
ParPyDTK2 Documentation

Release 1.1.0

Qiao Chen

Sep 28, 2018

CONTENTS:

1	Introduction	1
1.1	Background & Motivation	1
1.2	License	1
1.3	About Me	2
2	Installation	3
2.1	Install MOAB	3
2.2	Install DTK2	4
2.3	Install ParPyDTK2	5
2.4	Using our Docker container	6
3	Some Details	7
3.1	Meshless/Mesh-free	7
3.2	Global IDs/Handles	7
3.3	Treatment of Empty Partitions	7
4	AWLS Extension	8
4.1	Weighted Least Square (WLS) Formulation	8
4.2	Improving the Strategy of Choosing Local Stencil	10
4.3	Adaptation with Non-smooth Functions	11
4.4	Results	12
4.5	Usage	16
5	A Demo	19
6	API	22
6.1	ParPyDTK2 Python API	22
6.2	ParPyDTK2 C++ API	35
7	Indices and tables	51
	Python Module Index	52
	Index	53

INTRODUCTION

1.1 Background & Motivation

Multi-physics coupling has become one of the popular topics. People try to put different models together and simulate the behaviors in a coupled fashion. With the popularity of *partitioned approach*, i.e. solve different domains with different solvers and couple the interface conditions as those solvers boundary conditions, a robust and accurate interface solution remapping operator is needed. The solution transfer problem on its own is not an easy task, since it involves the following research aspects:

1. **numerical method**, i.e. consistent, conservative, high-order convergence.
2. **geometry and data structure**, i.e. efficient and robust treatments of *mesh association* of two (potentially more) general surfaces that come from different discretization methods (*FEM*, *FVM*, *FDM*, etc.) thus having different resolutions.
3. **parallel rendezvous & HPC**, i.e. handling migrating meshes that have different parallel partitions.

[Data Transfer Kit-2.0 \(DTK2\)](#) is a package that is developed at the [Oak Ridge National Laboratory](#). [DTK2](#) provides parallel solution transfer services with *meshless* (a.k.a. *mesh-free*) methods, which are relatively easy to implement and computational efficient. Particularly, we are interested in its *modified moving least square*¹ method that is an improvement of traditional MLS fitting in terms of robustness on featured geometries.

[Mesh-Oriented datABase \(MOAB\)](#) is an array-based general purpose mesh library with MPI support. Array-based mesh data structure is more efficient in both computational cost and memory usage compared to traditional pointer-based data structures. [MOAB](#) has been adapted in [DTK2](#), so we choose to use it as our mesh database for this work.

In multi-physics coupling, a flexible software framework is must. The fact is: the physics solvers may be implemented in different programming languages or shipped as executable binaries (typically commercial codes), this makes using static languages difficult. Python, on the other hand, can easily glue different languages together and drive executable binaries smoothly. Its built-in reference counting, garbage collection, and pass-by-reference make it as one of the best choices for developing multi-physics coupling frameworks. In addition, MPI is well supported through the [mpi4py](#) package. **This motivates us to develop a Python interface for DTK2!**

1.2 License

This package is distributed under MIT License. For detailed information, please take a look at the [LICENSE](#) file.

¹ Slattery, S. Hamilton, T. Evans, “A Modified Moving Least Square Algorithm for Solution Transfer on a Spacer Grid Surface”, ANS MC2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method, Nashville, Tennessee · April 19–23, 2015, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2015).

1.3 About Me

I am a Ph.D. candidate who work with [Dr. Jim Jiao](#) on high-order numerical methods. This work is for testing our software framework of multi-physics coupling in general, *conjugate heat transfer* (CHT) in particular.

Note: Please be aware that I may not have time to maintain this package.

INSTALLATION

Installing this package is not a trivial task due to its heavy dependencies. ParPyDTK2 has the following installation requirements:

1. C++11 compiler
2. MPI
3. [MOAB](#)
4. [DTK2](#) and [Trilinos](#)
5. Python ≥ 3.5
6. [mpi4py](#)
7. [NumPy](#)
8. [setuptools](#)

And both [MOAB](#) and [DTK2](#) have their own dependencies.

In addition, to build the documentation, the following packages are needed:

1. [Sphinx](#)
2. [Doxygen](#)
3. [breathe](#)
4. [numpydoc](#)

The good news is you can install these easily through `pip`.

2.1 Install MOAB

The [MOAB official README](#) has a very clear description of the installation process. Here we take an excerpt of our [MOAB Docker image building script](#):

```
$ git clone --depth=1 https://bitbucket.org/fathomteam/moab.git
$ cd moab
$ autoreconf -fi
$ ./configure \
  --prefix=/usr/local \
  --with-mpi \
  CC=mpicc \
  CXX=mpicxx \
  FC=mpif90 \
```

(continues on next page)

(continued from previous page)

```

F77=mpif77 \
--enable-optimize \
--enable-shared=yes \
--with-blas=-lopenblas \
--with-lapack=-lopenblas \
--with-scotch=/usr/lib \
--with-metis=/usr/lib/x86_64-linux-gnu \
--with-eigen3=/usr/include/eigen3 \
--with-x \
--with-cgns \
--with-netcdf \
--with-hdf5=/usr/lib/hdf5-openmpi \
--with-hdf5-ldflags="-L/usr/lib/hdf5-openmpi/lib" \
--enable-ahf=yes \
--enable-tools=yes
$ make && sudo make install

```

Notice that this is for system installation. Install to your preferred locations if you don't have the root access. Also, turn off those optional packages if you don't have them, only MPI and HDF5 are necessary.

Warning: You must build it into a shared object!

Note: If you use Ubuntu >= 17.10, all those optional packages are likely to be available through apt.

2.2 Install DTK2

DTK2 is shipped as a sub-module of [Trilinos](#), so building [Trilinos](#) is needed. For people who are not familiar with [Trilinos](#), this can be tricky. Therefore, an excerpt of our [DTK2 Docker image building script](#) might be helpful:

```

$ export TRILINOS_VERSION=12-12-1
$ git clone --depth 1 --branch trilinos-release-${TRILINOS_VERSION}
$ cd Trilinos
$ git clone --depth 1 --branch dtk-2.0 \
  https://github.com/unifem/DataTransferKit.git
$ mkdir build && cd build
$ cmake \
  -DCMAKE_INSTALL_PREFIX:PATH=/usr/local \
  -DCMAKE_BUILD_TYPE:STRING=RELEASE \
  -DCMAKE_VERBOSE_MAKEFILE:BOOL=OFF \
  -DCMAKE_SHARED_LIBS:BOOL=ON \
  -DTPL_ENABLE_MPI:BOOL=ON \
  -DTPL_ENABLE_Boost:BOOL=ON \
  -DBoost_INCLUDE_DIRS:PATH=/usr/include/boost \
  -DTPL_ENABLE_Libmesh:BOOL=OFF \
  -DTPL_ENABLE_MOAB:BOOL=ON \
  -DMOAB_INCLUDE_DIRS=${MOAB_ROOT}/include \
  -DMOAB_LIBRARY_DIRS=${MOAB_ROOT}/lib \
  -DTPL_ENABLE_Netcdf:BOOL=ON \
  -DTPL_ENABLE_BinUtils:BOOL=OFF \
  -DTrilinos_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=OFF \

```

(continues on next page)

(continued from previous page)

```

-DTrilinos_ENABLE_ALL_PACKAGES=OFF \
-DTrilinos_EXTRA_REPOSITORIES="DataTransferKit" \
-DTrilinos_ENABLE_EXPLICIT_INSTANTIATION:BOOL=ON \
-DTrilinos_ASSERT_MISSING_PACKAGES:BOOL=OFF \
-DTrilinos_ENABLE_TESTS:BOOL=OFF \
-DTrilinos_ENABLE_EXAMPLES:BOOL=OFF \
-DTrilinos_ENABLE_CXX11:BOOL=ON \
-DTrilinos_ENABLE_Tpetra:BOOL=ON \
-DTpetra_INST_INT_UNSIGNED_LONG:BOOL=ON \
-DTPL_ENABLE_BLAS:BOOL=ON \
-DTPL_BLAS_LIBRARIES=/usr/lib/x86_64-linux-gnu/libopenblas.so \
-DTPL_ENABLE_LAPACK:BOOL=ON \
-DTPL_LAPACK_LIBRARIES=/usr/lib/x86_64-linux-gnu/libopenblas.so \
-DTPL_ENABLE_Eigen:BOOL=ON \
-DTPL_Eigen_INCLUDE_DIRS=/usr/include/eigen3 \
-DTrilinos_ENABLE_DataTransferKit=ON \
-DDataTransferKit_ENABLE_DBC=ON \
-DDataTransferKit_ENABLE_TESTS=ON \
-DDataTransferKit_ENABLE_EXAMPLES=OFF \
-DDataTransferKit_ENABLE_ClangFormat=OFF \
-DTPL_ENABLE_BoostLib:BOOL=OFF \
-DBUILD_SHARED_LIBS:BOOL=ON
$ make && sudo make install

```

Again, this assumes root access, adjust this based on your situation. **DTK2** needs to stay in the root directory of **Trilinos** and be turned on through switches `DTrilinos_EXTRA_REPOSITORIES` and `DTrilinos_ENABLE_DataTransferKit`. The environment var `MOAB_ROOT` is the place where you install **MOAB**.

Note: We recommend that install **DTK2** from [my personal forked repo](#) since we may add/modify the source codes to make **DTK2** more advanced.

2.3 Install ParPyDTK2

Once you have the dependencies setup, installing ParPyDTK2 can be very easy. The easiest way is through PyPI:

```
$ sudo pip3 install parpydtk2
```

However, this assumes that ParPyDTK2 can find **MOAB** and **DTK2** on the system. With different specifications of `install` command, ParPyDTK2 can automatically add different paths to search for **MOAB** and **DTK2**.

```
$ pip3 install parpydtk2 --user
```

will assume **MOAB** and **DTK2** can be found in `USER_BASE/{include,lib}`.

```
$ pip3 install parpydtk2 --prefix=...
```

will allow ParPyDTK2 to search **MOAB** and **DTK2** under the `prefix` directory.

The preferred way is to define the environment variables `PARPYDTK2_MOAB_ROOT` and `PARPYDTK2_DTK_ROOT` before you do `pip install`. For instance,

```
$ export PARPYDTK2_MOAB_ROOT=/path/to/moab/root
$ export PARPYDTK2_DTK_ROOT=/path/to/dtk/root
$ pip3 install parpydtk2 --user
```

Warning: We don't mark `mpi4py` as installation dependency, so you need to install it manually before you install ParPyDTK2. `pip3 install mpi4py` is just fine.

Of course, you can install from source, which can be obtained [here](#). Just make sure you have all Python *dependencies* installed.

```
$ git clone -b parallel https://github.com/chiao45/parpydtk2.git
$ cd parpydtk2
$ python3 setup.py install --user
```

2.4 Using our Docker container

You can try the package through our pre-built [Docker container](#). Two driver scripts are provided in order to easily use the container:

1. `parpydtk2_desktop.py`
2. `parpydtk2_jupyter.py`

The former will launch a desktop environment through VNC, while the latter will run the container as a Jupyter server.

