TensorToolbox Documentation

Release 0.3.3

Daniele Bigoni

January 19, 2015

CHAPTER

ONE

INTRODUCTION

CHAPTER

TWO

TUTORIAL

2.1 Tensor Wrapper

The tensor wrapper is a data structure which mimics the behavior of a numpy.ndarray and associates each item of the tensor with the evaluation of a user-defined function on the corresponding grid point.

Let for example $\mathcal{X} = \times_{i=1}^{d} \mathbf{x}_i$, where $\mathbf{x}_i$ define the position of the grid points in the i-th direction. Let us consider the function $f : \mathcal{X} \to \mathbb{R}^{n_1 \times \ldots \times n_m}$. Let us define the tensor valued tensor $\mathcal{A} = f(\mathcal{X})$. Thus any entry $\mathcal{A}[i_1, \ldots, i_d] = f(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_d})$ is a tensor in $\mathbb{R}^{n_1 \times \ldots \times n_m}$. The storage of the whole tensor $\mathcal{A}$ can be problematic for big d and m, and not necessary if one is just willing to sample values from it.

The `TensorWrapper` allows the access to the elements of $\mathcal{A}$ which however are not all allocated, but computed on-the-fly and stored in a hash-table data structure (a Python dictionary). The `TensorWrapper` can be reshaped and accessed as if it was a numpy.ndarray (including slicing of indices). Additionally it allows the existence of multiple views of the tensor, sharing among them the allocated data, and it allows the *Quantics* folding used within the *Quantics Tensor Train* [2][1] routines `QTTvec`.

In the following we will use a simple example to show the capabilities of this data structure. We will let $d = 2$ and $f : \mathcal{X} \to \mathbb{R}$.

2.1.1 Construction

In order to **construct** a `TensorWrapper` we need first to define a grid and a function.

```
>>> import numpy as np
>>> import itertools
>>> import TensorToolbox as TT
>>> d = 2
>>> x_fine = np.linspace(-1,1,7)
>>> params = {'k': 1.}
>>> def f(X,params):
>>>     return np.max(X) * params['k']
>>> TW = TT.TensorWrapper( f, [ x_fine ]*d, params, dtype=float )
```

2.1.2 Access and data

The `TensorWrapper` can then be **accessed** as a numpy.ndarray:

```
>>> TW[1,2]
-0.33333333333333337
```

This access to the `TensorWrapper` has caused the evaluation of the function f and the storage of the associated value. In order to check the **fill level** of the `TensorWrapper`, we do:

```
>>> TW.get_fill_level()
1
```

The **evaluation indices** at which the function has been evaluated can be retrived this way:

```
>>> TW.get_fill_idxs()
[(1, 2)]
```

The `TensorWrapper` can be accessed using also **slicing** along some of the coordinates:

```
>>> TW[:,1:6:2]
array([[-0.6666666666666674, 0.0, 0.6666666666666652],
    [-0.6666666666666674, 0.0, 0.6666666666666652],
    [-0.3333333333333337, 0.0, 0.6666666666666652],
    [0.0, 0.0, 0.6666666666666652],
    [0.3333333333333326, 0.3333333333333326, 0.6666666666666652],
    [0.6666666666666652, 0.6666666666666652, 0.6666666666666652],
    [1.0, 1.0, 1.0]], dtype=object)
```

The **data** already computed are stored in the dictionary `TensorWrapper.data`, which one can access and modify at his/her own risk. The data can be **erased** just by resetting the `TensorWrapper.data` field:

```
>>> TW.data = {}
```

The constructed `TensorWrapper` to which has not been applied any of the view/extension/reshaping functions presented in the following, is called the **global** tensor wrapper. The shape informations regarding the global wrapper can be *always* accessed by:

```
>>> TW.get_global_shape()
(7, 7)
>>> TW.get_global_ndim()
2
>>> TW.get_global_size()
49
```

If no view/extension/reshaping has been applied to the `TensorWrapper`, then the same output is obtained by:

```
>>> TW.get_shape()
(7, 7)
>>> TW.get_ndim()
2
>>> TW.get_size()
49
```

or by

```
>>> TW.shape
(7, 7)
>>> TW.ndim
2
>>> TW.size
49
```

Note: If any view/extension/reshape has been applied to the `TensorWrapper`, then the output of `TensorWrapper.get_global_shape()` and `TensorWrapper.get_shape()` will differ. Anyway `TensorWrapper.get_global_shape()` will *always* return the information regarding the **global** tensor wrapper.

2.1.3 Views

The `TensorWrapper` allows the definition of multiple views over the defined tensor. The information regarding each view are contained in the dictionary `TensoWrapper.maps`. The main view is called `full` and is defined at construction time. Additional views can be defined through the function `TensorWrapper.set_view()`. Let's continue the previous example, by adding a new view to the wrapper with a coarser grid.

```
>>> x_coarse = np.linspace(-1,1,4)
>>> TW.set_view( 'coarse', [x_coarse]*d )
```

Note: The grid of the `full` view must contain the grids associated to the new view.

The different views can be accessed separately, but they all refer to the same global data structure. In order to access the `TensorWrapper` through one of its views, the view must be **activated**:

```
>>> TW.set_active_view('coarse')
>>> TW[2,:]
>>> TW.set_active_view('full')
>>> TW[1,:]
>>> TW[:,2]
```

The following figure shows the global grid as well as its two views, the `full` and the `coarse` views. The allocated indicies are also highlighted.

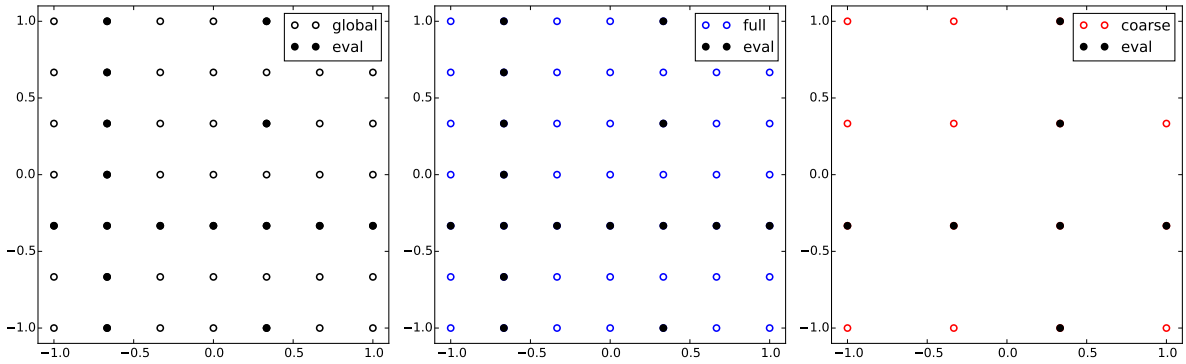

Figure 2.1: The global tensor and two of its views. The `full` view corresponds by default to the global tensor. The `coarse` is contained in the `full` view. The uniquely allocated values of the tensor are shown in the different views.

The shape characteristics of the active view can be accessed through `TensorWrapper.get_view_shape()` and the corresponding commands for `ndim` and `size`. For example:

```
>>> TW.set_active_view('full')
>>> TW.get_view_shape()
(7, 7)
>>> TW.get_shape()
(7, 7)
>>> TW.set_active_view('coarse')
>>> TW.get_global_shape()
(7, 7)
>>> TW.get_view_shape()
(4, 4)
>>> TW.get_shape()
(4, 4)
>>> TW.shape
(4, 4)
```

2.1.4 Grid refinement

The *global* grid can be refined using the function `TensorWrapper.refine()`, provinding a grid which contains the previous one. This refinement does not alter the allocated data which is instead preserved and mapped to the new mesh.

```
>>> x_ffine = np.linspace(-1,1,13)
>>> TW.refine([x_ffine]*d)
```

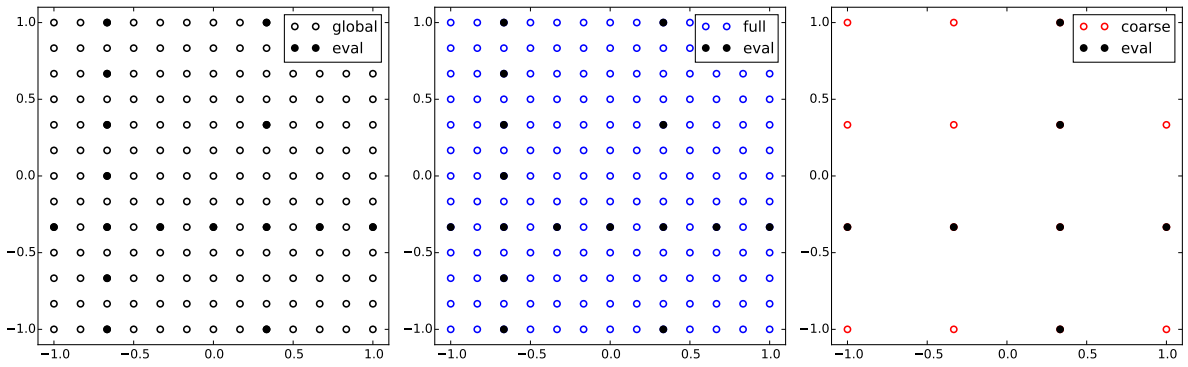

Figure 2.2: The global tensor and the two views defined, after the grid refinement.

2.1.5 Quantics extension

The quantics extension is used for extending the indices of the tesnor to the next power of `Q`. The extension is performed so that the last coordinate point is appended to the coordinate points the necessary number of times. In order to apply the extension on a particular view, one needs to activate the view and then use the method `TensorWrapper.set_Q()`.

```
>>> TW.set_active_view('full')
>>> TW.get_view_shape()
(13, 13)
>>> TW.get_extended_shape()
(13, 13)
>>> TW.set_Q(2)
>>> TW.get_extended_shape()
(16, 16)
>>> TW.get_shape()
(16, 16)
>>> TW.shape
(16, 16)
```

We can see that `TensorWrapper.get_extended_shape()` returns the same output of `TensorWrapper.get_viw_shape()` if no quantics extension has been applied.

