (*ndarray, optional*) – M-by-1 matrix of outputs corresponding to rows of *X* (default None)
- **df** (*ndarray, optional*) – M-by-m matrix of samples of gradients, arranged as rows (default None)

- **weights** (*ndarray, optional*) – M-by-1 matrix of weights associated with rows of *X*
- **sstype** (*str, optional*) – defines subspace type to compute. Default is 'AS' for active subspace, which requires *df*. Other options are *OLS* for a global linear model, *QPHD* for a global quadratic model, and *OPG* for local linear models. The latter three require *X* and *f*.
- **ptype** (*str, optional*) – defines the partition type. Default is 'EVG' for largest eigenvalue gap. Other options are 'RS', which is an estimate of the response surface error, and 'LI', which is a heuristic proposed by Bing Li based on subspace errors and eigenvalue decay.
- **nboot** (*int, optional*) – number of bootstrap samples used to estimate the error in the estimated subspace (default 0 means no bootstrap estimates)

Notes

Partition type 'RS' and 'LI' require nboot to be greater than 0 (and probably something more like 100) to get bootstrap estimates of the subspace error.

partition(*n*)

Partition the eigenvectors to define the active subspace.

A convenience function for partitioning the full set of eigenvectors to separate the active from inactive subspaces.

Parameters **n** (*int*) – the dimension of the active subspace

active_subspaces.subspaces.**active_subspace**(*df*, *weights*)

Compute the active subspace.

Parameters

- **df** (*ndarray*) – M-by-m matrix containing the gradient samples oriented as rows
- **weights** (*ndarray*) – M-by-1 weight vector, corresponds to numerical quadrature rule used to estimate matrix whose eigenspaces define the active subspace

Returns

- **e** (*ndarray*) – m-by-1 vector of eigenvalues
- **W** (*ndarray*) – m-by-m orthogonal matrix of eigenvectors

active_subspaces.subspaces.**eig_partition**(*e*)

Partition the active subspace according to largest eigenvalue gap.

Parameters **e** (*ndarray*) – m-by-1 vector of eigenvalues

Returns

- **n** (*int*) – dimension of active subspace
- **ediff** (*float*) – largest eigenvalue gap

active_subspaces.subspaces.**errbnd_partition**(*e*, *sub_err*)

Partition the active subspace according to response surface error.

Uses an a priori estimate of the response surface error based on the eigenvalues and subspace error to determine the active subspace dimension.

Parameters

- **e** (*ndarray*) – m-by-1 vector of eigenvalues
- **sub_err** (*ndarray*) – m-by-1 vector of estimates of subspace error

Returns

- **n** (*int*) – dimension of active subspace
- **errbnd** (*float*) – estimate of error bound

Notes

The error bound should not be used as an estimated error. The bound is only used to estimate the subspace dimension.

active_subspaces.subspaces.**ladle_partition**(*e*, *li_F*)
Partition the active subspace according to Li's criterion.

Uses a criterion proposed by Bing Li that combines estimates of the subspace with estimates of the eigenvalues.

Parameters

- **e** (*ndarray*) – m-by-1 vector of eigenvalues
- **li_F** (*float*) – measures error in the subspace

Returns

- **n** (*int*) – dimension of active subspace
- **G** (*ndarray*) – metrics used to determine active subspace dimension

active_subspaces.subspaces.**ols_subspace**(*X*, *f*, *weights*)
Estimate one-dimensional subspace with global linear model.

Parameters

- **X** (*ndarray*) – M-by-m matrix of input samples, oriented as rows
- **f** (*ndarray*) – M-by-1 vector of output samples corresponding to the rows of *X*
- **weights** (*ndarray*) – M-by-1 weight vector, corresponds to numerical quadrature rule used to estimate matrix whose eigenspaces define the active subspace

Returns

- **e** (*ndarray*) – m-by-1 vector of eigenvalues
- **W** (*ndarray*) – m-by-m orthogonal matrix of eigenvectors

Notes

Although the method returns a full set of eigenpairs (to be consistent with the other subspace functions), only the first eigenvalue will be nonzero, and only the first eigenvector will have any relationship to the input parameters. The remaining m-1 eigenvectors are only orthogonal to the first.

active_subspaces.subspaces.**opg_subspace**(*X*, *f*, *weights*)
Estimate active subspace with local linear models.

This approach is related to the sufficient dimension reduction method known sometimes as the outer product of gradient method. See the 2001 paper 'Structure adaptive approach for dimension reduction' from Hristache, et al.