SOME DETAILS

3.1 Meshless/Mesh-free

Point clouds are easier to use (for the end user) compared to meshes. However, there are some drawbacks: 1) using meshes can achieve linear time complexity for mesh associations¹; 2) numerical integrations are not easy; 3) handling cell-averaged data fields, e.g. solutions that come from FVM, potential can lead to problems.

For 3), typically, people assume cell-averaged data to be cell-centered data, which limits to 2nd-order of accuracy. (Notice that this is actually not a big problem in practice.)

3.2 Global IDs/Handles

MOAB uses global IDs for parallel communications as well as **DTK2**. Therefore, ParPyDTK2 expects the user to provide this information. For most applications, this can be computed offline.

The global IDs are unique handles of the vertices in a point cloud. For instance, if one wants to distribute two triangles, then each of them has local IDs from 0 to 2, but unique global IDs that range from 1 to 4.

Note: ParPyDTK2 uses 1-based indexing for global IDs

3.3 Treatment of Empty Partitions

Both **MOAB** and **DTK2** don't natively support empty partitions, which occur pretty frequently in practice. For instance, couple a serial solver with a parallel solver, or couple a commercial code with an open-sourced one through socket and the incoming data partition graph from the commercial code probably doesn't align with that of the open-sourced side.

To support empty partitions, we duplicate the first node in the master process to the processes that are empty. However, this is not complete yet, because the user is not aware of this, so that the values between across duplicated nodes are not synchronized. Of course, we don't want to let the user to explicitly handle this extra layer of communication. To resolve this, a member function, called `resolve_empty_partitions()`, is added to class `IMeshDB` and must be called **collectively** whenever the user updates the field values through `assign_field()`.

¹ Jiao X, Edelsbrunner X, Heath MT. Mesh association: formulation and algorithms. In 8th International Meshing Roundtable. Sandia Report, South Lake Tahoe, CA, 1999; 75–82.

AWLS EXTENSION

Although the *modified moving least square* in DTK2 is a very advanced data remapping method, there are still rooms for improvement.

1. Whenever dealing with Vandermonde systems, stability must be taken into consideration.
2. The local support of the Vandermonde systems should be adaptive based on the properties of local stencils.

The MLS method solves the local problem in the normal equation setting, which doubles the condition number of the system. MMLS solves the local system with *truncated singular value decomposition* (TSVD), but the problem is that the truncated terms in the Vandermonde systems cannot be controlled. Notice that in order to perform the TSVD, an condition number estimator is needed, and MMLS uses the one in LAPACK for general matrices that requires an LU factorization, which takes another $O(n^3)$ FLOPs.

Regarding implementation details, MMLS uses the global cartesian coordinates for constructing Vandermonde matrices and calls SVD incrementally to ensure the resulting systems are full rank, and this procedure is less stable and costly.

For the local stencil choice, directly using the query results is risky, because the stencil might be too large. In general, the number of points in the local stencils should be determined by the column sizes of the Vandermonde systems.

Moreover, after the data has been remapped, an intermediate target solutions are obtained. Combining with the input source data, *a posteriori* error analysis can be performed in order to fix non-smooth values.

With these considerations, we have implemented *adaptive weighted least square* (AWLS) as an extension of the **meshless** methods in DTK2.

4.1 Weighted Least Square (WLS) Formulation

4.1.1 Formulation

Given the source input discrete function values \mathbf{f}^s and target output \mathbf{f}^t , we want to construct a *transfer operator* \mathbf{T} , s.t.

$$\mathbf{f}^t = \mathbf{T} \mathbf{f}^s \quad (4.1)$$

Clearly, such an operator is rectangle with sizes n by m , where n is the number of nodes in target grid while m for that of the source grid.

Therefore, for each target node i , we have an “interpolation” that reads:

$$f_i^t = T_{i,J} \mathbf{f}_J^s \quad (4.2)$$

where \mathbf{J} is the local support stencil around target node i . Denote $\mathbf{c}^T = T_{i,\mathbf{J}}$, and localize the stencil \mathbf{J} around the target node \mathbf{x}_i and denote the localized stencil to be \mathbf{u} , i.e. $\mathbf{u} = \mathbf{x} - \mathbf{x}_i$, we can then build the *generalized Vandermonde system* (assume 2D):

$$\mathbf{V} = \begin{bmatrix} 1 & \mathbf{u}_1 & \mathbf{u}_2 & \mathbf{u}_1^2 & \mathbf{u}_1\mathbf{u}_2 & \mathbf{u}_2^2 \end{bmatrix} \quad (4.3)$$

So the coefficients \mathbf{c}^T can be solved with the following fitting problem:

$$\mathbf{V}^T \mathbf{c} = \mathbf{e}_1 \quad (4.4)$$

If \mathbf{V} is square system, i.e. typical Vandermonde system, then (4.4) is just an interpolation problem. When we have more points than the number of columns in \mathbf{V} , it results a rectangle system thus requiring least square minimization.

Note that a more intuitive derivation of (4.4) is to first form a system of polynomial coefficients for \mathbf{J} and explicitly compute the values of any points in \mathbf{u} .

$$\begin{aligned} \mathbf{V}\mathbf{C} &= \mathbf{I} \\ \mathbf{C} &= \mathbf{V}^+ \end{aligned}$$

Where \mathbf{V}^+ is the *pseudo-inverse* of \mathbf{V} . Given a point \mathbf{p} in the local coordinate system \mathbf{u} , its Vandermonde component is:

$$\mathbf{p}_V = [1 \ p_1 \ p_2 \ p_1^2 \ p_1 p_2 \ p_2^2]^T \quad (4.5)$$

Then evaluating (4.5) in the system of polynomial coefficients, i.e. (4.1.1), is just to perform $\mathbf{p}_V^T \mathbf{C}$. It's worth noting that for the fitting problem, where the query point is always the center (the target node of interest) thus having the the Vandermonde component $[1 \ 0 \ 0 \ 0 \ 0 \ 0]^T$. Therefore, the explicit computation reduces to (4.4).

It is well-known that the Vandermonde systems are ill-conditioned, so a balancing technique is needed. A typical way is to do a column scaling: $\mathbf{V}\mathbf{S}$. Typical choices of the diagonal matrix \mathbf{S} are 1) *algebraic scaling* that is based on the norms of column vectors of \mathbf{V} and 2) *geometric scaling* that is based on the radii of the local stencil \mathbf{J} . Here we choose the latter, and the local stencil radii is chosen as:

$$r = \max_{j \in \mathbf{J}} \left(\max_{1 \leq d \leq D} (|\mathbf{u}_d^j|) \right) \quad (4.6)$$

Where D is the spatial dimension. Then the column scaling matrix is (assume 2D):

$$\mathbf{S} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/r & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/r & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/r^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/r^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/r^2 \end{bmatrix} \quad (4.7)$$

Note: \mathbf{S} doesn't affect the solution.

Now, the least square problem (4.1.1) can be formulated as:

$$\min(\|\mathbf{V}\mathbf{C} - \mathbf{I}\|_{\mathbf{W}}) \quad (4.8)$$

We choose to use the family of *radius basis functions* (RBF) as the diagonal row weighting matrix \mathbf{W} .

Note: \mathbf{W} does affect the solution.

With (4.7) and (4.8), we have a weighted and balanced generalized Vandermonde system:

$$\hat{V} = WVS \quad (4.9)$$

Plug (4.9) into (4.4) and reorganize it, we have:

$$\begin{aligned} \hat{V}^T W^{-1} c &= S^{-1} e_1 \\ \hat{V}^T \hat{c} &= e_1 \end{aligned}$$

Where $\hat{c} = W^{-1}c$.

4.1.2 Solving WLS

We follow the technique of using *truncated QR with column pivoting* (TQRCP) introduced here¹ to solve the problem (4.1.1).

The first step is to decompose \hat{V} with QRCP:

$$\hat{V}P = QR \quad (4.10)$$

The truncation step is to utilize a condition number estimator for the upper triangular system R that will report the rank of the system— k . Combine this with (4.10) and plug into (4.1.1), we have:

$$\begin{aligned} P_{:,1:k} R_{1:k,1:k}^T Q_{:,1:k}^T \hat{c} &= e_1 \\ R_{1:k,1:k}^T Q_{:,1:k}^T \hat{c} &= P_{:,1:k}^T e_1 \\ Q_{:,1:k}^T \hat{c} &= R_{1:k,1:k}^{-T} P_{:,1:k}^T e_1 \\ \hat{c} &= Q_{:,1:k} R_{1:k,1:k}^{-T} P_{:,1:k}^T e_1 \\ c &= W Q_{:,1:k} R_{1:k,1:k}^{-T} P_{:,1:k}^T e_1 \end{aligned}$$

This procedure is very efficient and robust. The dominated computation cost comes from QRCP, i.e. $O(n^3)$, where n is some measure of the size of local problems. It's worth noting that QRCP is called only once for each of the local problems.

4.2 Improving the Strategy of Choosing Local Stencil

The original DTK2 stencil choice is based on a global configuration parameter of either `knn` (k nearest neighborhoods) or `radius`. Moreover, the resulting neighbors are used to be the support of the local problems, i.e. number of points used in the Vandermonde systems. This procedure is not robust for problems with point clouds that are not uniformly distributed. To overcome this, we have implemented the following three steps for querying and choosing local supports.

1. Perform `knn` search with a small number of neighborhoods to estimate the radii of the one-ring neighborhood of the local point clouds, say h .
2. Perform `radius` search with a relatively large radius so that we have enough candidates for next step. The radius is chosen as: $r = ah$.
3. Choose the support stencil, i.e. J from the candidates based on the number of columns (coefficients) in the Vandermonde systems, i.e. $|J| = \rho c_d$. Notice that c_d are 3, 6, and 10 for dimension (subscript d) 1, 2, and 3 for quadratic polynomial basis functions, respectively.

¹ R. Conley, T. J. Delaney, and X. Jiao. Overcoming element quality dependence of finite elements with adaptive extended stencil FEM (AES-FEM). Int. J. Numer. Meth. Engrg., vol. 108, pp. 1054–1085, 2016.

Note: We choose $a = 5$ for candidate selections and observe that better results can be obtained with $\rho \approx 3$.

Note: Theoretically, steps 1 and 2 can be achieved with `knn` search if we can determine the number points in step 3. However, since we have ran into robustness issues with `DTK2` while using `knn` as the primary search mechanism, we rely on the `radius` search.

With this strategy, the user doesn't need to provide a radius. However, we still believe using topology based query is still efficient and better, but this requires meshes, which is an arduous task for this work.

4.2.1 Automate Parallel Mesh Rendezvous

While use `DTK2` in parallel, each operator is built based on the cartesian bounding box the target point cloud with an approximated extension. For this extension length, we have implemented the following strategy:

$$r = \max(\alpha h_b, \frac{\beta h_b}{N^{1/d}}, r_u) \quad (4.11)$$

Where h_b is the longest edge in the cartesian bounding box, N is the number of nodes, d is the topological dimension, and r_u is user-provided radius extension. Notice that the second term essentially tries to estimate the average cell-size of the point cloud.

Note: We choose $\alpha = 0.1$ (10%) and $\beta = 5$ as default parameters.

4.3 Adaptation with Non-smooth Functions

A common challenge of solution/data remapping is to handle non-smooth functions, i.e. avoiding [Gibbs phenomenon](#). We apply a simple bounding strategy that enforces the transferred solutions are bounded locally with respect to the sources values in the stencil \mathcal{J} .