Using the following code we can investigate the content of the extended tensor wrapper and plot it as shown in the following figure.

```
>>> A = TW[:,:]
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(6,5))
>>> plt.imshow(A,interpolation='none')
>>> plt.tight_layout()
>>> plt.show(False)
```

2.1.6 Reshape

The shape of each view can be changed as long as the size returned by `TensorWrapper.get_extended_size()` is unchanged. This means that if *no quantics* extension has been applied, the size must correspond to `TensorWrapper.get_view_size()`. If a *quantics* extension has been applied, the size must correspond to `TensorWrapper.get_extended_size()`.

For example let us reshape the *quantics* extended `full` view of the tensor to the shape (4,16).

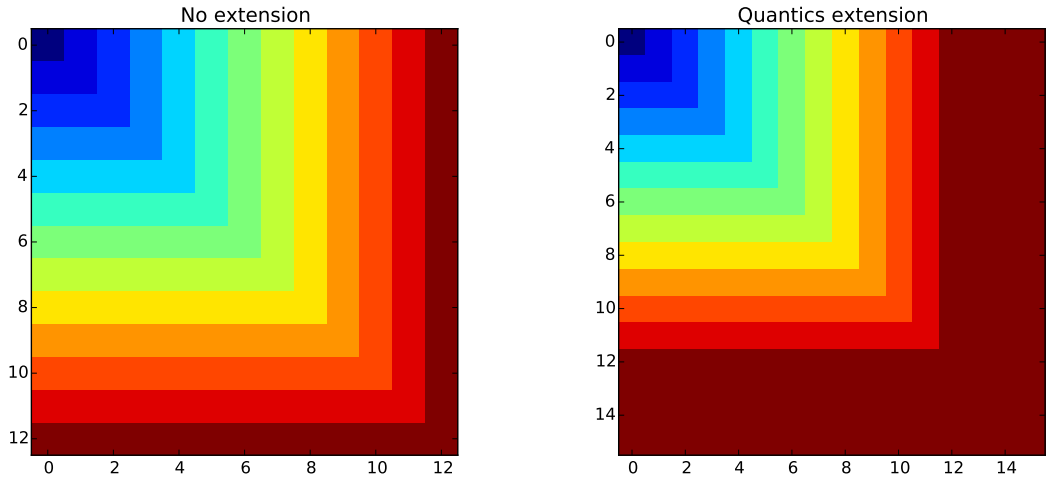

Figure 2.3: The *Quantics* extension applied to the `full` view results in the repetition of its limit values in the tensor grid.

```
>>> TW.set_active_view('full')
>>> TW.reshape((8,32))
>>> TW.get_extended_shape()
(16, 16)
>>> TW.get_shape()
(8, 32)
>>> TW.shape
(8, 32)
```

This results in the following reshaping of the tensor view:

```
>>> A = TW[:,:]
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(12,5))
>>> plt.imshow(A,interpolation='none')
>>> plt.tight_layout()
>>> plt.show(False)
```

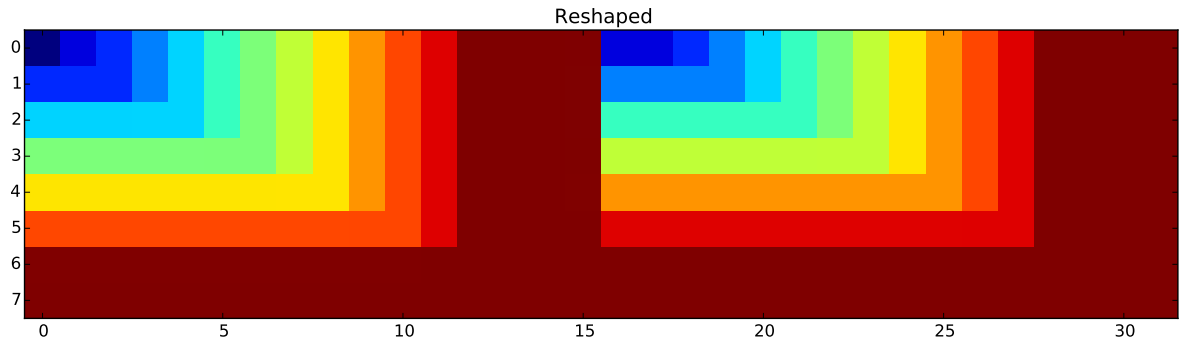

Figure 2.4: Reshaping of the *quantics* extended `full` view.

The *quantics* extension is used mainly to obtain a complete folding of base `Q`. In this case this is obtained by:

```
>>> import math
>>> TW.reshape( [2] * int(round(math.log(TW.size,2))) )
>>> TW.get_extended_shape()
```

```
(16, 16)
>>> TW.get_shape()
(2, 2, 2, 2, 2, 2, 2, 2)
>>> TW.shape
(2, 2, 2, 2, 2, 2, 2, 2)
```

We finally can reset the shape to the *view* shape using:

```
>>> TW.reset_shape()
```

2.1.7 Summary of shapes

Information regarding several shape transformations are always hold in the data structure. A hierarchy of shapes is used. The top shape is the **global** shape. In the following table we list the different shapes, their description and the main functions related and affecting them.

Shape	Description	Functions
Global	This is the underlying shape of the `TensorWrapper`.	`get_global_shape()`, `get_global_ndim()`, `get_global_size()`, `refine()`
View	Multiple views can be defined for a `TensorWrapper`. The views are defined as nested grids into the global grid. The default view is called `full` and is defined automatically at construction time	`set_view()`, `set_active_view()`, `get_view_shape()`, `get_view_ndim()`, `get_view_size()`, `refine()`
Quantics Extended	Each view can be extended to the next power of `Q` in order to allow the *quantics* folding [2][1] of the tensor.	`set_Q()`, `get_extended_shape()`, `get_extended_ndim()`, `get_extended_size()`
Reshape	This is the result of the reshape of the tensor. If any of the preceding shape transformations have been applied, then the reshape is applied to the lowest transformation.	`reshape()`, `get_shape()`, `get_ndim()`, `get_size()`, `shape`, `ndim`, `size`

Warning: If a shape at any level is modified, every lower reshaping is automatically erased, due to possible inconsistency. For example, if a view is modified, any quantics extension and/or reshape of the view are reset.

Note: The `refine()` function erases all the quantics extensions and the reshapes of each view, but not the views themselves. Instead for each view, the `refine()` function updates the corresponding indices, fitting the old views to the new refinement.

2.1.8 Storage

Instances of the class `TensorWrapper` can be stored in files and reloaded as needed. The class `TensorWrapper` extends the class `storable_object`, which is responsible for storing objects in the `TensorToolbox`.

For the sake of efficiency and readability of the code, the `TensorWrapper` is stored in two different files with a common file name `filename`:

- `filename.pkl` is a serialized version of the object thorugh the pickle library. The `TensorWrapper` serializes a minimal amount of auxiliary information needed for the definition of shapes, meshes, etc. The allocated data are not serialized using pickle, because when the amount of data is big, this would result in a very slow storage.
- `filename.h5` is a binary file containing the allocated data of the `TensorWrapper`. This file is generated using h5py and results in fast loading, writing and appending of data.

Let us store the `TensorWrapper`, we have been using up to now.

```
>>> TW.set_store_location('tensorwrapper')
>>> TW.store(force=True)
```

Check that the files have been stored:

```
$ ls
tensorwrapper.h5  tensorwrapper.pkl  WrapperExample.py
```

Let's now reload the `TensorWrapper`:

```
>>> TW = TT.load('tensorwrapper')
```

The storage of the tensor wrapper can also be triggered using a timer. This is mostly useful when many time consuming computations need to be performed in order to allocate the desired entries of the tensor, and one wants to have always a backup copy of the data. The trigger for the storage is checked any time a new entry needs to be allocated fo storage.

For example, we can set the storage frequency to 5s:

```
>>> import time
>>> TW.data = {}
>>> TW[1,2]
>>> TW.set_store_freq( 5 )
>>> time.sleep(6.0)
>>> TW[3,5]
```

Checking the output we see:

```
$ ls
tensorwrapper.h5      tensorwrapper.pkl     WrapperExample.py
tensorwrapper.h5.old  tensorwrapper.pkl.old
```

where the files `.pkl` and `.h5` are the files stored when the time-trigger is activated, while the files `.pkl.old` and `h5.old` are backup files containing the data stored in the previous example.

2.2 Tensor Train Vectors

2.3 Tensor Train Matrices

2.4 Quantics Tensor Train Vectors

2.5 Quantics Tensor Train Matrices

2.6 Spectral Tensor Train

2.7 Multi-linear algebra

CHAPTER

THREE

API

3.1 Tensor Wrapper

class TensorToolbox.core.**TensorWrapper**(*f*, *X*, *params=None*, *W=None*, *twtype='array'*, *data=None*, *dtype=<type 'object'>*, *store_file=''*, *store_object=None*, *store_freq=None*, *store_overwrite=False*, *empty=False*, *maxprocs=None*, *marshal_f=True*)

A tensor wrapper is a data structure W that given a multi-dimensional scalar function f(X,params), and a set of coordinates {{x1}_i1,{x2}_i2,..,{xd}_id} indexed by the multi index {i1,..,id}, let you access f(x1_i1,..,xd_id) by W[i1,..,id]. The function evaluations are performed "as needed" and stored for future accesses.