Parameters

- **X** (*ndarray*) – M-by-m matrix of input samples, oriented as rows
- **f** (*ndarray*) – M-by-1 vector of output samples corresponding to the rows of X
- **weights** (*ndarray*) – M-by-1 weight vector, corresponds to numerical quadrature rule used to estimate matrix whose eigenspaces define the active subspace

Returns

- **e** (*ndarray*) – m-by-1 vector of eigenvalues
- **W** (*ndarray*) – m-by-m orthogonal matrix of eigenvectors

active_subspaces.subspaces.**qphd_subspace**(*X*, *f*, *weights*)

Estimate active subspace with global quadratic model.

This approach is similar to Ker-Chau Li's approach for principal Hessian directions based on a global quadratic model of the data. In contrast to Li's approach, this method uses the average outer product of the gradient of the quadratic model, as opposed to just its Hessian.

Parameters

- **X** (*ndarray*) – M-by-m matrix of input samples, oriented as rows
- **f** (*ndarray*) – M-by-1 vector of output samples corresponding to the rows of X
- **weights** (*ndarray*) – M-by-1 weight vector, corresponds to numerical quadrature rule used to estimate matrix whose eigenspaces define the active subspace

Returns

- **e** (*ndarray*) – m-by-1 vector of eigenvalues
- **W** (*ndarray*) – m-by-m orthogonal matrix of eigenvectors

active_subspaces.subspaces.**sorted_eigh**(*C*)

Compute eigenpairs and sort.

Parameters **C** (*ndarray*) – matrix whose eigenpairs you want

Returns

- **e** (*ndarray*) – vector of sorted eigenvalues
- **W** (*ndarray*) – orthogonal matrix of corresponding eigenvectors

Notes

Eigenvectors are unique up to a sign. We make the choice to normalize the eigenvectors so that the first component of each eigenvector is positive. This normalization is very helpful for the bootstrapping.

3.1.7 Utils

Designs

Utilities for constructing design-of-experiments.

active_subspaces.utils.designs.**gauss_hermite_design**(*N*)

Tensor product Gauss-Hermite quadrature points.

Parameters **N** (*int[]*) – contains the number of points per dimension in the tensor product design

Returns **design** – N-by-m matrix that contains the design points

Return type ndarray

active_subspaces.utils.designs.**interval_design**(*a*, *b*, *N*)

Equally spaced points on an interval.

Parameters

- **a** (*float*) – the left endpoint of the interval
- **b** (*float*) – the right endpoint of the interval
- **N** (*int*) – the number of points in the design

Returns N-by-1 matrix that contains the design points in the interval. It does not contain the endpoints.

Return type design, ndarray

active_subspaces.utils.designs.**maximin_design**(*vert*, *N*)

Multivariate maximin design constrained by a polytope.

Parameters

- **vert** (*ndarray*) – the vertices that define the m-dimensional polytope. The shape of *vert* is M-by-m, where M is the number of vertices.
- **N** (*int*) – the number of points in the design

Returns **design** – N-by-m matrix that contains the design points in the polytope. It does not contain the vertices.

Return type ndarray

Notes

The objective function used to find the design is the negative of the minimum distance between points in the design and the given vertices. The routine uses the scipy.minimize function with the SLSQP method to minimize the function. The constraints are given by the polytope defined by the vertices. The scipy.spatial packages turns the vertices into a set of linear inequality constraints.

The optimization is nonlinear and nonconvex with many local minima. Any reasonable local minima is likely to give a good design. However, to increase robustness, we use three random starting points in the minimization and use the design with the lowest objective value.

Misc

Miscellaneous utilities.

class active_subspaces.utils.misc.**BoundedNormalizer**(*lb*, *ub*)

A class for normalizing bounded inputs.

lb

ndarray

a matrix of size m-by-1 that contains lower bounds on the simulation inputs

ub

ndarray

a matrix of size m-by-1 that contains upper bounds on the simulation inputs

See also:

`utils.misc.UnboundedNormalizer`

normalize(*X*)
Return corresponding points shifted and scaled to [-1,1]^m.

Parameters **X** (*ndarray*) – contains all input points one wishes to normalize. The shape of *X* is M-by-m. The components of each row of *X* should be between *lb* and *ub*.

Returns **X_norm** – contains the normalized inputs corresponding to *X*. The components of each row of *X_norm* should be between -1 and 1.

Return type ndarray

unnormalize(*X*)
Return corresponding points shifted and scaled to *[lb, ub]*.

Parameters **X** (*ndarray*) – contains all input points one wishes to unnormalize. The shape of *X* is M-by-m. The components of each row of *X* should be between -1 and 1.

Returns **X_unnorm** – contains the unnormalized inputs corresponding to *X*. The components of each row of *X_unnorm* should be between *lb* and *ub*.