Given the intermediate solutions \hat{f}_t after applying the `WLS` fitting, then the following limitation is performed:

$$f_i^t = \begin{cases} \hat{f}_i^t & \hat{f}_i^t \in \text{range}(\mathbf{f}_{\mathcal{J}^s}) \\ \max(\mathbf{f}_{\mathcal{J}^s}) & \hat{f}_i^t \approx \max \\ \min(\mathbf{f}_{\mathcal{J}^s}) & \text{ow} \end{cases} \quad (4.12)$$

In solution transfer, the strategy above may crop high-order values to first order in smooth regions, this is particularly true when the two grids resolutions differ too much from each other. Therefore, we need a mechanism that can pre-screen the non-smooth regions. We utilize a gradient based smoothness indicator, which is similar to those used in WENO schemes.

$$\frac{|\hat{f}_i^t - f_1^s|}{\epsilon h} \leq \sigma \quad (4.13)$$

where $\epsilon = \max_{j \in \mathcal{J}}(|f_j^s|)$. This scheme is efficient and doesn't require geometry data. The drawback is also obvious—it's too simple and tuning σ is not easy.

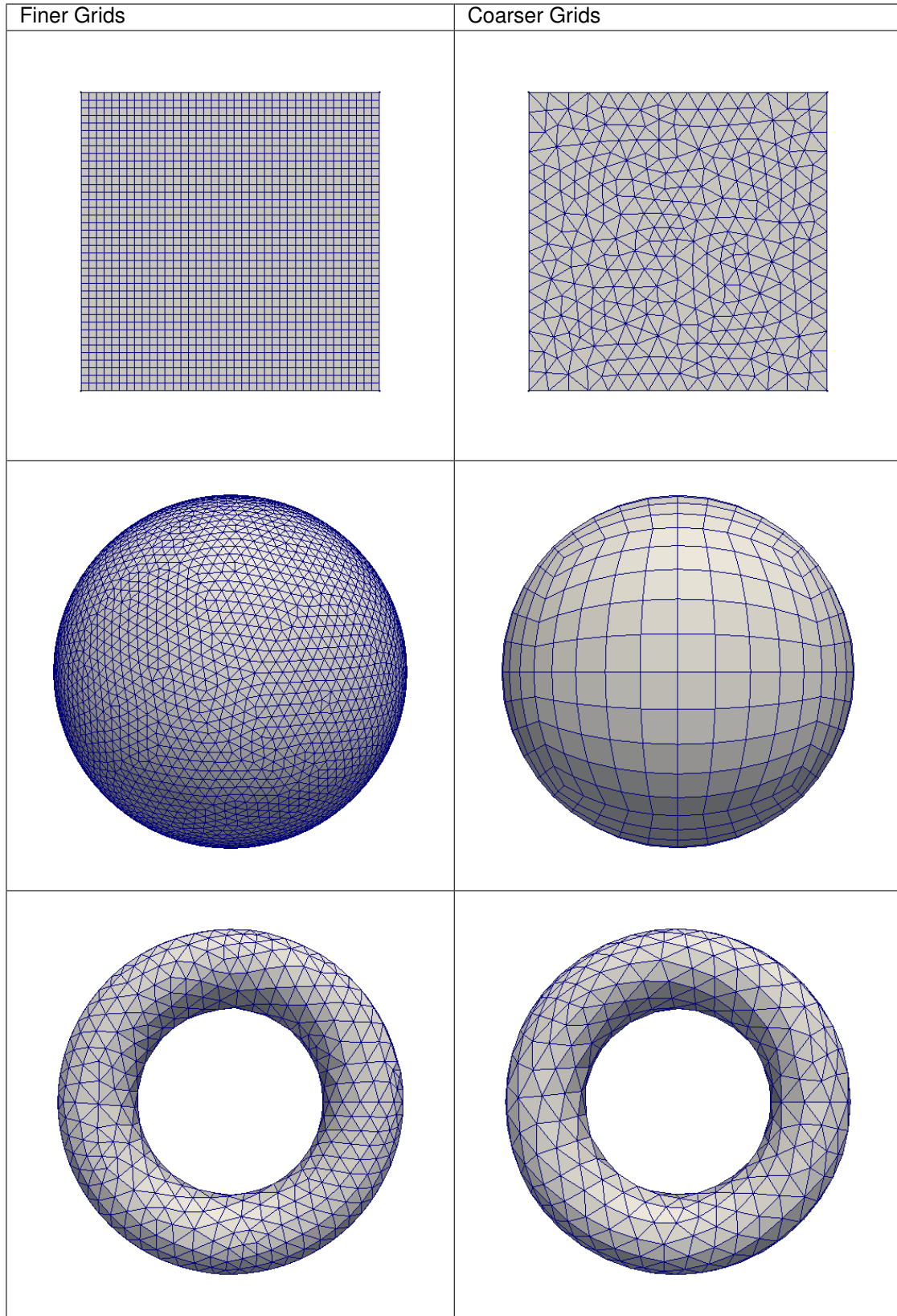
4.4 Results

4.4.1 Convergence Tests

We have tested the AWLS on the following settings:

1. plane surface,
2. spherical surface, and
3. torus surface.

For the plane surface, we use structured grids on one side and triangular meshes on the other side. For setting 2, we use *cubed-sphere* grids and triangular meshes. For the last case, we use triangular meshes with different resolutions. All grids are uniformly refined by three levels in order to study the convergences. Notice that only the coarsest levels are shown in the following figures and all tests are transferred from fine grids to the corresponding coarse ones.



The convergence metric is:

$$c = \left| \frac{\log_2(e_3/e_1)}{\log_2(h_3/h_1)} \right|$$

Where h is some consistent measures of the grids. The error metric we use is relative ℓ_2 errors:

$$e = \frac{\|f_h - f\|_2}{\|f\|_2}$$

For the plane surface, the following two models are tested:

1. $f(x, y) = e^{x+y}$, and
2. $f(x, y) = \sin(\frac{\pi x}{2}) \cos(\frac{\pi y}{2})$.

We use [WENDLAND21](#) as weighting scheme and choose $\rho = 3$ (18 points in stencils), the results are:

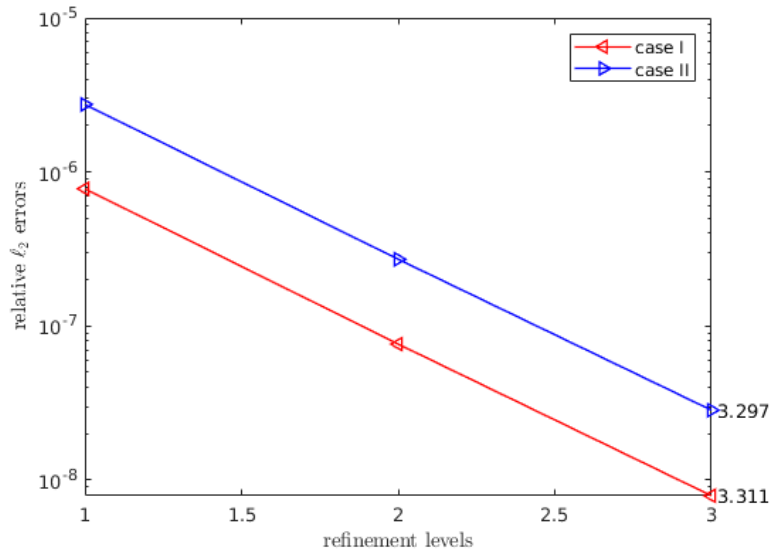


Fig. 1: Convergence of the plane surface

For the 3D cases, we choose the following models:

1. $f(x, y, z) = (\sin(x) + \cos(y))z$, and
2. $f(x, y, z) = e^{x+y+z}$.

We use [WENDLAND21](#) as weighting scheme and choose $\rho = 3.2$ (32 points in stencils) for the spherical surface and $\rho = 2.3$ (23 points in the stencils) for the torus surface, the results are:

Note that for all cases, the super-convergence phenomenon is observed, i.e. the convergence rate is greater than $(p + 1)$ -st order.

The torus program script can be obtained [here](#), and the corresponding grids are stored in compressed npz file that can be obtained [here](#).

Since the spherical surface is really special due to its smoothness and symmetry, an almost- $(p + 2)$ -nd order super-convergence can be obtained with large stencils and [WU2](#) weighting schemes. The following results are with $\rho = 6.8$ (68 points in stencils):

However, this is less practical and hard to apply on general surfaces. As a matter of fact, we didn't observe this with the torus setting.

You can obtain the test grids in VTK [here](#).

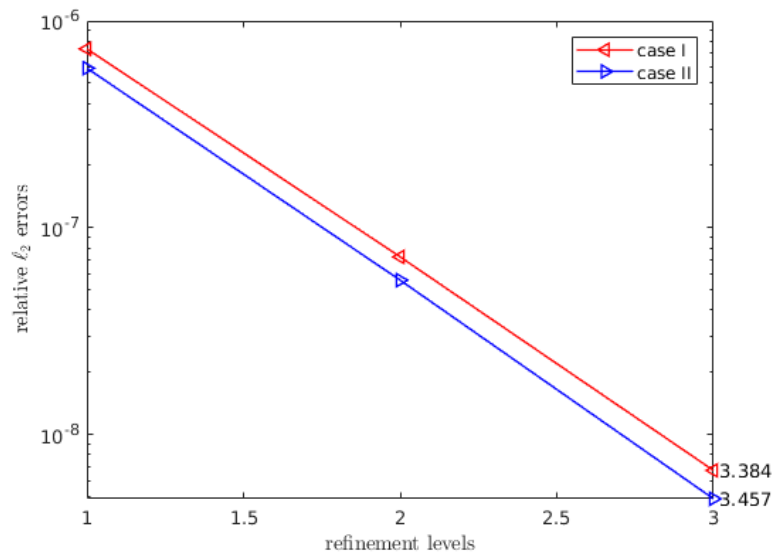


Fig. 2: Convergence of the spherical surface

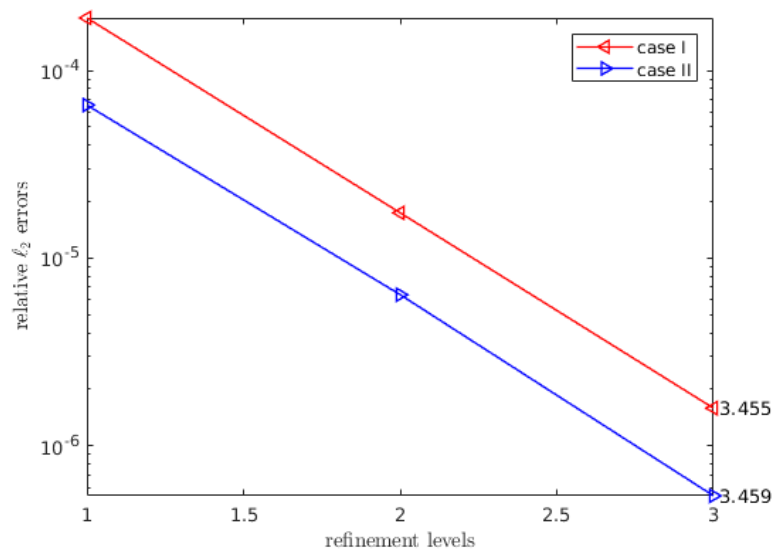
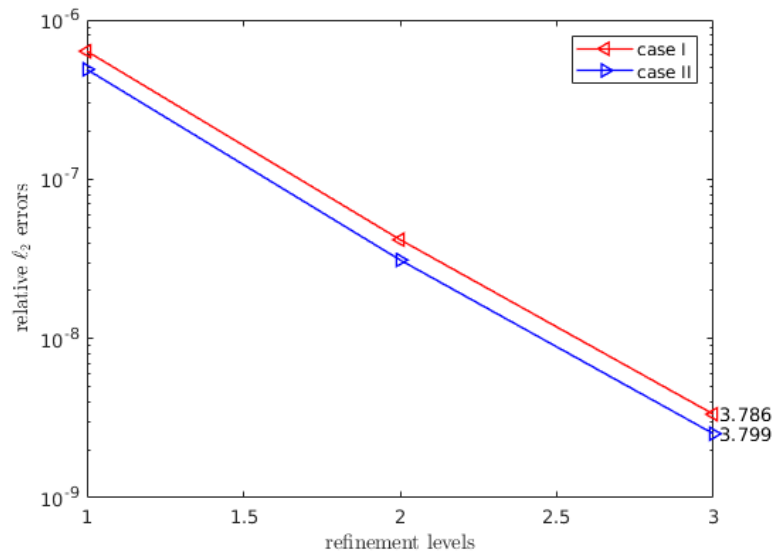