Parameters

- **f** – multi-dimensional scalar function of type f(x,params), x being a list.
- **X** (*list*) – list of arrays with coordinates for each dimension
- **params** (*tuple*) – parameters to be passed to function f
- **W** (*list*) – list of arrays with weights for each dimension
- **twtype** (*string*) – 'array' values are stored whenever computed, 'view' values are never stored and function f is always called
- **data** (*dict*) – initialization data of the Tensor Wrapper (already computed entries)
- **dtype** (*type*) – type of output to be expected from f
- **store_file** (*str*) – file where to store the data
- **store_object** (*object*) – a storable object that must be stored in place of the TensorWrapper
- **store_freq** (*bool*) – how frequently to store the TensorWrapper (seconds)
- **store_overwrite** (*bool*) – whether to overwrite pre-existing files.
- **empty** (*bool*) – Creates an instance without initializing it. All the content can be initialized using the `setstate()` function.
- **maxprocs** (*int*) – Number of processors to be used in the function evaluation (MPI)
- **marshal_f** (*bool*) – whether to marshal the function or not

Several shape parameters are used by the TensorWrapper in order to keep track of reshaping and slicings, without affecting the underlying shape of the tensor which is always preserved. The following table lists the existing shapes and their meaning.

Shape attribute/function	Applied transformations (ordered)	Description
`get_global_shape()`	None	The original shape of the tensor. This shape can be modified only through a refinement of the grid using the function `refine()`.
`get_view_shape()`	VIEW	The particular view of the tensor, defined by the view in `TensorWrapper.maps` set active using `set_active_view()`.
`get_extended_shape()`	VIEW, QUANTICS	The shape of the extended tensor in order to allow for the quantics folding with basis `TensorWrapper.Q`.
`get_ghost_shape()`	VIEW, QUANTICS, RESHAPE	The shape of the tensor reshaped using `reshape()`. If a Quantics folding is pre-applied, then the reshape is on the extended shape.
`get_shape()`	VIEW, QUANTICS, RESHAPE, FIX_IDXS	The shape of the tensor with `fix_indices()` and `release_indices()`. This is the view that is always used when the tensor is accessed through the function `__getitem__()` (i.e. `TW[...]`)

__getitem__(*idxs_in*)

extended_is_view(*idxs*)

Returns True if the idxs is in the view shape. False if it is outside

fix_indices(*idxs*, *dims*)

Fix some of the indices in the tensor wrapper and reshape/resize it accordingly. The internal storage of the data is still done with respect to the global indices, but once some indices are fixed, the TensorWrapper can be accessed using just the remaining free indices.

Parameters

- **idxs** (*list*) – list of indices to be fixed
- **dims** (*list*) – list of dimensions to which the indices refer to

get_extended_ndim()

If the quantics folding has been performed on the current view, then this returns the number of dimensions of the extended tensor to the next power of Q. If the folding has not been performed, this returns an error.

get_extended_shape()

If the quantics folding has been performed on the current view, then this returns the shape of the extended tensor to the next power of Q. If the folding has not been performed, this returns the view shape.

get_extended_size()

If the quantics folding has been performed on the current view, then this returns the size of the extended tensor to the next power of Q. If the folding has not been performed, this returns an error.

get_ghost_ndim()

If the `ghost_shape` is set for this view, then it returns the number of dimensions obtained after quantics folding by the function `set_Q()` or after reshaping by the function `reshape()`. Otherwise the number of dimensions of the view is returned.

get_ghost_shape()

If the `ghost_shape` is set for this view, then it returns the shape obtained after quantics folding by the function `set_Q()` or after reshaping by the function `reshape()`. Otherwise the shape of the extended shape is returned.

get_ghost_size()

If the `ghost_shape` is set for this view, then it returns the size obtained after quantics folding by the function `set_Q()` or after reshaping by the function `reshape()`. Otherwise the size of the view is returned.

get_global_ndim()
Always returns the ndim of the underlying tensor

get_global_shape()
Always returns the shape of the underlying tensor

get_global_size()
Always returns the size of the underlying tensor

get_ndim()
Always returns the number of dimensions of the tensor view

get_shape()
Always returns the shape of the actual tensor view

get_size()
Always returns the size of the tensor view

get_view_ndim()
Always returns the ndim of the current view

get_view_shape()
Always returns the shape of the current view

get_view_size()
Always returns the size of the current view

ghost_to_global(*idxs*)
This maps the index from the current ghost shape of the view to the global shape.

Parameters **idxs_in** (*list*) – list of indices to be transformed

---

Note: no slicing is admitted here. Preprocess `idxs` with `expand_idxs()` if slicing is required.

---

ghost_to_view(*idxs*)
This maps the index from the current ghost shape of the view to the view shape.

Parameters **idxs_in** (*list*) – list of indices to be transformed

---

Note: no slicing is admitted here. Preprocess `idxs` with `expand_idxs()` if slicing is required.

---

global_to_ghost(*idxs*)
This maps the index from the global shape to the ghost shape.

Parameters **idxs** (*list*) – list of indices to be transformed

---

Note: no slicing is admitted here. Preprocess `idxs` with `expand_idxs()` if slicing is required.

For `TensorWrapper()` `A`, this corresponds to:

```
>>> A.view_to_ghost( A.global_to_view( idxs ) )
```

global_to_view(*idxs*)
This maps the index from the global shape to the view shape.

Parameters **idxs** (*tuple*) – tuple representing an index to be transformed.

Note: no slicing is admitted here. Preprocess `idxs` with `expand_idxs()` if slicing is required.

Note: this returns an error if the `idxs` do not belong to the index mapping of the current view.

---

refine(*X_new*, *tol=None*)
Refine the global discretization. The new discretization must contain the old one.

This function takes care of updating all the indices in the global view as well in all the other views.

Parameters

- **X_new** (*list*) – list of coordinates of the new refinement
- **tol** (*float*) – tolerance for the matching of coordinates

> **Warning:** Any existing reshaping of the views is discarded.

release_indices()
Release all the indices in the tensor wrapper which were fixed using `fix_indices()`.

reset_ghost_shape()
Reset the shape of the tensor erasing the reshape and quantics foldings.

reset_shape()
Reset the shape of the tensor erasing the reshape and quantics foldings.

reshape(*newshape*)
Reshape the tensor. The number of items in the new shape must be consistent with `get_extended_size()`, i.e. with the number of items in the extended quantics size or the view size if `Q` is not set for this view.

This will unset any fixed index for the current view set using `fix_indices()`.

Parameters **newshape** (*list*) – new shape to be applied to the tensor.

set_Q(*Q*)
Set the quantics folding base for the current view.

This will unset any fixed index for the current view set using `fix_indices()`.

Parameters **Q** (*int*) – folding base.

set_active_view(*view*)
Set a view among the ones in `self.maps`.

Parameters **view** (*str*) – name of the view to be set as active

set_active_weights(*flag*)
Set whether to use the weights or not.

Parameters **flag** (*bool*) – If `True` the items returned by the Tensor Wrapper will be weighted according to the weights provided at construction time. If `False` the original values of the function will be returned.

set_view(*view*, *X_map*, *tol=None*)
Set or add a view to `self.maps`. This resest all the existing reshape parameters in existing views.

Parameters

- **view** (*str*) – name of the view to be added
- **X_map** (*list*) – list of coordinates of the new view
- **tol** (*float*) – tolerance for the matching of coordinates

set_weights(*W*)
Set a new list of weights for the tensor :param list W: list of np.ndarray with weights for each dimension

shape_to_ghost(*idxs_in*)
This maps the index from the current shape of the view (fixed indices) to the ghost shape.

Parameters **idxs_in** (*list*) – list of indices to be transformed

> **Note:** slicing is admitted here.

shape_to_global(*idxs*)
This maps the index from the current shape of the view to the global shape.

Parameters **idxs_in** (*list*) – list of indices to be transformed

Note: no slicing is admitted here. Preprocess `idxs` with `expand_idxs()` if slicing is required.

shape_to_view(*idxs*)
This maps the index from the current shape of the view to the view shape.

Parameters **idxs_in** (*list*) – list of indices to be transformed

Note: no slicing is admitted here. Preprocess `idxs` with `expand_idxs()` if slicing is required.

to_v_0_3_0(*store_location*)
Upgrade to v0.3.0

Parameters **filename** (*string*) – path to the filename. This must be the main filename with no extension.

view_to_ghost(*idxs*)
This maps the index from the view to the ghost shape.

Parameters **idxs** (*list*) – list of indices to be transformed

Note: no slicing is admitted here. Preprocess `idxs` with `expand_idxs()` if slicing is required.

Note: this returns an error if the ghost shape is obtained by quantics folding, because the one view index can be pointing to many indices in the folding.

view_to_global(*idxs*)
This maps the index in view to the global indices of the full tensor wrapper.

Parameters **idxs_in** (*list*) – list of indices to be transformed

Note: no slicing is admitted here. Preprocess `idxs` with `expand_idxs()` if slicing is required.

3.2 Tensor Train Vectors

class `TensorToolbox.core.`**TTvec**(*A*, *store_location=''*, *store_object=None*, *store_freq=1*, *store_overwrite=False*, *multidim_point=None*)
Constructor of multidimensional tensor in Tensor Train format [3]

Parameters

- **A** (*Candecomp,ndarray,TT,TensorWrapper*) – Available input formats are Candecomp, full tensor in numpy.ndarray, Tensor Train structure (list of cores), or a Tensor Wrapper.
- **store_location** (*string*) – Store computed values during construction on the specified file path. The stored values are ttcross_Jinit and the values used in the TensorWrapper. This permits a restart from already computed values. If empty string nothing is done. (method==’ttcross’)
- **store_object** (*string*) – Object to be stored (default are the tensor wrapper and ttcross_Jinit)
- **store_freq** (*int*) – storage frequency. `store_freq==1` stores intermediate values at every iteration. The program stores data every `store_freq` internal iterations. If `store_object` is a SpectralTensorTrain, then `store_freq` determines the number of seconds every which to store values.
- **multidim_point** (*int*) – If the object A returns a multidimensional array, then this can be used to define which point to apply ttcross to.