Return type ndarray

class active_subspaces.utils.misc.**Normalizer**
An abstract class for normalizing inputs.

normalize(*X*)
Return corresponding points in normalized domain.

Parameters **X** (*ndarray*) – contains all input points one wishes to normalize

Returns **X_norm** – contains the normalized inputs corresponding to *X*

Return type ndarray

Notes

Points in *X* should be oriented as an m-by-n ndarray, where each row corresponds to an m-dimensional point in the problem domain.

unnormalize(*X*)
Return corresponding points shifted and scaled to [-1,1]^m.

Parameters **X** (*ndarray*) – contains all input points one wishes to unnormalize

Returns **X_unnorm** – contains the unnormalized inputs corresponding to *X*

Return type ndarray

Notes

Points in *X* should be oriented as an m-by-n ndarray, where each row corresponds to an m-dimensional point in the normalized domain.

class active_subspaces.utils.misc.**UnboundedNormalizer**(*mu*, *C*)
A class for normalizing unbounded, Gaussian inputs to standard normals.

mu

ndarray

a matrix of size m-by-1 that contains the mean of the Gaussian simulation inputs

L

ndarray

a matrix size m-by-m that contains the Cholesky factor of the covariance matrix of the Gaussian simulation inputs.

See also:

`utils.misc.BoundedNormalizer`

Notes

A simulation with unbounded inputs is assumed to have a Gaussian weight function associated with the inputs. The covariance of the Gaussian weight function should be full rank.

normalize(*X*)

Return points transformed to a standard normal distribution.

Parameters **X** (*ndarray*) – contains all input points one wishes to normalize. The shape of *X* is M-by-m. The components of each row of *X* should be a draw from a Gaussian with mean *mu* and covariance *C*.

Returns **X_norm** – contains the normalized inputs corresponding to *X*. The components of each row of *X_norm* should be draws from a standard multivariate normal distribution.

Return type ndarray

unnormalize(*X*)

Transform points to original Gaussian.

Return corresponding points transformed to draws from a Gaussian distribution with mean *mu* and covariance *C*.

Parameters **X** (*ndarray*) – contains all input points one wishes to unnormalize. The shape of *X* is M-by-m. The components of each row of *X* should be draws from a standard multivariate normal.

Returns **X_unnorm** – contains the unnormalized inputs corresponding to *X*. The components of each row of *X_unnorm* should represent draws from a multivariate normal with mean *mu* and covariance *C*.

Return type ndarray

active_subspaces.utils.misc.**atleast_2d**(*A*, *oned_as='row'*)

Ensures the array *A* is at least two dimensions.

Parameters

- **A** (*ndarray*) – matrix
- **oned_as** (*str, optional*) – should be either 'row' or 'col'. It determines whether the array *A* should be expanded as a 2d row or 2d column (default 'row')

active_subspaces.utils.misc.**atleast_2d_col**(*A*)

Wrapper for *atleast_2d(A, 'col')*

Notes

Thanks to Trent Lukaczyk for these functions!

active_subspaces.utils.misc.**atleast_2d_row**(*A*)
Wrapper for *atleast_2d(A, 'row')*

Notes

Thanks to Trent Lukaczyk for these functions!

active_subspaces.utils.misc.**conditional_expectations**(*f*, *ind*)
Compute conditional expectations and variances for given function values.

Parameters

- **f** (*ndarray*) – an ndarry of function evaluations
- **ind** (*ndarray[int]*) – index array that tells which values of f correspond to the same value for the active variable.

Returns

- **Ef** (*ndarray*) – an ndarray containing the conditional expectations
- **Vf** (*ndarray*) – an ndarray containing the conditional variances

Notes

This function computes the mean and variance for all values in the ndarray f that have the same index in *ind*. The indices in *ind* correspond to values of the active variables.

active_subspaces.utils.misc.**process_inputs**(*X*)
Check a matrix of input values for the right shape.

Parameters **X** (*ndarray*) – contains input points. The shape of X should be M-by-m.

Returns

- **X** (*ndarray*) – the same as the input
- **M** (*int*) – number of rows in X
- **m** (*int*) – number of columns in X

active_subspaces.utils.misc.**process_inputs_outputs**(*X*, *f*)
Check matrix of input values and a vector of outputs for correct shapes.

Parameters

- **X** (*ndarray*) – contains input points. The shape of X should be M-by-m.
- **f** (*ndarray*) – M-by-1 matrix

Returns

- **X** (*ndarray*) – the same as the input
- **f** (*ndarray*) – the same as the output
- **M** (*int*) – number of rows in X
- **m** (*int*) – number of columns in X

Plotters

Utilities for plotting quantities computed in active subspaces.

active_subspaces.utils.plotters.**eigenvalues**(*e*, *e_br=None*, *out_label=None*, *opts=None*)
Plot the eigenvalues with bootstrap ranges.