Fig. 3: Convergence of the torus surface



4.4.2 Non-smooth Functions

As a preliminary report, we perform the discontinuous test with the heaviside function on the plane surface with $\sigma = 2$. The results are:

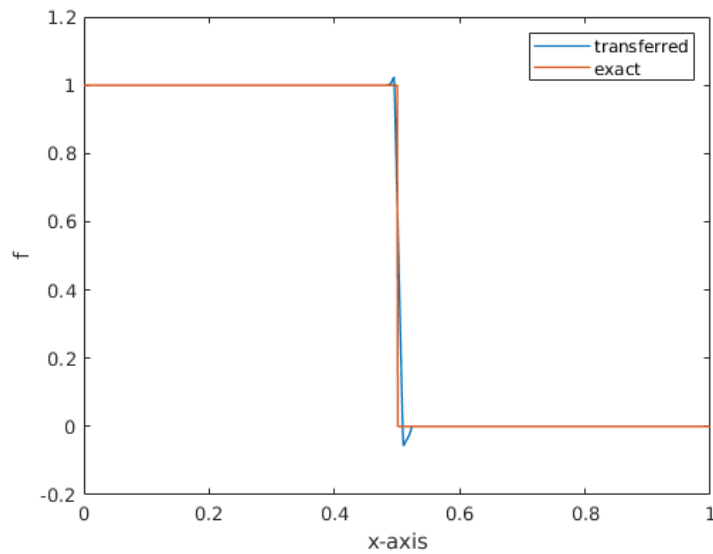


Fig. 4: Heaviside function w/o resolving non-smooth regions

4.5 Usage

In order to use AWLS, you need to *install* the DTK2 package from [my personal forked version](#).

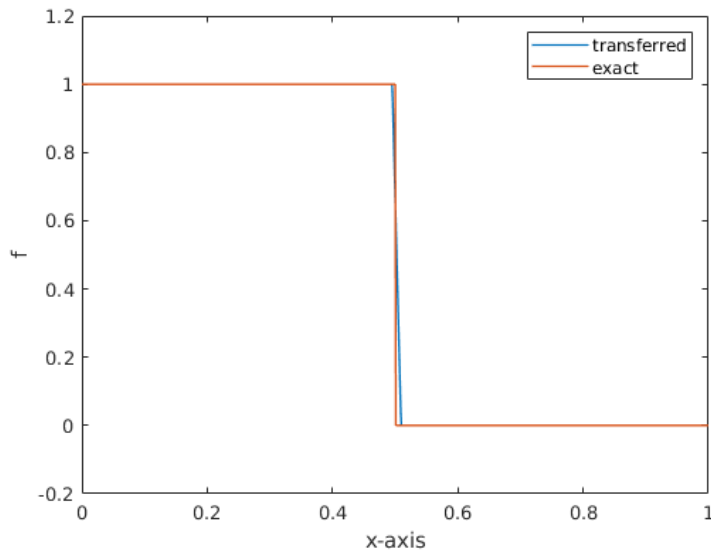


Fig. 5: Heaviside function with resolving non-smooth regions

Note: I didn't decide to make a PR due to DTK2 is probably deprecated in the official repository???

To determine the backend DTK2, a static member function is implemented:

```
>>> from parpydtk2 import *
>>> assert not Mapper.is_dtk2_backend()
```

To enable AWLS support, you just need to call `awls_conf()`:

```
>>> mapper.awls_conf()
```

The default settings are:

1. `method` is set to *AWLS*,
2. `basis` is set to *WENDLAND21*,
3. α is set to 0.1 in (4.11),
4. β is set to 5 in (4.11),
5. ρ is set to 3 for *choosing local stencils*

A complete list of parameters are:

```
>>> mapper.awls_conf(
...     ref_r_b=...,
...     ref_r_g=...,
...     dim=...,
...     alpha=...,
...     beta=...,
...     basis=...,
...     rho=...,
...     verbose=True,
... )
```

Where `ref_r_b` and `ref_r_g` are r_u in (4.11) for blue and green participants. `dim` is the topological dimension used in (4.11), which is important if you need to transfer surface data (90% cases) in 3D space in order to estimate the proper of mesh cell sizes.

In order to resolve discontinuous solutions, you need to pass in at least one additional parameter `resolve_disc` to `transfer_data()`:

```
>>> bf_tag, gf_tag = 'foo', 'bar'
>>> mapper.transfer_data(bf_tag, gf_tag, direct=B2G, resolve_disc=True)
```

The default is σ in (4.13) is 2. To use another value, simply do:

```
>>> mapper.transfer_data(..., resolve_disc=True, sigma=...)
```

Note: Resolving discontinuous solutions only works with *AWLS* method under CHIAO45 backend.

A DEMO

Here, we show a demo for transferring solutions between two meshes of the unit square. We let the blue mesh participant run in parallel with two cores, while the green side is treated as a serial mesh.

```
1 import numpy as np
2 from mpi4py import MPI
3 from parpydtk2 import *
4
5 comm = MPI.COMM_WORLD
6
7 blue, green = create_imeshdb_pair(comm)
8
9 assert comm.size == 2
10 rank = comm.rank
```

For demo purpose, we construct the meshes globally.

```
12 # create blue meshes on all processes
13 cob = np.empty(shape=(16, 3), dtype=float, order='C')
14 dh = 0.3333333333333333
15 index = 0
16 x = 0.0
17 for i in range(4):
18     y = 0.0
19     for j in range(4):
20         cob[index] = np.asarray([x, y, 0.0])
21         index += 1
22         y += dh
23     x += dh
24
25 # global IDs, one based
26 bgids = np.arange(16, dtype='int32')
27 bgids += 1
```

The blue side has 16 nodes, the following is for the green side:

```
29 # create green meshes on all processes
30 cog = np.empty(shape=(36, 3), dtype=float, order='C')
31 dh = 0.2
32 index = 0
33 x = 0.0
34 for i in range(6):
35     y = 0.0
36     for j in range(6):
37         cog[index] = np.asarray([x, y, 0.0])
```

(continues on next page)

(continued from previous page)

```

38         index += 1
39         y += dh
40         x += dh
41
42     # global IDs
43     ggids = np.arange(36, dtype='int32')
44     ggids += 1

```

The green participant has 36 nodes. The next step is to put the data in the two mesh databases.

```

46 # creating the mesh database
47 blue.begin_create()
48 # local vertices and global IDs
49 lcob = cob[8 * rank:8 * (rank + 1), :].copy()
50 lbuids = buids[8 * rank:8 * (rank + 1)].copy()
51 blue.create_vertices(lcob)
52 blue.assign_uids(lbuids)
53 # do not use trivial global ID strategy
54 blue.finish_create(False)
55 blue.create_field('b')

```

As we can see, we equally distributed the mesh into the two cores as well as the corresponding global IDs. In line 54, the `False` flag indicates that the mesh database should use the user-provided global IDs.

Warning: Creating vertices and assigning global IDs must be called between `begin_create()` and `finish_create()`! Otherwise, exceptions are thrown.

Warning: Creating fields must be done after `finish_create()`!

Here is the treatment for the “serial” participant:

```

58 # NOTE that green is assumed to be serial mesh
59 green.begin_create()
60 # only create on master rank
61 if not rank:
62     green.create_vertices(cob)
63 # since green is serial, we just use the trivial global IDs
64 green.finish_create() # empty partition is resolve here
65 green.create_field('g') # must after finish create
66
67 assert green.has_empty()

```

As we can see, only the master process has data.

Note: The *duplicated* node is handled inside `finish_create()`

With the two participants ready, we can now create our *Mapper*.

```

69 # create our analytical model, i.e. 10+sin(x)*cos(y)
70 bf = 10.0 + np.sin(lcob[:, 0]) * np.cos(lcob[:, 1])
71 gf = np.sin(cog[:, 0]) * np.cos(cog[:, 1]) + 10.0

```

(continues on next page)

(continued from previous page)

```

72
73 # Construct our mapper
74 mapper = Mapper(blue=blue, green=green)
75
76 # using mmls, blue radius 1.0 green radius 0.6
77 mapper.method = MMLS
78 mapper.radius_b = 1.0
79 mapper.radius_g = 0.6
80 mapper.dimension = 2
81
82 # Mapper initialization region
83 mapper.begin_initialization()
84 mapper.register_coupling_fields(bf='b', gf='g', direct=B2G)
85 mapper.register_coupling_fields(bf='b', gf='g', direct=G2B)
86 mapper.end_initialization()

```

Line 69-70 just create an analytic model for error analysis. Line 77-80 are for parameters, for this case, we use MMLS (default) with blue side radius 1.0 and green side radius 0.6 for searching.

The important part is from line 83 to 86. Particularly speaking, the function `register_coupling_fields()`. It takes three parameters, where the first two are string tokens that represents the data fields in blue and green. The `direct` is to indicate the transfer direction, e.g. `B2G` stands for blue to green.

Warning: `register_coupling_fields()` must be called within `begin_initialization()` and `end_initialization()`.

```

88 # NOTE that the following only runs on green master mesh
89 if not rank:
90     green.assign_field('g', gf)
91 # Since green is serial and has empty partition, we must call this to
92 # resolve asynchronous values
93 green.resolve_empty_partitions('g')

```

The above section is to assign values on the green participant. Notice that it is a “serial” mesh, so we only assign values on the master process. But resolve the *duplicated* node is needed, this is done in line 93.

Finally, the solution transfer part is pretty straightforward:

```

94 # solution transfer region
95 mapper.begin_transfer()
96 mapper.transfer_data(bf='b', gf='g', direct=G2B)
97 err_b = (bf - blue.extract_field('b'))/10
98 mapper.transfer_data(bf='b', gf='g', direct=B2G)
99 err_g = (gf - green.extract_field('g'))/10
100 mapper.end_transfer()
101
102 comm.barrier()
103
104 print(rank, 'blue L2-error=%.3e' % (np.linalg.norm(err_b)/np.sqrt(err_b.size)))
105 if rank == 0:
106     print(0, 'green L2-error=%.3e' % (np.linalg.norm(err_g)/np.sqrt(err_g.size)))

```

This code can be obtained [here parallel2serial.py](#).