__getitem__(*idxs*)

build(*eps=1e-10*, *method='svd'*, *rs=None*, *fix_rank=False*, *Jinit=None*, *delta=0.0001*, *maxit=100*, *mv_eps=1e-06*, *mv_maxit=100*, *kickrank=None*)

Common interface for the construction of the approximation.

Parameters

- **eps** (*float*) – [default == 1e-10] For method=='svd': precision with which to approximate the input tensor. For method=='ttcross': TT-rounding tolerance for rank-check.
- **method** (*string*) – 'svd' use singular value decomposition to construct the TT representation [3], 'ttcross' use low rank skeleton approximation to construct the TT representation [4], 'ttdmrg' uses Tensor Train Renormalization Cross to construct the TT representation [5][6], 'ttdmrgcross' uses 'ttdmrg' with 'ttcross' approximation of supercores
- **rs** (*list*) – list of integer ranks of different cores. If `None` then the incremental TTcross approach will be used. (method=='ttcross')
- **fix_rank** (*bool*) – determines whether the rank is allowed to be increased (method=='ttcross')
- **Jinit** (*list*) – list of list of integers containing the r starting columns in the lowrankapprox routine for each core. If `None` then pick them randomly. (method=='ttcross')
- **delta** (*float*) – accuracy parameter in the TT-cross routine (method=='ttcross'). It is the relative error in Frobenious norm between two successive iterations.
- **maxit** (*int*) – maximum number of iterations in the lowrankapprox routine (method=='ttcross')
- **mv_eps** (*float*) – accuracy parameter for each usage of the maxvol algorithm (method=='ttcross')
- **mv_maxit** (*int*) – maximum number of iterations in the maxvol routine (method=='ttcross')
- **fix_rank** – Whether the rank is allowed to increase
- **kickrank** (*int*) – rank overshooting for 'ttdmrg'

get_data_F_norm()

Used to get the Frobeniuos norm of the underlying data. This needs to be redefined in QTTvec in order to get the Frobeniuous norm of the real tensor.

get_ttdmrg_real_subtensor(*C*, *idx*)

Used to get the real subtensor of the underlying data. This needs to be redefined in QTTvec in order to get the subtensor of the real tensor.

inner_ttcross(*rs*, *Jinit*, *delta*, *maxit*, *mv_eps*, *mv_maxit*, *store_init=True*)

Construct a TT representation of A using TT cross

Parameters

- **rs** (*list*) – list of upper ranks (A.ndim)
- **Jinit** (*list*) – list (A.ndim-1) of lists of init indices
- **mv_eps** (*float*) – MaxVol accuracy
- **mv_maxit** (*int*) – maximum number of iterations for MaxVol
- **store_init** (*bool*) – indicates whether to store the init `Jinit` (see `outer_ttcross`)

interpolate(*Ms=None*, *eps=1e-08*, *is_sparse=None*)

Interpolates the values of the TTvec at arbitrary points, using the interpolation matrices `Ms`.

Parameters

- **Ms** (*list*) – list of interpolation matrices for each dimension. Ms[i].shape[1] == self.shape()[i]
- **eps** (*float*) – tolerance with which to perform the rounding after interpolation
- **is_sparse** (*list*) – is_sparse[i] is a bool indicating whether Ms[i] is sparse or not. If 'None' all matrices are non sparse

Returns TTvec interpolation

Return type TTvec

```
>>> from DABISpectralToolbox import DABISpectral1D as S1D
>>> Ms = [ S1D.LinearInterpolationMatrix(X[i],XI[i]) for i in range(d) ]
>>> is_sparse = [True]*d
>>> TTapproxI = TTapprox.interpolate(Ms,eps=1e-8,is_sparse=is_sparse)
```

project(*Vs=None*, *Ws=None*, *eps=1e-08*, *is_sparse=None*)

Project the TTvec onto a set of basis provided, using the Generalized Vandermonde matrices `Vs` and weights `Ws`.

Parameters

- **Vs** (*list*) – list of generalized Vandermonde matrices for each dimension. Ms[i].shape[1] == self.shape()[i]
- **Ws** (*list*) – list of weights for each dimension. Ws[i].shape[0] == self.shape()[i]
- **eps** (*float*) – tolerance with which to perform the rounding after interpolation
- **is_sparse** (*list*) – is_sparse[i] is a bool indicating whether Ms[i] is sparse or not. If 'None' all matrices are non sparse

Returns TTvec containting the Fourier coefficients

Return type TTvec

```
>>> from DABISpectralToolbox import DABISpectral1D as S1D
>>> P = S1D.Poly1D(S1D.JACOBI,(0,0))
>>> x,w = S1D.Quadrature(10,S1D.GAUSS)
>>> X = [x]*d
>>> W = [w]*d
>>> # Compute here the TTapprox at points X
>>> TTapprox = TTvec(....)
>>> # Project
>>> Vs = [ P.GradVandermonde1D(x,10,0,norm=False) ] * d
>>> is_sparse = [False]*d
>>> TTfourier = TTapprox.project(Vs,W,eps=1e-8,is_sparse=is_sparse)
```

rounding(*eps*)

TT-rounding

rounding2(*eps*)

TT-rounding

shape()

Returns the shape of the tensor represented

ttcross(*eps*, *rs*, *Jinit*, *delta*, *maxit*, *mv_eps*, *mv_maxit*, *fix_rank=False*)

Construct a TT representation of A using TT cross. This routine manage the outer loops for incremental ttcross or passes everything to ttcross if rs are specified.

Parameters

- **eps** (*float*) – tolerance with which to perform the TT-rounding and check the rank accuracy
- **rs** (*list*) – list of upper ranks (A.ndim)

- **Jinit** (*list*) – list (A.ndim-1) of lists of init indices
- **delta** (*float*) – TT-cross accuracy
- **maxit** (*int*) – maximum number of iterations for ttcross
- **mv_eps** (*float*) – MaxVol accuracy
- **mv_maxit** (*int*) – maximum number of iterations for MaxVol
- **fix_rank** (*bool*) – Whether the rank is allowed to increase

ttdmrg(*eps*, *Jinit*, *maxit*, *mv_eps*, *mv_maxit*, *kickrank=None*, *cross=False*, *store_init=True*, *loop_detection=False*)
Construct a TT representation of A using TT-Density matrix renormalization group

Parameters

- **eps** (*float*) – Frobenious tolerance of the approximation
- **Jinit** (*list*) – listo (A.ndim-1) of lists of init indices
- **maxit** (*int*) – maximum number of iterations of the ttdmrg
- **mv_eps** (*float*) – MaxVol accuracy
- **mv_maxit** (*int*) – maximum number of iterations for MaxVol
- **kickrank** (*int*) – rank overshooting
- **cross** (*bool*) – if True it uses ttcross for the supercores. If False it uses plain SVD.
- **store_init** (*bool*) – whether to store the initial indices used (restarting from the same indices will lead to the same construction).
- **loop_detection** (*bool*) – whether to check for loops. (Never occurred that we needed it)

ttdmrgcross(*eps*, *Jinit*, *maxit*, *mv_eps*, *mv_maxit*, *kickrank=None*, *store_init=True*, *loop_detection=False*)
Construct a TT representation of A using TT-Density matrix renormalization group

Parameters

- **eps** (*float*) – Frobenious tolerance of the approximation
- **Jinit** (*list*) – listo (A.ndim-1) of lists of init indices
- **maxit** (*int*) – maximum number of iterations of the ttdmrg
- **mv_eps** (*float*) – MaxVol accuracy
- **mv_maxit** (*int*) – maximum number of iterations for MaxVol
- **kickrank** (*int*) – rank overshooting
- **cross** (*bool*) – if True it uses ttcross for the supercores. If False it uses plain SVD.
- **store_init** (*bool*) – whether to store the initial indices used (restarting from the same indices will lead to the same construction).
- **loop_detection** (*bool*) – whether to check for loops. (Never occurred that we needed it)

3.3 Tensor Train Matrices

class TensorToolbox.core.**TTmat**(*A*, *nrows*, *ncols*, *is_sparse=None*, *sparse_ranks=None*, *store_location=''*, *store_object=None*, *store_freq=1*, *store_overwrite=False*)
Constructor of multidimensional matrix in Tensor Train format

Parameters

- **A** (*Candecomp,ndarray,TT*) – Available input formats are Candecomp, full tensor in numpy.ndarray, Tensor Train structure (list of cores), list of sparse matrices of sizes (r_{i-1}*r_{i}*nrows x ncols) (used for fast dot product - limited support for other functionalities)
- **nrows** (*list,int*) – If int then the row size will be the same in all dimensions, if list then len(nrows) == len(self.TT) (numer of cores) and row size will change for each dimension.
- **ncols** (*list,int*) – If int then the column size will be the same in all dimensions, if list then len(ncols) == len(self.TT) (numer of cores) and column size will change for each dimension.
- **is_sparse** (*bool*) – [default == False] if True it uses sparsity to accelerate some computations
- **sparse_ranks** (*list*) – [default==None] mandatory argument when A is a list of sparse matrices. It contains integers listing the TT-ranks of the matrix.

Note: the method __getitem__ is not overwritten, thus the indices used to access the tensor refer to the flatten versions of the matrices composing the matrix tensor.

---

build(*eps=1e-10*, *method='svd'*, *rs=None*, *fix_rank=False*, *Jinit=None*, *delta=0.0001*, *maxit=100*, *mv_eps=1e-06*, *mv_maxit=100*, *kickrank=2*)

Common interface for the construction of the approximation.