Parameters

- **e** (*ndarray*) – k-by-1 matrix that contains the estimated eigenvalues
- **e_br** (*ndarray, optional*) – lower and upper bounds for the estimated eigenvalues. These are typically computed with a bootstrap. (default None)
- **out_label** (*str, optional*) – a label for the quantity of interest (default None)
- **opts** (*dict, optional*) – a dictionary with some plot options (default None)

See also:

utils.plotters.eigenvectors(), utils.plotters.subspace_errors()

active_subspaces.utils.plotters.**eigenvectors**(*W*, *W_br=None*, *in_labels=None*, *out_label=None*, *opts=None*)
Plot the estimated eigenvectors with optional bootstrap ranges.

Parameters

- **W** (*ndarray*) – m-by-k matrix that contains k of the estimated eigenvectors from the active subspace analysis.
- **W_br** (*ndarray, optional*) – m-by-(2*k) matrix that contains estimated upper and lower bounds on the components of the eigenvectors (default None)
- **in_labels** (*str[], optional*) – list of labels for the simulation's inputs (default None)
- **out_label** (*str, optional*) – a label for the quantity of interest (default None)
- **opts** (*dict, optional*) – a dictionary with some plot options (default None)

See also:

utils.plotters.subspace_errors(), utils.plotters.eigenvalues()

Notes

This function will plot at most the first four eigevectors in a four-subplot figure. In other words, it only looks at the first four columns of *W*.

active_subspaces.utils.plotters.**plot_opts**(*savefigs=True*, *figtype='.eps'*)
A few options for the plots.

Parameters

- **savefigs** (*bool*) – save figures into a separate figs director
- **figtype** (*str*) – a file extention for the type of image to save

Returns **opts** – the chosen options. The keys in the dictionary are *figtype*, *savefigs*, and *font*. The *font* is a dictionary that sets the font properties of the figures.

Return type dict

active_subspaces.utils.plotters.**subspace_errors**(*sub_br*, *out_label=None*, *opts=None*)
Plot the estimated subspace errors with bootstrap ranges.

Parameters

- **sub_br** (*ndarray*) – (k-1)-by-3 matix that contains the lower bound, mean, and upper bound of the subspace errors for each dimension of subspace.
- **out_label** (*str, optional*) – a label for the quantity of interest (default None)
- **opts** (*dict, optional*) – a dictionary with some plot options (default None)

See also:

utils.plotters.eigenvectors(),utils.plotters.eigenvalues()

active_subspaces.utils.plotters.**sufficient_summary**(*y*, *f*, *out_label=None*, *opts=None*)
Make a summary plot with the given predictors and responses.

Parameters

- **y** (*ndarray*) – M-by-1 or M-by-2 matrix that contains the values of the predictors for the summary plot.
- **f** (*ndarray*) – M-by-1 matrix that contains the corresponding responses
- **out_label** (*str, optional*) – a label for the quantity of interest (default None)
- **opts** (*dict, optional*) – a dictionary with some plot options (default None)

Notes

If *y.shape[1]* is 1, then this function produces only the univariate summary plot. If *y.shape[1]* is 2, then this function produces both the univariate and the bivariate summary plot, where the latter is a scatter plot with the first column of *y* on the horizontal axis, the second column of *y* on the vertical axis, and the color corresponding to *f*.

active_subspaces.utils.plotters.**zonotope_2d_plot**(*vertices*, *design=None*, *y=None*, *f=None*, *out_label=None*, *opts=None*)
A utility for plotting (m,2) zonotopes with designs and quadrature rules.

Parameters

- **vertices** (*ndarray*) – M-by-2 matrix that contains the vertices that define the zonotope
- **design** (*ndarray, optional*) – N-by-2 matrix that contains a design-of-experiments on the zonotope. The plot will contain the Delaunay triangulation of the points in *design* and *vertices*. (default None)
- **y** (*ndarray, optional*) – K-by-2 matrix that contains points to be plotted inside the zonotope. If *y* is given, then *f* must be given, too. (default None)
- **f** (*ndarray, optional*) – K-by-1 matrix that contains a color value for the associated points in *y*. This is useful for plotting function values or quadrature rules with the zonotope. If *f* is given, then *y* must be given, too. (default None)
- **out_label** (*str, optional*) – a label for the quantity of interest (default None)
- **opts** (*dict, optional*) – a dictionary with some plot options (default None)

Notes

This function makes use of the scipy.spatial routines for plotting the zonotopes.