6.1 ParPyDTK2 Python API

6.1.1 Default

Main module interface of ParPyDTK2

`parpydtk2.B2G`

bool – boolean flag of `True` denotes transferring direction from blue to green

`parpydtk2.G2B`

bool – boolean flag of `False` denotes transferring direction from green to blue

`parpydtk2.MMLS`

int – flag (0) represents using *modified moving least square* method

`parpydtk2.SPLINE`

int – flag (1) represents using *spline interpolation* method

`parpydtk2.N2N`

int – flag (2) represents using *nearest node projection* method

`parpydtk2.AWLS`

int – flag (3) represents using *adaptive weighted least square* method

`parpydtk2.N2N_MATCH`

int – flag (4) represents using *matching n2n* method

`parpydtk2.WENDLAND2`

int – flag (0) represents using *Wendland 2nd-order* RBF weights

`parpydtk2.WENDLAND4`

int – flag (1) represents using *Wendland 4th-order* RBF weights

`parpydtk2.WENDLAND6`

int – flag (2) represents using *Wendland 6th-order* RBF weights

`parpydtk2.WU2`

int – flag (3) represents using *Wu 2nd-order* RBF weights

`parpydtk2.WU4`

int – flag (4) represents using *Wu 4th-order* RBF weights

`parpydtk2.WU6`

int – flag (5) represents using *Wu 6th-order* RBF weights

`parpydtk2.BUHMANN3`

int – flag (6) represents using *Buhmann 3rd-order* RBF weights

`parpydtk2.WENDLAND21`

int – flag (7) represents using *Wendland 2nd-order 1st dimension* RBF weights

`parpydtk2.create_imeshdb_pair(comm=None)`

Create a pair of interface mesh databases

This is a convenient and safe way to create a pair of imeshdbs, i.e. blue and green participants with a unified communicator. Please use this API instead of directly creating *IMeshDB*.

Parameters `comm` (*MPI.Comm (optional)*) – MPI communicator, default is `None` or `MPI_COMM_WORLD`.

Returns blue and green participants

Return type tuple of *IMeshDB*

Examples

Create mesh databases with `MPI_COMM_WORLD`

```
>>> from parpydtk2 import *
>>> blue, green = create_imeshdb_pair()
```

Create mesh databases with explicit communicator

```
>>> from mpi4py import *
>>> from parpydtk2 import *
>>> comm = MPI.COMM_WORLD
>>> blue, green = create_imeshdb_pair(comm)
```

`parpydtk2.get_include()`

Get the abs include path

6.1.2 Error Handling

The error handler module

`parpydtk2.error_handle.ERROR_CODE`

int – set this to nonzero values one exceptions have been raised

Examples

```
>>> import parpydtk2 as dtk
>>> try:
...     # your programs here
... except Exception:
...     dtk.error.ERROR_CODE = 1
...     raise
```

6.1.3 Interface Mesh & Mapper

`class parpydtk2.IMeshDB(comm=None)`

Interface mesh database

ParPyDTK2 utilizes MOAB as the underlying mesh database. MOAB is an array based mesh library that is adapted by DTK2. With array based mesh library, the memory usage and computational cost are lower than typical pointer based data structure. The mesh concept in this work is simple since only meshless methods are utilized, the only additional attribute one needs is the *global IDs/handles*, which are used by both MOAB and DTK2. For most applications, the global IDs can be computed offline.

One thing is not directly supported by IMeshDB is I/O. However, since this is a Python module and only points clouds are needed, one can easily use a tool (e.g. [meshio](#)) to load the mesh.

comm
MPI.Comm – MPI communicator

ranks
int – size of comm

rank
int – rank of comm

size
int – point cloud size, i.e. number of vertices

gsize
int – global point cloud size

bbox
np.ndarray – local bounding box array of shape (2,3)

gbbox
np.ndarray – global bounding box array of shape (2,3)

Constructor

Parameters **comm** (*MPI.Comm* (optional)) – if no communicator or None is passed in, then `MPI_COMM_WORLD` will be used

Examples

```
>>> # implicit communciator
>>> import parpydtk2 as dtk
>>> mdb = dtk.IMeshDB()
```

```
>>> # explicit communicator
>>> from mpi4py import MPI
>>> import parpydtk2 as dtk
>>> mdb = dtk.IMeshDB(MPI.COMM_WOLRD)
```

Notes

Since the mesh database participants appear at least in pairs, a preferred way to construct IMeshDB is to use the wrapper API `create_imeshdb_pair()`, e.g.

```
>>> from parpydtk2 import *
>>> blue, green = create_imeshdb_pair()
```

assign_field (*self, unicode field_name, ndarray values*)
 Assign values to a field

Note: *values* size must be at least $\text{size} \times \text{dim}$

Parameters

- **field_name** (*str*) – name of the field
- **values** (*np.ndarray*) – input source values

See also:

[`extract_field\(\)`](#) extract value from a field

[`size`](#) check the size of a mesh set

assign_gids (*self*, *__Pyx_memviewslice gids*)

Assign global IDs

Internally, both DTK and MOAB use so-called global IDs/handles communications. Each node has its own local IDs/handles and a unique global ID.

Parameters **gids** (*np.ndarray*) – global IDs

See also:

[`create_vertices\(\)`](#) create vertices

[`extract_gids\(\)`](#) extract global IDs

bbox

np.ndarray – local bounding box

The bounding box is stored simply in a 2x3 array, where the first row stores the maximum bounds while minimum bounds for the second row.

Warning: Bounding box is valid only after `finish_create()`.

See also:

[`gbbox`](#) global bounding box

begin_create (*self*)

Begin to create/manipulate the mesh

This function must be called in order to let the mesh database be aware that you will create meshes.

See also:

[`finish_create\(\)`](#) finish creating mesh

comm

MPI.Comm – communicator

create_field (*self*, *unicode field_name*, *int dim=1*)

Create a data field for solution transfer

This is the core function to register a field so that you can then transfer its values to other domains. The *dim* parameter determines the data type of the field. By default, it's 1, i.e. scalar fields. For each node, a tensor

of (1x“dim“) can be registered. For instance, to transfer forces and displacements in FSI applications, `dim` is 3 (for 3D problems).

Parameters

- **field_name** (*str*) – name of the field
- **dim** (*int*) – dimension of the field, i.e. scalar, vector, tensor

Examples

```
>>> from parpydtk2 import *
>>> mdb1 = IMeshDB()
>>> mdb1.begin_create()
>>> # creating the meshdb
>>> mdb1.finish_create()
>>> mdb1.create_field('heat flux')
```

create_vertices (*self*, *__Pyx_memviewslice* *coords*)
Create a set of coordinates

Note: The *coords* must be C-ordering with *ndim*=2!

Parameters **coords** (*np.ndarray*) – nx3 coordinates in double precision

See also:

assign_gids () assign global IDs

extract_vertices () extract vertex coordinates

Examples

```
>>> from parpydtk2 import *
>>> import numpy as np
>>> mdb1 = IMeshDB()
>>> mdb1.begin_create()
>>> verts = np.zeros((2,3)) # two nodes
>>> verts[1][0] = 1.0
>>> mdb1.create_vertices(verts)
```

created (*self*)
Check if the mesh database has been created or not

Returns True if *finish_create* () has been called

Return type bool

empty (*self*)
Check if this is an empty partition

extract_field (*self*, *unicode* *field_name*, *__Pyx_memviewslice* *buffer=None*, *reshape=False*)
Extact the values from a field

Warning: if *buffer* is passed in, it must be 1D

Parameters

- **field_name** (*str*) – name of the field
- **buffer** (*np.ndarray*) – 1D buffer
- **reshape** (*bool*) – *True* if we reshape the output, only for vectors/tensors

Returns field data values

Return type np.ndarray

extract_gids (*self*)
Extract global IDs/handles

Warning: This function should be called once you have finished *assign_gids()*.

Returns array of *size* that stores the integer IDs

Return type np.ndarray

extract_vertices (*self*)
Extract coordinate

Warning: This function should be called once you have finished *create_vertices()*.

Returns (nx3) array that stores the coordinate values

Return type np.ndarray

See also:

size get the mesh size

field_dim (*self, unicode field_name*)
Check the field data dimension

Parameters **field_name** (*str*) – name of the field

Returns data field dimension of *field_name*

Return type int

finish_create (*self, trivial_gid=True*)
finish mesh creation

This method finalizes the interface mesh database by communicating the bounding boxes and empty partitions. Also, setting up the DTK managers happens here.

Warning: You must call this function once you have done with manipulating the mesh, i.e. vertices and global IDs.

Parameters **trivial_gid** (*bool*) – *True* if we use MOAB trivial global ID computation

Notes

By `trivial_gid`, it means simply assigning the global IDs based on the size of the mesh. This is useful in serial settings or transferring solutions from a serial solver to a partitioned one.

gbbox

np.ndarray – global bounding box

The bounding box is stored simply in a 2x3 array, where the first row stores the maximum bounds while minimum bounds for the second row.

Warning: Bounding box is valid only after `finish_create()`.

See also:

bbox local bounding box

gsize

int – Get the global point cloud size

has_empty(self)

Check if an empty partition exists

has_field(self, unicode field_name)

Check if a field exists

Parameters **field_name** (*str*) – name of the field

Returns *True* if this meshdb has *field_name*

Return type *bool*

rank

int – get the rank

ranks

int – Get the communicator size

resolve_empty_partitions(self, unicode field_name)

Resolve asynchronous values on empty partitions

ParPyDTK2 doesn't expect `assign_field()` should be called collectively. Therefore, a collective call must be made for resolving empty partitions.

Warning: This must be called collectively even on empty partitions

Note: You should call this function following assignment

Parameters **field_name** (*str*) – name of the field

Examples

```
>>> if rank == 0:
...     mdb.assign_field('flux', values)
>>> mdb.resolve_empty_partitions('flux')
```

Notes

This API is not available in C++ level, therefore, one needs to implement this if he/she wants to use the C++ API.

size

int – Get the size of a set

class parpydtk2.**Mapper** (*blue, green, profiling=True, verbose=True, **kwargs*)
DTK2 wrapper

The meshless methods in DTK2, including *modified moving least square*, *spline interpolation* and *nearest node projection* methods are wrapped within this class.