Parameters

- **eps** (*float*) – [default == 1e-10] For method=='svd': precision with which to approximate the input tensor. For method=='ttcross': TT-rounding tolerance for rank-check.
- **method** (*string*) – 'svd' use singular value decomposition to construct the TT representation [3], 'ttcross' use low rank skeleton approximation to construct the TT representation [4], 'ttdmrg' uses Tensor Train Renormalization Cross to construct the TT representation [5][6], 'ttdmrgcross' uses 'ttdmrg' with 'ttcross' approximation of supercores
- **rs** (*list*) – list of integer ranks of different cores. If `None` then the incremental TTcross approach will be used. (method=='ttcross')
- **fix_rank** (*bool*) – determines whether the rank is allowed to be increased (method=='ttcross')
- **Jinit** (*list*) – list of list of integers containing the r starting columns in the lowrankapprox routine for each core. If `None` then pick them randomly. (method=='ttcross')
- **delta** (*float*) – accuracy parameter in the TT-cross routine (method=='ttcross'). It is the relative error in Frobenious norm between two successive iterations.
- **maxit** (*int*) – maximum number of iterations in the lowrankapprox routine (method=='ttcross')
- **mv_eps** (*float*) – accuracy parameter for each usage of the maxvol algorithm (method=='ttcross')
- **mv_maxit** (*int*) – maximum number of iterations in the maxvol routine (method=='ttcross')
- **fix_rank** – Whether the rank is allowed to increase
- **kickrank** (*int*) – rank overshooting for 'ttdmrg'

3.4 Quantics Tensor Train Vectors

class TensorToolbox.core.**QTTvec**(*A*, *global_shape=None*, *base=2*, *store_location=''*, *store_object=None*, *store_freq=1*, *store_overwrite=False*, *multidim_point=None*)

Constructor of multidimensional tensor in Quantics Tensor Train format [1][2].

Parameters

- **A** (*ndarray,TensorWrapper,TT*) – Available input formats are full tensor in numpy.ndarray, Tensor Wrapper, Tensor Train structure (list of cores)
- **global_shape** (*list*) – Argument to be provided if A is the list of cores of a TT format.
- **base** (*int*) – base selected to do the folding

build(*eps=1e-10*, *method='svd'*, *rs=None*, *fix_rank=False*, *Jinit=None*, *delta=0.0001*, *maxit=100*, *mv_eps=1e-06*, *mv_maxit=100*, *kickrank=None*)

Common interface for the construction of the approximation.

Parameters

- **eps** (*float*) – [default == 1e-10] For method=='svd': precision with which to approximate the input tensor. For method=='ttcross': TT-rounding tolerance for rank-check.
- **method** (*string*) – 'svd' use singular value decomposition to construct the TT representation [3], 'ttcross' use low rank skeleton approximation to construct the TT representation [4], 'ttdmrg' uses Tensor Train Renormalization Cross to construct the TT representation [5][6], 'ttdmrgcross' uses 'ttdmrg' with 'ttcross' approximation of supercores
- **rs** (*list*) – list of integer ranks of different cores. If None then the incremental TTcross approach will be used. (method=='ttcross')
- **fix_rank** (*bool*) – determines whether the rank is allowed to be increased (method=='ttcross')
- **Jinit** (*list*) – list of list of integers containing the r starting columns in the lowrankapprox routine for each core. If None then pick them randomly. (method=='ttcross')
- **delta** (*float*) – accuracy parameter in the TT-cross routine (method=='ttcross'). It is the relative error in Frobenious norm between two successive iterations.
- **maxit** (*int*) – maximum number of iterations in the lowrankapprox routine (method=='ttcross')
- **mv_eps** (*float*) – accuracy parameter for each usage of the maxvol algorithm (method=='ttcross')
- **mv_maxit** (*int*) – maximum number of iterations in the maxvol routine (method=='ttcross')
- **fix_rank** – Whether the rank is allowed to increase
- **kickrank** (*int*) – rank overshooting for 'ttdmrg'

get_data_F_norm()

Used to get the Frobeniuos norm of the underlying data.

get_folded_shape()

Return the shape of the folded tensor (list of lists)

get_global_ndim()

Return the ndim of the original tensor

get_global_shape()

Return the shape of the original tensor

get_q_shape()
Return the shape of the base "base" shape of the tensor

get_ttdmrg_real_subtensor(*C*, *idxs*)
Used to get the Frobeniuos norm of a subtensor of the underlying data.

Parameters

- **C** (*np.ndarray*) – Extracted 4-d tensor with shape len(l_idx) x n x m x len(r_idx)
- **idxs** (*list*) – List of tuples of the form (l_idx, slice, slice, r_idx)

Return np.ndarray Creal 1-d array containing the filtered values belonging to the real tensor

interpolate(*Ms=None*, *eps=1e-08*, *is_sparse=None*)
Interpolates the values of the QTTvec at arbitrary points, using the interpolation matrices `Ms`.

Parameters

- **Ms** (*list*) – list of interpolation matrices for each dimension. Ms[i].shape[1] == self.shape()[i]
- **eps** (*float*) – tolerance with which to perform the rounding after interpolation
- **is_sparse** (*list*) – is_sparse[i] is a bool indicating whether Ms[i] is sparse or not. If 'None' all matrices are non sparse [sparsity is not exploited]

Returns QTTvec interpolation

Return type QTTvec

```
>>> from DABISpectralToolbox import DABISpectral1D as S1D
>>> Ms = [ S1D.LinearInterpolationMatrix(X[i],XI[i]) for i in range(d) ]
>>> is_sparse = [True]*d
>>> TTapproxI = TTapprox.interpolate(Ms,eps=1e-8,is_sparse=is_sparse)
```

project(*Vs=None*, *Ws=None*, *eps=1e-08*, *is_sparse=None*)
Project the QTTvec onto a set of basis provided, using the Generalized Vandermonde matrices `Vs` and weights `Ws`.

Parameters

- **Vs** (*list*) – list of generalized Vandermonde matrices for each dimension. Ms[i].shape[1] == self.shape()[i]
- **Ws** (*list*) – list of weights for each dimension. Ws[i].shape[0] == self.shape()[i]
- **eps** (*float*) – tolerance with which to perform the rounding after interpolation
- **is_sparse** (*list*) – is_sparse[i] is a bool indicating whether Ms[i] is sparse or not. If 'None' all matrices are non sparse [sparsity is not exploited]

Returns TTvec containting the Fourier coefficients

Return type TTvec

```
>>> from DABISpectralToolbox import DABISpectral1D as S1D
>>> P = S1D.Poly1D(S1D.JACOBI,(0,0))
>>> x,w = S1D.Quadrature(10,S1D.GAUSS)
>>> X = [x]*d
>>> W = [w]*d
>>> # Compute here the TTapprox at points X
>>> TTapprox = QTTvec(....)
>>> # Project
>>> Vs = [ P.GradVandermonde1D(x,10,0,norm=False) ] * d
>>> is_sparse = [False]*d
>>> TTfourier = TTapprox.project(Vs,W,eps=1e-8,is_sparse=is_sparse)
```

q_to_global(*idxs*)
This is a non-injective function from the q indices to the global indices

3.5 Quantics Tensor Train Matrices

class TensorToolbox.core.**QTTmat**(*A*, *base*, *nrows*, *ncols*, *is_sparse=None*, *store_location=''*, *store_object=None*, *store_freq=1*, *store_overwrite=False*)
Constructor of multidimensional matrix in Quantics Tensor Train format [1][2].

Parameters

- **A** (*ndarray,TT*) – Available input formats are full tensor in numpy.ndarray, Tensor Train structure (list of cores). If input is ndarray, then it must be in mattensor format (see aux.py)
- **base** (*int*) – folding base for QTT representation
- **nrows** (*int*) – If int then the row size will be the same in all dimensions, if list then len(nrows) == len(self.TT) (numer of cores) and row size will change for each dimension.
- **ncols** (*int*) – If int then the column size will be the same in all dimensions, if list then len(ncols) == len(self.TT) (numer of cores) and column size will change for each dimension.

build(*eps=1e-10*, *method='svd'*, *rs=None*, *fix_rank=False*, *Jinit=None*, *delta=0.0001*, *maxit=100*, *mv_eps=1e-06*, *mv_maxit=100*, *kickrank=2*)
Common interface for the construction of the approximation.

Parameters

- **eps** (*float*) – [default == 1e-10] For method=='svd': precision with which to approximate the input tensor. For method=='ttcross': TT-rounding tolerance for rank-check.
- **method** (*string*) – 'svd' use singular value decomposition to construct the TT representation [3], 'ttcross' use low rank skeleton approximation to construct the TT representation [4], 'ttdmrg' uses Tensor Train Renormalization Cross to construct the TT representation [5][6], 'ttdmrgcross' uses 'ttdmrg' with 'ttcross' approximation of supercores
- **rs** (*list*) – list of integer ranks of different cores. If `None` then the incremental TTcross approach will be used. (method=='ttcross')
- **fix_rank** (*bool*) – determines whether the rank is allowed to be increased (method=='ttcross')
- **Jinit** (*list*) – list of list of integers containing the r starting columns in the lowrankapprox routine for each core. If `None` then pick them randomly. (method=='ttcross')
- **delta** (*float*) – accuracy parameter in the TT-cross routine (method=='ttcross'). It is the relative error in Frobenious norm between two successive iterations.
- **maxit** (*int*) – maximum number of iterations in the lowrankapprox routine (method=='ttcross')
- **mv_eps** (*float*) – accuracy parameter for each usage of the maxvol algorithm (method=='ttcross')
- **mv_maxit** (*int*) – maximum number of iterations in the maxvol routine (method=='ttcross')
- **fix_rank** – Whether the rank is allowed to increase
- **kickrank** (*int*) – rank overshooting for 'ttdmrg'

get_full_ncols()
Returns the number of cols of the unfolded matrices

get_full_nrows()
Returns the number of rows of the unfolded matrices

get_ncols()
Returns the number of cols of the folded matrices

get_nrows()
Returns the number of rows of the folded matrices

ndims()
Return the number of dimensions of the tensor space

3.6 Weighted Tensor Train Vectors

class TensorToolbox.core.**WTTvec**(*A*, *W*, *store_location=''*, *store_object=None*, *store_freq=1*, *store_overwrite=False*, *multidim_point=None*)
Constructor of multidimensional tensor in Weighted Tensor Train format [3]

Parameters

- **A** (*Candecomp,ndarray,TT,TensorWrapper*) – Available input formats are Candecomp, full tensor in numpy.ndarray, Tensor Train structure (list of cores), or a Tensor Wrapper.
- **W** (*list*) – list of 1-dimensional ndarray containing the weights for each dimension.
- **store_location** (*string*) – Store computed values during construction on the specified file path. The stored values are ttcross_Jinit and the values used in the TensorWrapper. This permits a restart from already computed values. If empty string nothing is done. (method=='ttcross')
- **store_object** (*string*) – Object to be stored (default are the tensor wrapper and ttcross_Jinit)
- **store_freq** (*int*) – storage frequency. `store_freq==1` stores intermediate values at every iteration. The program stores data every `store_freq` internal iterations. If `store_object` is a SpectralTensorTrain, then `store_freq` determines the number of seconds every which to store values.
- **multidim_point** (*int*) – If the object A returns a multidimensional array, then this can be used to define which point to apply ttcross to.

apply_weights_on_data()
Apply the weights on the input data A

build(*eps=1e-10*, *method='svd'*, *rs=None*, *fix_rank=False*, *Jinit=None*, *delta=0.0001*, *maxit=100*, *mv_eps=1e-06*, *mv_maxit=100*, *kickrank=None*)
Common interface for the construction of the approximation.