QP Solver

Solvers for the linear and quadratic programs in active subspaces.

class active_subspaces.utils.qp_solver.**QPSolver**(*solver='GUROBI'*)
A class for solving linear and quadratic programs.

solver
str

identifies which linear program software to use

Notes

The class checks to see if Gurobi is present. If it is, it uses Gurobi to solve the linear and quadratic programs. Otherwise, it uses scipy implementations to solve the linear and quadratic programs.

linear_program_eq(*c*, *A*, *b*, *lb*, *ub*)
Solves an equality constrained linear program with variable bounds.

This method returns the minimizer of the following linear program.

minimize c^T x subject to A x = b lb <= x <= ub

Parameters

- **c** (*ndarray*) – m-by-1 matrix for the linear objective function
- **A** (*ndarray*) – M-by-m matrix that contains the coefficients of the linear equality constraints
- **b** (*ndarray*) – M-by-1 matrix that is the right hand side of the equality constraints
- **lb** (*ndarray*) – m-by-1 matrix that contains the lower bounds on the variables
- **ub** (*ndarray*) – m-by-1 matrix that contains the upper bounds on the variables

Returns **x** – m-by-1 matrix that is the minimizer of the linear program

Return type ndarray

linear_program_ineq(*c*, *A*, *b*)
Solves an inequality constrained linear program.

This method returns the minimizer of the following linear program.

minimize c^T x subject to A x >= b

Parameters

- **c** (*ndarray*) – m-by-1 matrix for the linear objective function
- **A** (*ndarray*) – M-by-m matrix that contains the coefficients of the linear equality constraints
- **b** (*ndarray*) – size M-by-1 matrix that is the right hand side of the equality constraints

Returns **x** – m-by-1 matrix that is the minimizer of the linear program

Return type ndarray

quadratic_program_bnd(*c*, *Q*, *lb*, *ub*)
Solves a quadratic program with variable bounds.

This method returns the minimizer of the following linear program.

minimize c^T x + x^T Q x subject to lb <= x <= ub

Parameters

- **c** (*ndarray*) – m-by-1 matrix that contains the coefficients of the linear term in the objective function
- **Q** (*ndarray*) – m-by-m matrix that contains the coefficients of the quadratic term in the objective function
- **lb** (*ndarray*) – m-by-1 matrix that contains the lower bounds on the variables
- **ub** (*ndarray*) – m-by-1 matrix that contains the upper bounds on the variables

Returns **x** – m-by-1 matrix that is the minimizer of the quadratic program

Return type ndarray

quadratic_program_ineq(*c*, *Q*, *A*, *b*)
Solves an inequality constrained quadratic program.

This method returns the minimizer of the following quadratic program.

minimize c^T x + x^T Q x subject to A x >= b

Parameters

- **c** (*ndarray*) – m-by-1 matrix that contains the coefficients of the linear term in the objective function
- **Q** (*ndarray*) – m-by-m matrix that contains the coefficients of the quadratic term in the objective function
- **A** (*ndarray*) – M-by-m matrix that contains the coefficients of the linear equality constraints
- **b** (*ndarray*) – M-by-1 matrix that is the right hand side of the equality constraints

Returns **x** – m-by-1 matrix that is the minimizer of the quadratic program.

Return type ndarray

Quadrature

Gaussian quadrature utilities for use with the Python Active-subspaces Utility Library.

active_subspaces.utils.quadrature.**gld**(*N*, *quadtype*)
One-dimensional Gaussian quadrature rule.

Parameters

- **N** (*int*) – number of nodes in the quadrature rule
- **quadtype** (*str*) – type of quadrature rule {'Legendre', 'Hermite'}

Returns

- **x** (*ndarray*) – N-by-1 array of quadrature nodes
- **w** (*ndarray*) – N-by-1 array of quadrature weights

See also:

`utils.quadrature.gauss_hermite()`

Notes

This computation is inspired by Walter Gautschi's code at https://www.cs.purdue.edu/archives/2002/wxg/codes/OPQ.html.

`active_subspaces.utils.quadrature.`**`gauss_hermite`**`(N)`
Tensor product Gauss-Hermite quadrature rule.

Parameters **N** (*int[]*) – number of nodes in each dimension of the quadrature rule

Returns

- **x** (*ndarray*) – N-by-1 array of quadrature nodes
- **w** (*ndarray*) – N-by-1 array of quadrature weights

Notes

This computation is inspired by Walter Gautschi's code at https://www.cs.purdue.edu/archives/2002/wxg/codes/OPQ.html.

`active_subspaces.utils.quadrature.`**`gauss_legendre`**`(N)`
Tensor product Gauss-Legendre quadrature rule.