In addition, if you build DTK2 from UNIFEM/CHIAO45 forked versions, then you can use the *adaptive weighted least square* fitting method.

blue_mesh

IMeshDB – blue mesh participant

green_mesh

IMeshDB – green mesh participant

comm

MPI.Comm – MPI communicator

ranks

int – size of comm

rank

int – rank of comm

dimension

int – spatial dimension

method

int – method flag, either MMLS, SPLINE, or N2N

basis

int – flag of basis function for weighting schemes used by MMLS and SPLINE

radius_b

float – radius used for searching on blue_mesh

radius_g

float – radius used for searching on green_mesh

leaf_b

int – kd-tree leaf size of blue_mesh

leaf_g

int – kd-tree leaf size of green_mesh

rho

double – local Vandermonde system row scaling factor

Constructor

Parameters

- **blue_mesh** (*IMeshDB*) – blue mesh participant
- **green_mesh** (*IMeshDB*) – green mesh participant
- **profiling** (*bool (optional)*) – whether or not do timing report, default is `True`.
- **verbose** (*bool (optional)*) – whether or not verbose printing, default is `True`.
- **stat_file** (*str (optional)*) – file for storing profiling information

Examples

```
>>> from parpydtk2 import *
>>> blue, green = IMeshDB(), IMeshDB()
>>> # initialize blue and green
>>> mapper = Mapper(blue=blue, green=green)
>>> # do work with mapper
```

awls_conf (*self, **kwargs*)

Configuration of Adaptive Weighted Least Square Fitting

Parameters

- **basis** (*int (optional)*) – basis weighting scheme, default is WENDLAND21
- **rho** (*float (optional)*) – number of rows in the local Vandermonde system, i.e. $\rho \times \text{col}$
- **ref_r_b** (*float (optional)*) – reference user-specified blue radius, i.e. r_u for blue
- **ref_r_g** (*float (optional)*) – reference user-specified green radius, i.e. r_u for green
- **dim** (*int (optional)*) – topological dimension, default is the surface dimension
- **verbose** (*bool (optional)*) – print verbose information/warning messages, default is `False`
- **alpha** (*float*) – the α parameter
- **beta** (*float*) – the β parameter
- **_ind_file** (*str (optional)*) – indicator result file, used in chiao45 dtk

Notes

`_ind_file` is for internal use to fine tune the parameter `sigma`. It will not be enabled in release build.

See also:

`enable_mmls_auto_conf()` configure radius for parallel

`mesh()`

basis

int – Get the basis function flag

See also:

method get the method tag

begin_initialization (*self*, ***kwargs*)

Initialization starter

This is a must-call function in order to indicate mapper that you are about to initialize/register coupling fields

See also:

register_coupling_fields() register coupled fields

end_initialization() finish initialization

begin_transfer (*self*, ***kwargs*)

Transfer starter

This is a must-call function to inidate the beginning of a transferring block

See also:

end_transfer() transfer closer

tranfer_data() transfer a coupled data fields

blue_mesh

IMeshDB – blue mesh

comm

MPI.Comm – Get the communicator

See also:

ranks get the total communicator size

rank “my” rank

dimension

int – Get the problem dimension

Note: this is the spacial dimension

enable_mmls_auto_conf (*self*, ***, *ref_r_b=None*, *ref_r_g=None*, *dim=None*, *verbose=False*, ***kwargs*)

Automatically set up radius parameter for MMLS

Warning: This method should be used only when the underlying DTK2 installation is from CHIAO45 forked version. Otherwise, you should **always** manully configure the radius parameters.

The following strategy is performed:

$$r = \max(\alpha h_b, \frac{\beta h_b}{N^{1/d}}, r_u)$$

where h_b is the the the maximum edge length of the global bounding box (*gbbox*); α is some ratio, say 0.1 (10%); β is the scaling factor the the estimated mesh size, which is given by $\frac{h_b}{N^{1/(d-1)}}$ and N is the

global mesh size (*gsize*); *d* is the spatial dimension; the last parameter r_u is provided by the user thus optional.

For UNIFEM/CHIAO45 DTK2, this parameter should be relatively large, because the final points in the Vandermonde system is determined by the column size, so that larger radius means that the system has larger candidate pool.

Warning: Regarding the spatial dimension, since this packages is mainly for interface/surface coupling thus the actual dimension is assumed to be one less than the spatial dimension, i.e. surface topological dimension. If this is not the case, the user needs to explicit pass in the dimension to override this default behavior.

Parameters

- **ref_r_b**(*float* (*optional*)) – reference user-specified blue radius, i.e. r_u for blue
- **ref_r_g**(*float* (*optional*)) – reference user-specified green radius, i.e. r_u for green
- **dim**(*int* (*optional*)) – topological dimension, default is the surface dimension
- **verbose**(*bool* (*optional*)) – print verbose information/warning messages, default is False
- **alpha**(*float*) – the α parameter
- **beta**(*float*) – the β parameter

See also:

[`is_dtk2_backend\(\)`](#) check backend installation of DTK2

end_initialization(*self*, ***kwargs*)

Initialization closer

This is a must-call function in order to tell the mapper we are ready

See also:

[`begin_initialization\(\)`](#) initialization starter

end_transfer(*self*, ***kwargs*)

Transfer closer

This is a must-call function to indicate we have finished a sequence of transferring requests

See also:

[`begin_transfer\(\)`](#) transfer starter

green_mesh

[`IMeshDB`](#) – green mesh

has_coupling_fields(*self*, *unicode bf*, *unicode gf*, *bool direct*)

Check if a coupled fields exists

Returns True if (bf,gf) exists in the `direct` direction

Return type bool

static is_dtk2_backend()

Check if the underlying DTK2 library is chiao45 forked version

Returns `False` if the user's DTK2 is from chiao45 forked repo

Return type `bool`

leaf_b

int – get the leaf size of blue mesh for kd-tree

Warning: This attribute only works when the underlying DTK2 is installed from UNIFEM or CHIAO45 forked versions.

See also:

leaf_g green leaf size of kd-tree

leaf_g

int – get the leaf size of the green mesh for kd-tree

Warning: This attribute only works when the underlying DTK2 is installed from CHIAO45 forked versions.

See also:

leaf_b blue leaf size of kd-tree

method

int – Get the method tag

See also:

basis the basis function and order attribute

static parameter_keys()

radius_b

float – physical domain radius support for blue mesh

Note: if blue does not use RBF-search, then -1.0 returned

See also:

radius_g green radius

radius_g

float – physical domain radius support for green mesh

Note: if green does not use RBF-search, then -1.0 returned

See also:

`radius_b` blue radius

rank

`int` – Check “my” rank

See also:

`ranks` get the total communicator size

`comm` MPI communicator

ranks

`int` – Check the total process number

See also:

`rank` “my” rank

register_coupling_fields (*self*, *unicode bf*, *unicode gf*, *bool direct*, ***kwargs*)
register a coupled fields

Note: we use boolean to indicate direction

Parameters

- **bf** (*str*) – blue mesh field name
- **gf** (*str*) – green mesh field name
- **direct** (*bool*) – *True* for blue->green, *False* for the opposite

See also:

`transfer_data()` transfer a coupled data fields

rho

float – local scaling factor

set_matching_flag_n2n (*self*, *bool matching*)
Set the matching flag for N2N

Note: this function will not throw even if you dont use n2n

Parameters **matching** (*bool*) – *True* if the interfaces are matching

transfer_data (*self*, *unicode bf*, *unicode gf*, *bool direct*, ***kwargs*)
Transfer (bf, gf) in the *direct* direction

Note: Parameters `resolve_disc` and `sigma` only work with UNIFEM/CHIAO45 DTK and AWLS method.

Parameters

- **bf** (*str*) – blue mesh field name
- **gf** (*str*) – green mesh field name
- **direct** (*bool*) – True for blue->green, False for the opposite
- **resolve_disc** (*bool (optional)*) – True if do post-processing for resolving non-smooth functions
- **sigma** (*float (optional)*) – threshold used in smoothness indicator

See also:

`register_coupling_fields()` register coupled fields

6.2 ParPyDTK2 C++ API

6.2.1 Common Definitions

group **common**

Defines

handle_moab_error (*__ret*)
macro to handle moab error

throw_error (*__msg*)
throw runtime_error exception

throw_error_if (*__cond, __msg*)
conditionally error throw

throw_noimpl (*__what*)
throw not implemented feature error

throw_noimpl_if (*__cond, __what*)
throw not implemented feature error with condition

show_warning (*__msg*)
log warning message in stderr

show_warning_if (*__cond, __msg*)
log warning with condition

show_experimental (*__msg*)
log experimental warning in stderr

show_experimental_if (*__cond, __msg*)
log experimental warning in stderr with condition

show_info (*__msg, __rank*)
show information in parallel

show_info_master (*__msg, __rank*)
show information only on master rank

streamer (*__rank*)
streaming message with specific rank

streamer_master (__rank)
streaming messages only on the master process

Typedefs

typedef entity_t
MOAB entity handle.

Enums

enum [anonymous]
root set
Values:
root_set = 0

6.2.2 Field Variables

class FieldData
a representation of MOAB tag for field data

Public Functions

FieldData (moab::Core &*mdb*, **const** std::string &*field_name*, int *dim* = 1)
constructor with moab instance

Note *set* is not the same as mesh set in MOAB

Parameters

- *mdb*: moab instance
- *field_name*: field name
- *dim*: field dimension

void **assign** (**const** moab::Range &*range*, **const** double **values*)
assign values

Parameters

- *range*: entity ranges, for this work, it should be vertices
- *values*: data values, for vector/tensor, C order is expected

void **assign_1st** (**const** moab::Range &*range*, **const** double **values*)
assign to first node

This function is used by Python for handling empty partitions

Parameters

- *range*: entity ranges

- `values`: values for the first node, at least size of `dim`

void **extract** (**const** moab::Range &*range*, double **values*) **const**
extract values

Parameters

- `range`: entity ranges, for this work, it should be vertices
- `values`: data values, for vector/tensor, C order is expected

void **extract_1st** (**const** moab::Range &*range*, double **values*) **const**
extract the value from the first node

Parameters

- `range`: entity ranges
- `values`: values for the first node, at least size of `dim`

operator const std::string&() **const**
brief implicitly cast to string

int **dim**() **const**
check the dimension

int **set**() **const**
check the set ID

const moab::Tag &**tag**() **const**
get MOAB tag

Protected Attributes

moab::Core &**mdb_**
reference to moab instance

std::string **fn_**
field name

int **set_**
set count

int **dim_**
field dimension

moab::Tag **tag_**
moab tag

class FieldDataSet
a set of field data

Public Types

typedef *base_t*::iterator **iterator**
iterator type

```
typedef base_t::const_iterator const_iterator
    constant iterator
```

Public Functions

```
virtual ~FieldDataSet ()
    destructor
```

```
bool has_field(const std::string &fn) const
    check if a field exist
```

Parameters

- *fn*: field name

```
void create(moab::Core &mdb, const std::string &field_name, int dim = 1)
    create an data field
```

Note *set* is not the same as mesh set in MOAB

Parameters

- *mdb*: moab data base
- *field_name*: field name
- *dim*: field dimension

```
FieldData &operator[] (const std::string &fn)
    get a reference to a field data
```

Note this overloads the base operator[]

Parameters

- *fn*: field name

```
const FieldData &operator[] (const std::string &fn) const
    get a const reference to a field data
```

Parameters

- *fn*: field name

```
iterator begin()
    get the first iterator
```

```
iterator end()
    get the end iterator
```

```
const_iterator begin() const
    get the constant iterator
```

```
const_iterator end() const
    get the constant end iterator
```


const_iterator **cbegin** () **const**
get the constant iterator

const_iterator **cend** () **const**
get the constant end iterator

Protected Attributes

base_t **fs_**
fields

Private Types

typedef std::unordered_map<std::string, *FieldData* *> **base_t**
data structure

6.2.3 Interface Mesh Database

class **IMeshDB**
interface mesh database, build on top of MOAB

imesh_py_interface

bool **created** () **const**
check created mesh database

void **begin_create** ()
begin to create mesh

void **create_vset** ()
create a new vertex set

void **create_vertices** (int *nv*, **const** double **coords*)
create vertices

Parameters

- *nv*: number of vertices
- *coords*: coords values

void **extract_vertices** (double **coords*) **const**
extract assigned coordinates

Note *coords* must be at least $n*3$ where n is the size of the mesh

Parameters

- *coords*: coordinates

void **assign_gids** (int *nv*, **const** int **gids*)
assign global IDs

Note *gids* should be one-based indices

Parameters

- *nv*: number of local vertices
- *gids*: global IDs

void **extract_gids** (int **gids*) **const**
extract global IDs

Parameters

- *gids*: global IDs

void **finish_create** (bool *trivial_gid* = true)
finish manipulating the mesh

NOTE that if your input mesh is element-based partition and the vertices you create are mesh nodes, then you have to specify the correct global IDs in parallel. However, if the coordinates are face centres, then the global IDs can be trivially computed by MOAB since there are no shared entities cross different processes.