Parameters

- **eps** (*float*) – [default == 1e-10] For method=='svd': precision with which to approximate the input tensor. For method=='ttcross': TT-rounding tolerance for rank-check.
- **method** (*string*) – 'svd' use singular value decomposition to construct the TT representation [3], 'ttcross' use low rank skeleton approximation to construct the TT representation [4], 'ttdmrg' uses Tensor Train Renormalization Cross to construct the TT representation [5][6], 'ttdmrgcross' uses 'ttdmrg' with 'ttcross' approximation of supercores
- **rs** (*list*) – list of integer ranks of different cores. If `None` then the incremental TTcross approach will be used. (method=='ttcross')

- **fix_rank** (*bool*) – determines whether the rank is allowed to be increased (method=='ttcross')
- **Jinit** (*list*) – list of list of integers containing the r starting columns in the lowrankapprox routine for each core. If `None` then pick them randomly. (method=='ttcross')
- **delta** (*float*) – accuracy parameter in the TT-cross routine (method=='ttcross'). It is the relative error in Frobenious norm between two successive iterations.
- **maxit** (*int*) – maximum number of iterations in the lowrankapprox routine (method=='ttcross')
- **mv_eps** (*float*) – accuracy parameter for each usage of the maxvol algorithm (method=='ttcross')
- **mv_maxit** (*int*) – maximum number of iterations in the maxvol routine (method=='ttcross')
- **fix_rank** – Whether the rank is allowed to increase
- **kickrank** (*int*) – rank overshooting for 'ttdmrg'

remove_weights_from_data()
Removes the weights from the input data A

3.7 Spectral Tensor Train

class TensorToolbox.core.**STT**(*f*, *grids*, *params*, *range_dim=0*, *marshal_f=True*, *surrogateONOFF=False*, *surrogate_type=None*, *orders=None*, *orderAdapt=None*, *eps=0.0001*, *method='ttdmrg'*, *rs=None*, *fix_rank=False*, *Jinit=None*, *delta=0.0001*, *maxit=100*, *mv_eps=1e-06*, *mv_maxit=100*, *kickrank=None*, *store_location=''*, *store_overwrite=False*, *store_freq=0*)
Constructor of the Spectral Tensor Train approximation [7]. Given a function `f(x,theta,params):(Is, It) -> R` with `dim(Is)=n` and `dim(It)=d`, construct an approximation of `g(theta,params): It -> h_t(Is)`. For example `Is` could be the discretization of a spatial dimension, and `It` some parameter space, so that `f(x,theta,params)` describes a scalar field depending some parameters that vary in `It`. The `params` in the definition of `f` can be constants used by the function or othere objects that must be passed to the function definition.

Parameters

- **f** (*function*) – multidimensional function to be approximated with format `f(x,theta,params)`
- **grids** (*list*) – this is a list with `len(grids)=dim(Is)+dim(It)` which can contain: a) 1-dimensional numpy.array of points discretizing the i-th dimension, b) a tuple `(PolyType,QuadType,PolyParams,span)` where `PolyType` is one of the polynomials available in `SpectralToolbox.Spectral1D` and `QuadType` is one of the quadrature rules associated to the selected polynomial and `PolyParams` are the parameters for the selected polynomial. `span` is a tuple defining the left and right end for dimension i (Example: `(-3,np.inf)`) c) a tuple `(QuadType,span)` where `QuadType` is one of the quadrature rules available in `SpectralToolbox.Spectral1D` without the selection of a particular polynomial type, and `span` is defined as above.
- **params** (*object*) – any list of parameters to be passed to the function `f`
- **range_dim** (*int*) – define the dimension of the spatial dimension `Is`. For functionals `f(theta,params)`, `dim(Is)=0`. For scalar fileds in 3D, `dim(Is)=3`.
- **marshal_f** (*bool*) – whether to marshal the function f or not. For MPI support, the function f must be marshalable (does this adverb exists??).

- **surrogateONOFF** (*bool*) – whether to construct the surrogate or not
- **surrogate_type** (*str*) – whether the surrogate will be an interpolating surrogate (`TensorTrain.LINEAR_INTERPOLATION` or `TensorTrain.LAGRANGE_INTERPOLATION`) or a projection surrogate (`TensorTrain.PROJECTION`)
- **orders** (*list*) – polynomial orders for each dimension if `TensorTrain.PROJECTION` is used. If `orderAdapt==True` then the `orders` are starting orders that can be increased as needed by the construction algorithm. If this parameter is not provided but `orderAdapt==True`, then the starting order is 1 for all the dimensions.
- **orderAdapt** (*bool*) – whether the order is fixed or not.
- **stt_store_location** (*str*) – path to a file where function evaluations can be stored and used in order to restart the construction.
- **stt_store_overwrite** (*bool*) – whether to overwrite pre-existing files
- **stt_store_freq** (*int*) – storage frequency. Determines every how many seconds the state is stored. `stt_store_freq==0` stores every time it is possible.
- **empty** (*bool*) – Creates an instance without initializing it. All the content can be initialized using the `setstate()` function.

Note: For a description of the remaining parameters see `TTvec`.

`__getitem__``()`

`__call__`(*x_in*, *verbose=False*)

Evaluate the surrogate on points `x_in`

Parameters **x_in** (*np.ndarray*) – 1 or 2 dimensional array of points in the parameter space where to evaluate the function. In 2 dimensions, each row is an entry, i.e. `x_in.shape[1] == self.param_dim`

Returns an array with dimension equal to the space dimension (`range_dim`) plus one. If `A` is the returned vector and `range_dim=2`, then `A[i,:,:]` is the value of the surrogate for `x_in[i,:]`

`integrate``()`

Compute the integral of the approximated function

Returns an array with dimension equal to the space dimension (`range_dim`), containing the value of the integral.

`prepare_TTapprox`(*force_redo=False*)

Prepares the TTapprox from the generic_approx

`to_v_0_3_0`(*store_location*)

Upgrade to v0.3.0

Parameters **filename** (*string*) – path to the filename. This must be the main filename with no extension.

3.8 Multi-linear algebra

`TensorToolbox.multilinalg.`**`mul`**(*A*, *B*)

•If A,B are TTvec/TTmat -> Hadamard product of two TT tensors

•If A TTvec/TTmat and B scalar -> multiplication by scalar

TensorToolbox.multilinalg.**kron**(*A*, *B*)
Kron product between two tensors in TT format. Complexity: O(1)

TensorToolbox.multilinalg.**contraction**(*A*, *U*)
Multidimensional contraction of tensor A with vectors in list U. Complexity: O(dnr^2)

Syntax: `W = contraction(A,U)`

Parameters

- **A** – Tensor in some form
- **U** – list of vectors of dimensions n_k for performing the contraction

TensorToolbox.multilinalg.**norm**(*A*, *ord='fro'*, *round_eps=1e-10*, *eps=0.0001*, *maxit=1000*, *pow_guess=None*, *info=False*)
Compute the norm of tensor A.

Syntax: `w = norm(A,[ord])`

Parameters

- **A** – Tensor in some form
- **ord** – Specifies the type of norm that needs to be computed. Available norms are: the Frobenius norm: 'fro'. If the input tensor is a WeightedTensorTrainVec, then this takes the Frobenious norm of the weighted TT, i.e. the continuos norm defined by the weights.

TensorToolbox.multilinalg.**sd**(*A*, *b*, *x0=None*, *eps=1e-08*, *maxit=1000*, *eps_round=1e-10*, *ext_info=False*)
Solves the system $Ax = b$ using the Steepest Descent method in Tensor Train format.

Parameters

- **A** (*TTmat*) – Tensor train matrix
- **b** (*TTvec/ndarray*) – Right hand side
- **x0** (*TTvec/ndarray*) – [default == `TensorToolbox.core.zerosvec()`] initial guess of solution `x`
- **eps** (*float*) – [default == 1e-8] stop criteria
- **maxit** (*int*) – [default == 1000] maximum number of iterations
- **eps_round** (*float*) – [default == 1e-10] accuracy for Tensor Train rounding operations
- **ext_info** (*bool*) – [default == False] whehter of not to have additional info returned

Returns

tuple `(x,conv,info)`

- `x` (TTvec): solution of the linear system if converged or last iterate if not converged
- `conv` (bool): True -> converged, False -> Not converged / Zero Inner Product exeception
- `info` (dict): `iter` -> total number of iterations; `r` -> last residual in TT format; `res` -> residual history

TensorToolbox.multilinalg.**cg**(*A*, *b*, *x0=None*, *eps=1e-08*, *maxit=1000*, *eps_round=1e-10*, *ext_info=False*)
Solves the system $Ax = b$ using the Conjugate Gradient method in Tensor Train format.