Parameters **N** (*int[]*) – number of nodes in each dimension of the quadrature rule

Returns

- **x** (*ndarray*) – N-by-1 array of quadrature nodes
- **w** (*ndarray*) – N-by-1 array of quadrature weights

Notes

This computation is inspired by Walter Gautschi's code at https://www.cs.purdue.edu/archives/2002/wxg/codes/OPQ.html.

`active_subspaces.utils.quadrature.`**`gh1d`**`(N)`
One-dimensional Gauss-Hermite quadrature rule.

Parameters **N** (*int*) – number of nodes in the quadrature rule

Returns

- **x** (*ndarray*) – N-by-1 array of quadrature nodes
- **w** (*ndarray*) – N-by-1 array of quadrature weights

See also:

`utils.quadrature.gauss_hermite()`

Notes

This computation is inspired by Walter Gautschi's code at https://www.cs.purdue.edu/archives/2002/wxg/codes/OPQ.html.

`active_subspaces.utils.quadrature.`**`gl1d`**`(N)`
One-dimensional Gauss-Legendre quadrature rule.

Parameters **N** (*int*) – number of nodes in the quadrature rule

Returns

- **x** (*ndarray*) – N-by-1 array of quadrature nodes
- **w** (*ndarray*) – N-by-1 array of quadrature weights

See also:

`utils.quadrature.gauss_legendre()`

Notes

This computation is inspired by Walter Gautschi's code at https://www.cs.purdue.edu/archives/2002/wxg/codes/OPQ.html.

`active_subspaces.utils.quadrature.`**`jacobi_matrix`**(*ab*)
Tri-diagonal Jacobi matrix of recurrence coefficients.

Parameters **ab** (*ndarray*) – N-by-2 array of recurrence coefficients

Returns **J** – (N-1)-by-(N-1) symmetric, tridiagonal Jacobi matrix associated with the orthogonal polynomials

Return type ndarray

See also:

`utils.quadrature.r_hermite()`, `utils.quadrature.gauss_hermite()`

Notes

This computation is inspired by Walter Gautschi's code at https://www.cs.purdue.edu/archives/2002/wxg/codes/OPQ.html.

`active_subspaces.utils.quadrature.`**`r_hermite`**(*N*)
Recurrence coefficients for the Hermite orthogonal polynomials.

Parameters **N** (*int*) – the number of recurrence coefficients

Returns **ab** – an *N*-by-2 array of the recurrence coefficients

Return type ndarray

See also:

`utils.quadrature.jacobi_matrix()`, `utils.quadrature.gauss_hermite()`

Notes

This computation is inspired by Walter Gautschi's code at https://www.cs.purdue.edu/archives/2002/wxg/codes/OPQ.html.

`active_subspaces.utils.quadrature.`**`r_jacobi`**(*N*, *l*, *r*, *a*, *b*)
Recurrence coefficients for the Legendre orthogonal polynomials.

Parameters

- **N** (*int*) – the number of recurrence coefficients
- **l** (*float*) – the left endpoint of the interval
- **r** (*float*) – the right endpoint of the interval
- **a** (*float*) – Jacobi weight parameter

- **b** (*float*) – Jacobi weight parameter

Returns **ab** – an *N*-by-2 array of the recurrence coefficients

Return type ndarray

See also:

`utils.quadrature.jacobi_matrix()`, `utils.quadrature.gauss_legendre()`

Notes

This computation is inspired by Walter Gautschi's code at https://www.cs.purdue.edu/archives/2002/wxg/codes/OPQ.html.

Response Surfaces (Utils)

Utilities for building response surface approximations.

class `active_subspaces.utils.response_surfaces.`**PolynomialApproximation**(*N=2*)

Least-squares-fit, global, multivariate polynomial approximation.

poly_weights

ndarray

an ndarray of coefficients for the polynomial approximation in the monomial basis

g

ndarray

contains the m coefficients corresponding to the degree 1 monomials in the polynomial approximation

H

ndarray

an ndarray of shape m-by-m that contains the coefficients of the degree 2 monomials in the approximation

See also:

`utils.response_surfaces.RadialBasisApproximation`

Notes

All attributes besides the degree *N* are set when the class's *train* method is called.

predict(*X*, *compgrad=False*)

Evaluate least-squares-fit polynomial approximation at new points.

Parameters

- **X** (*ndarray*) – an ndarray of points to evaluate the polynomial approximation. The shape is M-by-m, where m is the number of dimensions.
- **compgrad** (*bool, optional*) – a flag to decide whether or not to compute the gradient of the polynomial approximation at the points *X*. (default False)

Returns

- **f** (*ndarray*) – an ndarray of predictions from the polynomial approximation. The shape of *f* is M-by-1.