Parameters

- *trivial_gid*: *true* if we let MOAB to compute the GID

bool **empty** () **const**
check if empty partition

bool **has_empty** () **const**
check if any of the process has an empty partition

const std::vector<int> &**_m2s** () **const**
get a reference to the m2s pattern

Note This is used in Python level as “private” thus having “_”

int **size** () **const**
check mesh size

int **gsize** () **const**
check the global mesh size

void **get_bbox** (double **v*) **const**
get bounding box

Parameters

- *v*: values

void **get_gbbox** (double **v*) **const**
get global bounding box

Parameters

- *v*: values

void **create_field** (**const** std::string &*field_name*, int *dim* = 1)
create a field

Parameters

- `field_name`: field name
- `dim`: field dimension

bool **has_field**(const std::string &*field_name*) const
check if we have a field

Parameters

- `field_name`: field name

int **field_dim**(const std::string &*field_name*) const
check field dimension

Parameters

- `field_name`: field name

void **assign_field**(const std::string &*field_name*, const double **values*)
assign a value to a field

Parameters

- `field_name`: field name
- `values`: field data values

void **_assign_1st**(const std::string &*field_name*, const double **values*)
assign to the first node

This function is used by Python to resolve the issues when empty empty partitions happen. Therefore, this function has an “_” prefix to indicate “private” usage!

Parameters

- `field_name`: field name
- `values`: field data values, at least size of field dimension

void **extract_field**(const std::string &*field_name*, double **values*) const
extract value

Parameters

- `field_name`: field name
- `values`: field data values

void **_extract_1st**(const std::string &*field_name*, double **values*) const
extract first value

This function is used by Python to resolve the issues when empty empty partitions happen. Therefore, this function has an “_” prefix to indicate “private” usage!

Parameters

- `field_name`: field name
- `values`: field data values

Public Functions

IMeshDB (MPI_Comm *comm* = MPI_COMM_WORLD)
 constructor with communicator

Parameters

- *comm*: communicator

int **ranks** () **const**
 get total ranks

int **rank** () **const**
 get my rank

MPI_Comm **comm** () **const**
 get the communicator

Teuchos::RCP<moab::ParallelComm> **pcomm** () **const**
 get mesh

std::vector<DataTransferKit::MoabManager> &**mangers** ()
 get the manger

void **set_dimension** (int *dim*)
 set geometry dimension

Parameters

- *dim*: dimension

bool **ready** () **const**
 check if ready

dtk_field_t &**dtk_fields** ()
 get the dtk fields

Protected Types

typedef std::unordered_map<std::string, std::pair<int, Teuchos::RCP<Tpetra::MultiVector<double, int, DataTransferKit::Sup>>>
 handy typedef

Protected Attributes

moab::Core **mdb_**
 moab instance

Teuchos::RCP<moab::ParallelComm> **par_**
 moab parallel interface

std::vector<*entity_t*> **vsets_**
 vertex sets

std::vector<moab::Range> **locals_**
 local vertex range

std::vector<std::array<double, 6>> **bboxes_**
 bounding boxes

std::vector<std::array<double, 6>> **gbboxes_**
 global bounding boxes

FieldDataSet **fields_**
 field data set

bool **created_**
 flag to indicate whether users are done with creating mesh

moab::Tag **gidtag_**
 global ID tag

moab::Tag **parttag_**
 partition tag

bool **usergid_**
 flag to indicate if we have user computed global ID

std::vector<DataTransferKit::MoabManager> **mngers_**
 DTK MOAB manager.

dtk_field_t **dtkfields_**
 DTK multi vector for MOAB tags.

bool **empty_**
 check empty partition

bool **has_empty_**
 check if any process is empty

std::vector<int> **m2s_**
 comm pattern for master2slaves for handling empty partitions

int **gsize_**
 global size

Private Functions

void **init_** (bool *del* = false)
 helper for clean up mesh

Parameters

- *del*: whether or not delete mesh

void **reset_vecs_** ()
 handle all vectors

void **init_bbox_** (int *i* = -1)
 initialize empty bounding boxes

Parameters

- *i*: index if < 0 then init all

void **cmpt_bboxes_** ()
 compute all bounding box

6.2.4 Solution Mapper

class Mapper

the mapper interface for interface solution transfer

mapper_py_interface

int **ranks** () **const**
get the ranks

int **rank** () **const**
get my rank

MPI_Comm **comm** () **const**
get the communicator

void **set_dimension** (int *dim*)
set dimension

Parameters

- *dim*: geometry dimension

void **use_mmls** ()
use moving least square, this is the default method

See [*use_spline*](#), [*use_n2n*](#)

void **use_spline** ()
use spline interpolation method

See [*use_mmls*](#), [*use_n2n*](#)

void **use_n2n** (bool *matching* = false)
use node 2 node project

See [*use_mmls*](#), [*use_spline*](#)

Parameters

- *matching*: are the interfaces mathing?

void **use_awls** ()
use adaptive weighted least square fitting

See [*use_mmls*](#)

void **set_basis** (int *basis*)
set basis function, default is Wendland 4th order

See [*BasisFunctions*](#)

Parameters

- *basis*: basis function and order

void **use_knn_b** (int)
use knn for blue mesh

void **use_knn_g** (int)
use knn for green mesh

void **use_radius_b** (double *r*)
use radius for blue

See use_knn

Parameters

- *r*: physical domain radius support

void **use_radius_g** (double *r*)
use radius for green

See use_knn

Parameters

- *r*: physical domain radius support

int **check_method** () **const**
check method

int **check_basis** () **const**
check basis

int **knn_b** () **const**
check blue knn

Note if blue does not use knn, then negative value returned

int **knn_g** () **const**
check green knn

double **radius_b** () **const**
check blue radius

Note if blue does not use radius, then -1.0 is returned

double **radius_g** () **const**
check green radius

int **dimension** () **const**
get the dimension

QRCP_ONLY

void **set_leaf_b** (int *size*)
set the leaf size

Warning This method only works with unifem or chiao45 forked backend

Parameters

- `size`: the leaf size in kd-tree

void **set_leaf_g**(int *size*)
set the leaf size for green database

Warning This method only works with unifem or chiao45 forked backend

Parameters

- `size`: the leaf size in kd-tree

int **leaf_b**() **const**
get the blue leaf size

int **leaf_g**() **const**
get the green leaf size

void **_set_ind_file**(**const** std::string &*fn*)
set the indicator tuning filename

Note internal use

Note This only works with QRCP implementation, or AWLS method

Parameters

- `fn`: filename

void **_wipe_ind_file**()
wipe indicator file

void **set_rho**(double *rho*)
set local scaling rho

Note This only works with QRCP implementation, or AWLS method

Parameters

- `rho`: scaling factor

double **rho**() **const**
get local scaling rho

Public Functions

Mapper(std::shared_ptr<*IMeshDB*> *B*, std::shared_ptr<*IMeshDB*> *G*, **const** std::string &*version* = "",
 const std::string &*date* = "", bool *profiling* = true, **const** std::string &*stat_file* = "", bool
 verbose = true)
constructor

Parameters

- *B*: input blue mesh
- *G*: input green mesh

- `version`: passed from Python interface
- `date`: passed from Python interface
- `profiling`: whether do simple profiling, i.e. `wtime`
- `stat_file`: statistics filename
- `verbose`: doing verbose information printing

`std::shared_ptr<IMeshDB> blue_mesh() const`
get blue mesh

`std::shared_ptr<IMeshDB> green_mesh() const`
get green mesh

`void begin_initialization()`
begin initialization

`void register_coupling_fields(const std::string &bf, const std::string &gf, bool direct)`
register coupling fields

Parameters

- `bf`: blue meshdb field data
- `gf`: green meshdb field data
- `direct`: *true* for b->g, *false* for g->b

`bool has_coupling_fields(const std::string &bf, const std::string &gf, bool direct)`
check if a coupling data fields exists

Parameters

- `bf`: blue meshdb field data
- `gf`: green meshdb field data
- `direct`: *true* for b->g, *false* for g->b

`void end_initialization()`
end initialization

`void begin_transfer()`
begin to transfer data

`void transfer_data(const std::string &bf, const std::string &gf, bool direct, bool resolve_disc = false, double sigma = -1.0)`
transfer data

Notice that internally, the unifem backend occupies the `mode` parameter to indicate whether or not do post processing to resolve non-smooth solutions. Set `mode == Teuchos::TRANS`

Parameters

- `bf`: blue meshdb field data
- `gf`: green meshdb field data
- `direct`: *true* for b->g, *false* for g->b
- `resolve_disc`: (optional) if *true*, then try to resolve disc

- `sigma`: (optional) indicator threshold

```
void end_transfer()
    end transfer
```

Public Static Functions

```
static bool is_dtk2_backend()
    check the backend
```

In chiao45 version of DTK2, we modified the exception class to add a prefix of “chiao45”, so it’s feasible to query this information w/o adding a new API

Return if the underlying DTK2 is using in chiao45 forkedversion, then return *true*

Protected Attributes

```
std::shared_ptr<IMeshDB> B_
    blue mesh
```

```
std::shared_ptr<IMeshDB> G_
    green mesh
```

```
int dim_
    dimension
```

```
bool ready_
    flag to indicate the mapper is ready for transferring
```

```
bool profiling_
    whether do simple profiling
```

```
bool verbose_
    verbose output
```

```
std::unique_ptr<Teuchos::ParameterList> opts_[2]
    parameter list
```

```
std::map<std::pair<std::string, std::string>, Teuchos::RCP<DataTransferKit::MapOperator>>> operators_[2]
    transfer operators references
```

```
Teuchos::RCP<DataTransferKit::MapOperator> optrs_[2]
    actual transfer operators
```

```
std::map<std::pair<std::string, std::string>, std::vector<StatInfo>>> info_[2]
    information box for each of the registered pair of fields
```

```
std::map<std::pair<std::string, std::string>, unsigned long> counts_[2]
    total transfer counter
```

```
std::unique_ptr<std::ofstream> stat_
    the statistics file handle
```

```
int stat_write_freq_counter_
    statistics file dump freq counter
```

Protected Static Attributes

DataTransferKit::MapOperatorFactory **factory_**
map factory

const std::string **title_** = std::string(LEN1, '-')
title of verbose printing

const std::string **indentation_** = std::string(LEN2, ' ')
indentation

Private Functions

void **init_parlist_**()
initialize parameter list

template <bool *_Dir*, **typename** *_V*>
void **set_search** (**const** std::string &*type*, **const** std::string &*value*, **const** *_V* &*v*)
helper for set local search