Parameters

- **A** (*TTmat*) – Tensor train matrix
- **b** (*TTvec/ndarray*) – Right hand side

- **x0** (*TTvec/ndarray*) – [default == `TensorToolbox.core.zerosvec()`] initial guess of solution `x`
- **eps** (*float*) – [default == 1e-8] stop criteria for Bi-CGSTAB iterations
- **maxit** (*int*) – [default == 1000] maximum number of iterations for Bi-CGSTAB
- **eps_round** (*float*) – [default == 1e-10] accuracy for Tensor Train rounding operations
- **ext_info** (*bool*) – [default == False] whehter of not to have additional info returned

Returns

tuple `(x,conv,info)`

- `x` (TTvec): solution of the linear system if converged or last iterate if not converged
- `conv` (bool): True -> converged, False -> Not converged / Zero Inner Product exeception
- `info` (dict): `iter` -> total number of iterations; `r` -> last residual in TT format; `res` -> residual history

`TensorToolbox.multilinalg.`**`bicgstab`**(*A*, *b*, *x0=None*, *eps=1e-08*, *maxit=1000*, *eps_round=1e-10*, *ext_info=False*)

Solves the system $Ax = b$ using the Bi-Conjugate Gradient Stabilized method using Tensor Train format.

Parameters

- **A** (*TTmat*) – Tensor train matrix
- **b** (*TTvec*) – Right hand side
- **x0** (*TTvec*) – [default == `TensorToolbox.core.zerosvec()`] initial guess of solution `x`
- **eps** (*float*) – [default == 1e-8] stop criteria for Bi-CGSTAB iterations
- **maxit** (*int*) – [default == 1000] maximum number of iterations for Bi-CGSTAB
- **eps_round** (*float*) – [default == 1e-10] accuracy for Tensor Train rounding operations
- **ext_info** (*bool*) – [default == False] whehter of not to have additional info returned

Returns

tuple `(x,conv,info)`

- `x` (TTvec): solution of the linear system if converged or last iterate if not converged
- `conv` (bool): True -> converged, False -> Not converged / Zero Inner Product exeception
- `info` (dict): `iter` -> total number of iterations; `r` -> last residual in TT format; `rho` -> last value of dot(r0,r) must be bigger than np.spacing(1); `r0v` -> last value of dot(r0,v) must be bigger than np.spacing(1)

`TensorToolbox.multilinalg.`**`gmres`**(*A*, *b*, *x0=None*, *eps=1e-08*, *maxit=1000*, *restart=1000*, *eps_round=1e-10*, *ext_info=False*)

Solves the system $Ax = b$ using the Generalized Minimum Residual method with Modified Gram-Schmidt iterations using Tensor Train format.

Parameters

- **A** (*TTmat*) – Tensor train matrix
- **b** (*TTvec*) – Right hand side
- **x0** (*TTvec*) – [default == `TensorToolbox.core.zerosvec()`] initial guess of solution `x`
- **eps** (*float*) – [default == 1e-8] stop criteria for GMRES iterations

- **maxit** (*int*) – [default == 1000] maximum number of iterations for GMRES
- **restart** (*int*) – [default == 1000] restart constant for GMRES (nothing is implemented to retain information, i.e. Hessemberg and Krylov space are reset)
- **eps_round** (*float*) – [default == 1e-10] accuracy for Tensor Train rounding operations
- **ext_info** (*bool*) – [default == False] whehter of not to have additional info returned

Returns

tuple `(x,conv,info)`

- `x` (TTvec): solution of the linear system if converged or last iterate if not converged
- `conv` (bool): True -> converged, False -> Not converged / Zero Inner Product exception
- `info` (dict): `iter` -> total number of iterations; `TT_r` -> last residual in TT format; `res` -> norm of last residual; `err` -> residual history per iteration

Note not optimized for symmetric A

3.9 Canonical Decomposition

class `TensorToolbox.core.`**`Candecomp`**(*A*)

3.10 Miscellaneous

`TensorToolbox.core.`**`idxunfold`**(*dlist*, *idxs*)

Find the index corresponding to the unfolded (flat) version of a tensor

Parameters

- **dlist** (*list,int*) – list of integers containing the dimensions of the tensor
- **idxs** (*list,int*) – tensor index

Returns index for the flatten tensor

`TensorToolbox.core.`**`idxfold`**(*dlist*, *idx*)

Find the index corresponding to the folded version of a tensor from the flatten version

Parameters

- **dlist** (*list,int*) – list of integers containing the dimensions of the tensor
- **idx** (*int*) – tensor flatten index

Returns list of int – the index for the folded version

Note this routine can be used to get the indexes of a TTmat from indices of a matkron (matrix obtained using np.kron): (i,j) in N^d x N^d -> ((i_1,..,i_d),(j_1,..,j_d)) in (N x .. x N) x (N x .. x N)

`TensorToolbox.core.`**`expand_idxs`**(*idxs_in*, *shape*, *ghost_shape=None*, *fix_dims=None*, *fix_idxs=None*)

From a tuple of indicies, apply all the unslicing transformations and restore the fixed indices in order to extract values from a tensor with a certain `ghost_shape`. If `ghost_shape==None`, `fix_dims==None` and `fix_idxs==None`, then this performs only an unslicing of the index and it is assumed `ghost_shape=shape`.

Parameters

- **idxs_in** (*tuple*) – indexing tuple. The admissible slicing format is the same used in np.ndarray.

- **shape** (*tuple*) – shape of the tensor
- **ghost_shape** (*tuple*) – shape of the tensor without fixed indices
- **fix_dims** (*list*) – whether there are dimensions which had been fixed and need to be added.
- **fix_idxs** (*list*) – fixed idxs for each fixed dimension.

Returns tuple `(lidxs,out_shape,transpose_list_shape)`. `lidxs` is an iterator of the indices. `out_shape` is a tuple containing the shape of the output tensor. `transpose_list_shape` is a flag indicating whether the output format need to be transposed (behaving accordingly to np.ndarray).

`TensorToolbox.core.`**`matkron_to_mattensor`**(*A*, *nrows*, *ncols*)

This function reshapes a 2D-matrix obtained as kron product of len(nrows)==len(ncols) matrices, to a len(nrows)-tensor that can be used as input for the TTmat constructor. Applies the Van Loan-Pitsianis reordering of the matrix elements.

Parameters

- **A** (*ndarray*) – 2-dimensional matrix
- **nrows,ncols** (*list,int*) – number of rows and number of columns of the original matrices used for the kron product

`TensorToolbox.core.`**`mat_to_tt_idxs`**(*rowidxs*, *colidxs*, *nrows*, *ncols*)

Mapping from the multidimensional matrix indexing to the tt matrix indexing

(rowidxs,colidxs) = ((i_1,...,i_d),(j_1,...,j_d)) -> (l_1,...,l_d)

Parameters

- **rowidxs,colidxs** (*tuple,int*) – list of row and column indicies. len(rowidxs) == len(colidxs)
- **nrows,ncols** (*tuple,int*) – dimensions of matrices

Returns tuple,int indices in the tt format

`TensorToolbox.core.`**`tt_to_mat_idxs`**(*idxs*, *nrows*, *ncols*)

Mapping from the tt matrix indexing to the multidimensional matrix indexing

(l_1,...,l_d) -> (rowidxs,colidxs) = ((i_1,...,i_d),(j_1,...,j_d))

Parameters

- **idxs** (*tuple,int*) – list of tt indicies. len(idxs) == len(nrows) == len(ncols)
- **nrows,ncols** (*tuple,int*) – dimensions of matrices

Returns (rowidxs,colidxs) = ((i_1,..,i_d),(j_1,..,j_d)) indices in the matrix indexing

`TensorToolbox.core.`**`maxvol`**(*A*, *delta=0.01*, *maxit=100*)

Find the rxr submatrix of maximal volume in A(nxr), n>=r

Parameters

- **A** (*ndarray*) – two dimensional array with (n,r)=shape(A) where r<=n
- **delta** (*float*) – stopping cirterion [default=1e-2]
- **maxit** (*int*) – maximum number of iterations [default=100]

Returns `(I,AsqInv,it)` where `I` is the list or rows of A forming the matrix with maximal volume, `AsqInv` is the inverse of the matrix with maximal volume and `it` is the number of iterations to convergence

Raises raise exception if the dimension of A is r>n or if A is singular

Raises ConvergenceError if convergence is not reached in maxit iterations

TensorToolbox.core.**lowrankapprox**(*A*, *r*, *Jinit=None*, *delta=1e-05*, *maxit=100*, *maxvoleps=0.01*, *maxvolit=100*)

Given a matrix A nxm, find the maximum volume submatrix with rank r<n,m.

Parameters

- **A** (*ndarray*) – two dimensional array with dimension nxm
- **r** (*int*) – rank of the maxvol submatrix
- **Jinit** (*list*) – list of integers containing the r starting columns. If `None` then pick them randomly.
- **delta** (*float*) – accuracy parameter
- **maxit** (*int*) – maximum number of iterations in the lowrankapprox routine
- **maxvoleps** (*float*) – accuracy parameter for each usage of the maxvol algorithm

Parma int maxvolit maximum number of iterations in the maxvol routine

Returns `(I,J,AsqInv,it)` where `I` and `J` are the list of rows and columns of A that compose the submatrix of maximal volume, `AsqInv` is the inverse of such matrix and `it` is the number of iteration to convergence

TensorToolbox.core.**reort**(*u*, *uadd*)

Golub-Kahan reorthogonalization

TensorToolbox.core.**load**(*filename*, *load_data=True*)

Used to load TensorToolbox data.