- **df** (*ndarray*) – an ndarray of gradient predictions from the polynomial approximation. The shape of *df* is M-by-m.

train (*X*, *f*, *weights=None*)
Train the least-squares-fit polynomial approximation.

Parameters

- **X** (*ndarray*) – an ndarray of training points for the polynomial approximation. The shape is M-by-m, where m is the number of dimensions.
- **f** (*ndarray*) – an ndarray of function values used to train the polynomial approximation. The shape of *f* is M-by-1.
- **weights** (*ndarray, optional*) – an ndarray of weights for the least-squares. (default is None, which means uniform weights)

Notes

This method sets all the attributes of the class for use in the *predict* method.

class active_subspaces.utils.response_surfaces.**RadialBasisApproximation**(*N=2*)
Approximate a multivariate function with a radial basis.

A class for global, multivariate radial basis approximation with anisotropic squared-exponential radial basis and a weighted-least-squares-fit monomial basis.

radial_weights
ndarray

an ndarray of coefficients radial basis functions in the model

poly_weights
poly_weights

an ndarray of coefficients for the polynomial approximation in the monomial basis

K
ndarray

an ndarray of shape M-by-M that contains the matrix of radial basis functions evaluated at the training points

ell
ndarray

an ndarray of shape m-by-1 that contains the characteristic length scales along each of the inputs

See also:

utils.response_surfaces.PolynomialApproximation

Notes

All attributes besides the degree *N* are set when the class's *train* method is called.

predict (*X*, *compgrad=False*)
Evaluate the radial basis approximation at new points.

Parameters

- **X** (*ndarray*) – an ndarray of points to evaluate the polynomial approximation. The shape is M-by-m, where m is the number of dimensions.
- **compgrad** (*bool, optional*) – a flag to decide whether or not to compute the gradient of the polynomial approximation at the points *X*. (default False)

Returns

- **f** (*ndarray*) – an ndarray of predictions from the polynomial approximation. The shape of *f* is M-by-1.
- **df** (*ndarray*) – an ndarray of gradient predictions from the polynomial approximation. The shape of *df* is M-by-m.

Notes

I'll tell you what. I just refactored this code to use terminology from radial basis functions instead of Gaussian processes, and I feel so much better about it. Now I don't have to compute that silly prediction variance and try to pretend that it has anything to do with the actual error in the approximation. Also, computing that variance requires another system solve, which might be expensive. So it's both expensive and of dubious value. So I got rid of it. Sorry, Gaussian processes.

train (*X*, *f*, *v=None*, *e=None*)
Train the radial basis approximation.

Parameters

- **X** (*ndarray*) – an ndarray of training points for the polynomial approximation. The shape is M-by-m, where m is the number of dimensions.
- **f** (*ndarray*) – an ndarray of function values used to train the polynomial approximation. The shape of *f* is M-by-1.
- **v** (*ndarray, optional*) – contains the regularization parameters that model error in the function values (default None)
- **e** (*ndarray, optional*) – an ndarray containing the eigenvalues from the active subspace analysis. If present, the radial basis uses it to determine the appropriate anisotropy in the length scales. (default None)

Notes

The approximation uses an multivariate, squared exponential radial basis. If *e* is not None, then the radial basis is anisotropic with length scales determined by *e*. Otherwise, the basis is isotropic. The length scale parameters (i.e., the rbf shape parameters) are determined with a maximum likelihood heuristic inspired by techniques for fitting a Gaussian process model.

The approximation also includes a monomial basis with monomials of total degree up to order *N*. These are fit with weighted least-squares, where the weight matrix is the inverse of the matrix of radial basis functions evaluated at the training points.

This method sets all the attributes of the class for use in the *predict* method.

class active_subspaces.utils.response_surfaces.**ResponseSurface** (*N=2*)
An abstract class for response surfaces.

N
int

maximum degree of global polynomial in the response surface

Rsqr
float

the R-squared coefficient for the response surface

X
ndarray

an ndarray of training points for the response surface. The shape is M-by-m, where m is the number of dimensions.

f
ndarray

an ndarray of function values used to train the response surface. The shape of *f* is M-by-1.

See also:

`utils.response_surfaces.PolynomialApproximation`, `utils.response_surfaces.RadialBasisAppr`

`active_subspaces.utils.response_surfaces.`**`exponential_squared`**(*X1*, *X2*, *sigma*, *ell*)
Compute the matrix of radial basis functions.

Parameters

- **X1** (*ndarray*) – contains the centers of the radial functions
- **X2** (*ndarray*) – the evaluation points of the radial functions
- **sigma** (*float*) – scales the radial functions
- **ell** (*ndarray*) – contains the length scales of each dimension

Returns **C** – the matrix of radial functions centered at *X1* and evaluated at *X2*. The shape of *C* is *X1.shape[0]*-by-*X2.shape[0]*.