Parameters

- **filename** (*string*) – path to the filename. This must be the main filename with no extension.
- **load_data** (*bool*) – whether to load additional data from ".h5" files.

class TensorToolbox.core.**storable_object**(*store_location=''*, *store_freq=None*, *store_overwrite=False*, *store_object=None*)

Constructor of objects that can be stored

Parameters

- **store_location** (*string*) – path to the storage file
- **store_freq** (*int*) – Number of seconds between each storage
- **store_overwrite** (*bool*) – whether to overwrite existing files
- **store_object** (*object*) – parent object to be stored in place of the current object

Attributes: **serialize_list**: list of objects that must be serialized. **subserialize_list**: list of objects for which the serialization must be done separately.

h5load(*h5file*)

Used to load additional data in hdf5 format. To be redefined in subclasses.

h5store(*h5file*)

Used to store additional data in hdf5 format. To be redefined in subclasses.

load(*h5_location=None*)

Used to load additional data.

set_store_freq(*store_freq*)

Set a new store frequency for the object

Parameters **store_freq** (*int*) – new store location

set_store_location(*store_location*)

Set a new store location for the object

Parameters **store_location** (*string*) – new store location

store (*force=False*)
Used to store any object in the library.

Parameters **force** (*bool*) – force storage before time

to_v_0_3_0 (*store_location*)
To be implemented for objects that need to be upgraded to v0.3.0.

Parameters **filename** (*string*) – path to the filename. This must be the main filename with no extension.

TensorToolbox.core.**ttcross_store** (*path*, *TW*, *TTapp*)
Used to store the computed values of a TTcross approximation. Usually needed when the single function evaluation is demanding or when we need to restart TTcross later on.

Parameters

- **path** (*string*) – path pointing to the location where to store the data
- **TW** (*TensorWrapper*) – Tensor wrapper used to build the ttcross approximation. TW.get_data(), TW.get_X() and TW.get_params() will be stored.
- **TTapp** (*TTvec*) – TTcross approximation. TTapp.ttcross.Jinit will be stored.

Deprecated since version 0.3.0: Use the objects' methods `store()`.

TensorToolbox.core.**to_v_0_3_0** (*filename*)
Used to upgrade the storage version from version <0.3.0 to version 0.3.0

Parameters **filename** (*string*) – path to the filename. This must be the main filename with no extension.

TensorToolbox.core.**randvec** (*d*, *N*)
Returns the rank-1 multidimensional random vector in Tensor Train format

Args: d (int): number of dimensions N (int or list): If int then uniform sizes are used for all the dimensions, if list of int then len(N) == d and each dimension will use different size

Returns: TTvec The rank-1 multidim random vector in Tensor Train format

TensorToolbox.core.**zerosvec** (*d*, *N*)
Returns the rank-1 multidimensional vector of zeros in Tensor Train format

Args: d (int): number of dimensions N (int or list): If int then uniform sizes are used for all the dimensions, if list of int then len(N) == d and each dimension will use different size

Returns: TTvec The rank-1 multidim vector of zeros in Tensor Train format

TensorToolbox.core.**eye** (*d*, *N*)
Returns the multidimensional identity operator in Tensor Train format

Args: d (int): number of dimensions N (int or list): If int then uniform sizes are used for all the dimensions, if list of int then len(N) == d and each dimension will use different size

Returns: TTmat The multidim identity matrix in Tensor Train format

Note: TODO: improve construction avoiding passage through Candecomp

TensorToolbox.core.**randmat** (*d*, *nrows*, *ncols*)
Returns the rank-1 multidimensional random matrix in Tensor Train format

Args: d (int): number of dimensions nrows (int or list): If int then uniform sizes are used for all the dimensions, if list of int then len(nrows) == d and each dimension will use different size ncols (int or list): If int then uniform sizes are used for all the dimensions, if list of int then len(ncols) == d and each dimension will use different size

Returns: TTmat The rank-1 multidim random matrix in Tensor Train format

TensorToolbox.core.**QTTzerosvec**(*d*, *N*, *base*)

Returns the rank-1 multidimensional vector of zeros in Quantics Tensor Train format

Args: d (int): number of dimensions N (int or list): If int then uniform sizes are used for all the dimensions, if list of int then len(N) == d and each dimension will use different size base (int): QTT base

Returns: QTTvec The rank-1 multidim vector of zeros in Tensor Train format

class TensorToolbox.core.**SQTT**(*f*, *grids*, *params*, *range_dim=0*, *marshal_f=True*, *base=2*, *surrogateONOFF=False*, *surrogate_type=None*, *orders=None*, *orderAdapt=None*, *eps=0.0001*, *method='ttdmrg'*, *rs=None*, *fix_rank=False*, *Jinit=None*, *delta=0.0001*, *maxit=100*, *mv_eps=1e-06*, *mv_maxit=100*, *kickrank=None*, *store_location=''*, *store_overwrite=False*, *store_freq=0*)

Constructor of the Spectral Quantics Tensor Train approximation. Given a function `f(x,theta,params):(Is, It) -> R` with `dim(Is)=n` and `dim(It)=d`, construct an approximation of `g(theta,params): It -> h_t(Is)`. For example `Is` could be the discretization of a spatial dimension, and `It` some parameter space, so that `f(x,theta,params)` describes a scalar field depending some parameters that vary in `It`. The `params` in the definition of `f` can be constants used by the function or othere objects that must be passed to the function definition.

Parameters

- **f** (*function*) – multidimensional function to be approximated with format `f(x,theta,params)`
- **grids** (*list*) – this is a list with `len(grids)=dim(Is)+dim(It)` which can contain: a) 1-dimensional numpy.array of points discretizing the i-th dimension, b) a tuple `(PolyType,QuadType,PolyParams,span)` where `PolyType` is one of the polynomials available in `SpectralToolbox.Spectral1D` and `QuadType` is one of the quadrature rules associated to the selected polynomial and `PolyParams` are the parameters for the selected polynomial. `span` is a tuple defining the left and right end for dimension i (Example: `(-3,np.inf)`) c) a tuple `(QuadType,span)` where `QuadType` is one of the quadrature rules available in `SpectralToolbox.Spectral1D` without the selection of a particular polynomial type, and `span` is defined as above.
- **params** (*object*) – any list of parameters to be passed to the function `f`
- **range_dim** (*int*) – define the dimension of the spatial dimension `Is`. For functionals `f(theta,params)`, `dim(Is)=0`. For scalar fileds in 3D, `dim(Is)=3`.
- **marshal_f** (*bool*) – whether to marshal the function f or not. For MPI support, the function f must be marshalable (does this adverb exists??).
- **base** (*int*) – `base` parameter for Quantics Tensor Train
- **surrogateONOFF** (*bool*) – whether to construct the surrogate or not
- **surrogate_type** (*str*) – whether the surrogate will be an interpolating surrogate (`TensorTrain.LINEAR_INTERPOLATION` or `TensorTrain.LAGRANGE_INTERPOLATION`) or a projection surrogate (`TensorTrain.PROJECTION`)
- **orders** (*list*) – polynomial orders for each dimension if `TensorTrain.PROJECTION` is used. If `orderAdapt==True` then the `orders` are starting orders that can be increased as needed by the construction algorithm. If this parameter is not provided but `orderAdapt==True`, then the starting order is 1 for all the dimensions.
- **orderAdapt** (*bool*) – whether the order is fixed or not.
- **stt_store_location** (*str*) – path to a file where function evaluations can be stored and used in order to restart the construction.
- **stt_store_overwrite** (*bool*) – whether to overwrite pre-existing files

- **stt_store_freq** (*int*) – storage frequency. Determines every how many seconds the state is stored. `stt_store_freq==0` stores every time it is possible.
- **empty** (*bool*) – Creates an instance without initializing it. All the content can be initialized using the `setstate()` function.

Note: For a description of the remaining parameters see `TTvec`.

__getitem__()

__call__(*x_in*, *verbose=False*)

Evaluate the surrogate on points `x_in`

Parameters **x_in** (*np.ndarray*) – 1 or 2 dimensional array of points in the parameter space where to evaluate the function. In 2 dimensions, each row is an entry, i.e. `x_in.shape[1] == self.param_dim`

Returns an array with dimension equal to the space dimension (`range_dim`) plus one. If `A` is the returned vector and `range_dim=2`, then `A[i,:,:]` is the value of the surrogate for `x_in[i,:]`

3.11 Indices and tables

- *genindex*
- *modindex*
- *search*

CHAPTER

FOUR

REFERENCES

[1] BN Khoromskij and I Oseledets. Quantics-TT collocation approximation of parameter-dependent and stochastic elliptic PDEs. *Comput. Methods Appl. Math.*, 2010.

[2] BN Khoromskij. O (dlog N)-Quantics Approximation of Nd Tensors in High-Dimensional Numerical Modeling. *Constructive Approximation*, pages 257–280, 2011. doi:10.1007/s00365-011-9131-1.

[3] IV Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[4] Ivan Oseledets and Eugene Tyrtyshnikov. TT-cross approximation for multidimensional arrays. *Linear Algebra and its Applications*, 432(1):70–88, January 2010. doi:10.1016/j.laa.2009.07.024.

[5] Dmitry Savostyanov and Ivan Oseledets. Fast adaptive interpolation of multi-dimensional arrays in tensor train format. *The 2011 International Workshop on Multidimensional (nD) Systems*, pages 1–8, September 2011. doi:10.1109/nDS.2011.6076873.

[6] DV Savostyanov. Quasioptimality of maximum-volume cross interpolation of tensors. *arXiv preprint arXiv:1305.1818*, c:1–23, 2013. arXiv:arXiv:1305.1818v2.

[7] Daniele Bigoni, Allan P. Engsig-Karup, and Youssef M Marzouk. Spectral tensor-train decomposition. *arXiv preprint arXiv:1405.5713*, pages 28, 2015. URL: http://arxiv.org/abs/1405.5713, arXiv:1405.5713.

t

Symbols

A

B

C

E

F

G

V

W

Z