Return type ndarray

`active_subspaces.utils.response_surfaces.`**`grad_exponential_squared`**(*X1*, *X2*, *sigma*, *ell*)
Compute the matrices of radial basis function gradients.

Parameters

- **X1** (*ndarray*) – contains the centers of the radial functions
- **X2** (*ndarray*) – the evaluation points of the radial functions
- **sigma** (*float*) – scales the radial functions
- **ell** (*ndarray*) – contains the length scales of each dimension

Returns **dC** – the matrix of radial function gradients centered at *X1* and evaluated at *X2*. The shape of *dC* is *X1.shape[0]*-by-*X2.shape[0]*-by-m. *dC* is a three-dimensional ndarray. The third dimension indexes the partial derivatives in each gradient.

Return type ndarray

`active_subspaces.utils.response_surfaces.`**`grad_polynomial_bases`**(*X*, *N*)
Compute the gradients of the monomial bases.

Parameters

- **X** (*ndarray*) – contains the points to evaluate the monomials
- **N** (*int*) – the maximum degree of the monomial basis

Returns **dB** – contains the gradients of the monomials evaluate at *X*. *dB* is a three-dimensional ndarray. The third dimension indexes the partial derivatives in each gradient.

Return type ndarray

`active_subspaces.utils.response_surfaces.`**`index_set`**`(n, d)`
Enumerate multi-indices for a total degree of order *n* in *d* variables.

Parameters

- **n** (*int*) – degree of polynomial
- **d** (*int*) – number of variables, dimension

Returns **I** – multi-indices ordered as columns

Return type ndarray

`active_subspaces.utils.response_surfaces.`**`polynomial_bases`**`(X, N)`
Compute the monomial bases.

Parameters

- **X** (*ndarray*) – contains the points to evaluate the monomials
- **N** (*int*) – the maximum degree of the monomial basis

Returns

- **B** (*ndarray*) – contains the monomial evaluations
- **I** (*ndarray*) – contains the multi-indices that tell the degree of each univariate monomial term in the multivariate monomial

Simrunners

Utilities for running several simulations at different inputs.

class `active_subspaces.utils.simrunners.`**`SimulationGradientRunner`**`(dfun)`
Evaluates gradients at several input values.

A class for running several simulations at different input values that return the gradients of the quantity of interest.

dfun
function

a function that runs the simulation for a fixed value of the input parameters, given as an ndarray. It returns the gradient of the quantity of interest at the given input.

See also:

`utils.simrunners.SimulationRunner`

Notes

The function dfun should take an ndarray of size 1-by-m and return an ndarray of shape 1-by-m. This ndarray is the gradient of the quantity of interest from the simulation. Often, the function is a wrapper to a larger simulation code.

run (*X*)
Run at several input values.

Run the simulation at several input values and return the gradients of the quantity of interest.

Parameters **X** (*ndarray*) – contains all input points where one wishes to run the simulation. If the simulation takes m inputs, then *X* must have shape M-by-m, where M is the number of simulations to run.

Returns **dF** – contains the gradient of the quantity of interest at each given input point. The shape of *dF* is M-by-m.

Return type ndarray

Notes

In principle, the simulation calls can be executed independently and in parallel. Right now this function uses a sequential for-loop. Future development will take advantage of multicore architectures to parallelize this for-loop.

class active_subspaces.utils.simrunners.**SimulationRunner** (*fun*)
A class for running several simulations at different input values.

fun
function

runs the simulation for a fixed value of the input parameters, given as an ndarray

See also:

utils.simrunners.SimulationGradientRunner

Notes

The function fun should take an ndarray of size 1-by-m and return a float. This float is the quantity of interest from the simulation. Often, the function is a wrapper to a larger simulation code.

run (*X*)
Run the simulation at several input values.

Parameters **X** (*ndarray*) – contains all input points where one wishes to run the simulation. If the simulation takes m inputs, then *X* must have shape M-by-m, where M is the number of simulations to run.

Returns **F** – contains the simulation output at each given input point. The shape of *F* is M-by-1.

Return type ndarray

Notes

In principle, the simulation calls can be executed independently and in parallel. Right now this function uses a sequential for-loop. Future development will take advantage of multicore architectures to parallelize this for-loop.

This is the init file.

3.2 Contact

Questions or comments? Send an email to Paul Constantine (paul.constantine@mines.edu)

3.3 LICENSE

CHAPTER 4

Indexes

- genindex
- modindex
- search

a

A

B

C

D

E

U

V

W

X

Z