Template Parameters

- *_Dir*: direction
- *_V*: value type

Parameters

- *type*: search type
- *value*: tag for value
- *v*: actual value

template <bool *_Dir*, **typename** *_V*>
_V **get_search** (**const** std::string &*type*, **const** std::string &*value*, **const** *_V* &*dft*) **const**
helper for get search info

Template Parameters

- *_Dir*: direction
- *_V*: value type

Parameters

- *type*: search type
- *value*: tag for value
- *dft*: default value

void **_dump_stats**()
helper function for writing outputs

Private Static Functions

static std::string **parse_list_**(Teuchos::ParameterList &*list*)
parse and formatting a parameter list

Parameters

- *list*: parameter list

struct **StatInfo**

a simple structure to store discontinuous points and timing results

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

p

parpydtk2, [22](#)

parpydtk2.error_handle, [23](#)

A

assign_field() (parpydtk2.IMeshDB method), 24
 assign_gids() (parpydtk2.IMeshDB method), 25
 AWLS (in module parpydtk2), 22
 awls_conf() (parpydtk2.Mapper method), 30

B

B2G (in module parpydtk2), 22
 basis (parpydtk2.Mapper attribute), 29, 30
 bbox (parpydtk2.IMeshDB attribute), 24, 25
 begin_create() (parpydtk2.IMeshDB method), 25
 begin_initialization() (parpydtk2.Mapper method), 31
 begin_transfer() (parpydtk2.Mapper method), 31
 blue_mesh (parpydtk2.Mapper attribute), 29, 31
 BUHMANN3 (in module parpydtk2), 22

C

comm (parpydtk2.IMeshDB attribute), 24, 25
 comm (parpydtk2.Mapper attribute), 29, 31
 common::__anonymous0 (C++ type), 36
 common::root_set (C++ enumerator), 36
 create_field() (parpydtk2.IMeshDB method), 25
 create_imeshdb_pair() (in module parpydtk2), 23
 create_vertices() (parpydtk2.IMeshDB method), 26
 created() (parpydtk2.IMeshDB method), 26

D

dimension (parpydtk2.Mapper attribute), 29, 31

E

empty() (parpydtk2.IMeshDB method), 26
 enable_mmls_auto_conf() (parpydtk2.Mapper method), 31
 end_initialization() (parpydtk2.Mapper method), 32
 end_transfer() (parpydtk2.Mapper method), 32
 ERROR_CODE (in module parpydtk2.error_handle), 23
 extract_field() (parpydtk2.IMeshDB method), 26
 extract_gids() (parpydtk2.IMeshDB method), 27
 extract_vertices() (parpydtk2.IMeshDB method), 27

F

field_dim() (parpydtk2.IMeshDB method), 27

finish_create() (parpydtk2.IMeshDB method), 27

G

G2B (in module parpydtk2), 22
 gbbox (parpydtk2.IMeshDB attribute), 24, 28
 get_include() (in module parpydtk2), 23
 green_mesh (parpydtk2.Mapper attribute), 29, 32
 gsize (parpydtk2.IMeshDB attribute), 24, 28

H

handle_moab_error (C macro), 35
 has_coupling_fields() (parpydtk2.Mapper method), 32
 has_empty() (parpydtk2.IMeshDB method), 28
 has_field() (parpydtk2.IMeshDB method), 28

I

IMeshDB (class in parpydtk2), 23
 is_dtk2_backend() (parpydtk2.Mapper static method), 32

L

leaf_b (parpydtk2.Mapper attribute), 29, 33
 leaf_g (parpydtk2.Mapper attribute), 29, 33

M

Mapper (class in parpydtk2), 29
 method (parpydtk2.Mapper attribute), 29, 33
 MMLS (in module parpydtk2), 22

N

N2N (in module parpydtk2), 22
 N2N_MATCH (in module parpydtk2), 22

P

parameter_keys() (parpydtk2.Mapper static method), 33
 parpydtk2 (module), 22
 parpydtk2.error_handle (module), 23
 parpydtk2::entity_t (C++ type), 36
 parpydtk2::FieldData (C++ class), 36
 parpydtk2::FieldData::assign (C++ function), 36
 parpydtk2::FieldData::assign_1st (C++ function), 36
 parpydtk2::FieldData::dim (C++ function), 37

parpydtk2::FieldData::dim_ (C++ member), 37
 parpydtk2::FieldData::extract (C++ function), 37
 parpydtk2::FieldData::extract_1st (C++ function), 37
 parpydtk2::FieldData::FieldData (C++ function), 36
 parpydtk2::FieldData::fn_ (C++ member), 37
 parpydtk2::FieldData::mdb_ (C++ member), 37
 parpydtk2::FieldData::operator const std::string& (C++ function), 37
 parpydtk2::FieldData::set (C++ function), 37
 parpydtk2::FieldData::set_ (C++ member), 37
 parpydtk2::FieldData::tag (C++ function), 37
 parpydtk2::FieldData::tag_ (C++ member), 37
 parpydtk2::FieldDataSet (C++ class), 37
 parpydtk2::FieldDataSet::~FieldDataSet (C++ function), 38
 parpydtk2::FieldDataSet::base_t (C++ type), 39
 parpydtk2::FieldDataSet::begin (C++ function), 38
 parpydtk2::FieldDataSet::cbegin (C++ function), 38
 parpydtk2::FieldDataSet::cend (C++ function), 39
 parpydtk2::FieldDataSet::const_iterator (C++ type), 37
 parpydtk2::FieldDataSet::create (C++ function), 38
 parpydtk2::FieldDataSet::end (C++ function), 38
 parpydtk2::FieldDataSet::fs_ (C++ member), 39
 parpydtk2::FieldDataSet::has_field (C++ function), 38
 parpydtk2::FieldDataSet::iterator (C++ type), 37
 parpydtk2::FieldDataSet::operator[] (C++ function), 38
 parpydtk2::IMeshDB (C++ class), 39
 parpydtk2::IMeshDB::assign_1st (C++ function), 41
 parpydtk2::IMeshDB::extract_1st (C++ function), 41
 parpydtk2::IMeshDB::m2s (C++ function), 40
 parpydtk2::IMeshDB::assign_field (C++ function), 41
 parpydtk2::IMeshDB::assign_gids (C++ function), 39
 parpydtk2::IMeshDB::bboxes_ (C++ member), 42
 parpydtk2::IMeshDB::begin_create (C++ function), 39
 parpydtk2::IMeshDB::cmpt_bboxes_ (C++ function), 43
 parpydtk2::IMeshDB::comm (C++ function), 42
 parpydtk2::IMeshDB::create_field (C++ function), 40
 parpydtk2::IMeshDB::create_vertices (C++ function), 39
 parpydtk2::IMeshDB::create_vset (C++ function), 39
 parpydtk2::IMeshDB::created (C++ function), 39
 parpydtk2::IMeshDB::created_ (C++ member), 43
 parpydtk2::IMeshDB::dtk_field_t (C++ type), 42
 parpydtk2::IMeshDB::dtk_fields (C++ function), 42
 parpydtk2::IMeshDB::dtkfields_ (C++ member), 43
 parpydtk2::IMeshDB::empty (C++ function), 40
 parpydtk2::IMeshDB::empty_ (C++ member), 43
 parpydtk2::IMeshDB::extract_field (C++ function), 41
 parpydtk2::IMeshDB::extract_gids (C++ function), 40
 parpydtk2::IMeshDB::extract_vertices (C++ function), 39
 parpydtk2::IMeshDB::field_dim (C++ function), 41
 parpydtk2::IMeshDB::fields_ (C++ member), 43
 parpydtk2::IMeshDB::finish_create (C++ function), 40
 parpydtk2::IMeshDB::gbboxes_ (C++ member), 43
 parpydtk2::IMeshDB::get_bbox (C++ function), 40
 parpydtk2::IMeshDB::get_gbbox (C++ function), 40
 parpydtk2::IMeshDB::gidtag_ (C++ member), 43
 parpydtk2::IMeshDB::gsize (C++ function), 40
 parpydtk2::IMeshDB::gsize_ (C++ member), 43
 parpydtk2::IMeshDB::has_empty (C++ function), 40
 parpydtk2::IMeshDB::has_empty_ (C++ member), 43
 parpydtk2::IMeshDB::has_field (C++ function), 41
 parpydtk2::IMeshDB::IMeshDB (C++ function), 42
 parpydtk2::IMeshDB::init_ (C++ function), 43
 parpydtk2::IMeshDB::init_bbox_ (C++ function), 43
 parpydtk2::IMeshDB::locals_ (C++ member), 42
 parpydtk2::IMeshDB::m2s_ (C++ member), 43
 parpydtk2::IMeshDB::mangers (C++ function), 42
 parpydtk2::IMeshDB::mdb_ (C++ member), 42
 parpydtk2::IMeshDB::mngers_ (C++ member), 43
 parpydtk2::IMeshDB::par_ (C++ member), 42
 parpydtk2::IMeshDB::parttag_ (C++ member), 43
 parpydtk2::IMeshDB::pcomm (C++ function), 42
 parpydtk2::IMeshDB::rank (C++ function), 42
 parpydtk2::IMeshDB::ranks (C++ function), 42
 parpydtk2::IMeshDB::ready (C++ function), 42
 parpydtk2::IMeshDB::reset_vecs_ (C++ function), 43
 parpydtk2::IMeshDB::set_dimension (C++ function), 42
 parpydtk2::IMeshDB::size (C++ function), 40
 parpydtk2::IMeshDB::userid_ (C++ member), 43
 parpydtk2::IMeshDB::vsets_ (C++ member), 42
 parpydtk2::Mapper (C++ class), 44
 parpydtk2::Mapper::dump_stats (C++ function), 49
 parpydtk2::Mapper::set_ind_file (C++ function), 46
 parpydtk2::Mapper::wipe_ind_file (C++ function), 46
 parpydtk2::Mapper::B_ (C++ member), 48
 parpydtk2::Mapper::begin_initialization (C++ function), 47
 parpydtk2::Mapper::begin_transfer (C++ function), 47
 parpydtk2::Mapper::blue_mesh (C++ function), 47
 parpydtk2::Mapper::check_basis (C++ function), 45
 parpydtk2::Mapper::check_method (C++ function), 45
 parpydtk2::Mapper::comm (C++ function), 44
 parpydtk2::Mapper::counts_ (C++ member), 48
 parpydtk2::Mapper::dim_ (C++ member), 48
 parpydtk2::Mapper::dimension (C++ function), 45
 parpydtk2::Mapper::end_initialization (C++ function), 47
 parpydtk2::Mapper::end_transfer (C++ function), 48
 parpydtk2::Mapper::factory_ (C++ member), 49
 parpydtk2::Mapper::G_ (C++ member), 48
 parpydtk2::Mapper::get_search (C++ function), 49
 parpydtk2::Mapper::green_mesh (C++ function), 47
 parpydtk2::Mapper::has_coupling_fields (C++ function), 47
 parpydtk2::Mapper::indentation_ (C++ member), 49
 parpydtk2::Mapper::info_ (C++ member), 48
 parpydtk2::Mapper::init_parlist_ (C++ function), 49
 parpydtk2::Mapper::is_dtk2_backend (C++ function), 48

parpydtk2::Mapper::knn_b (C++ function), 45
 parpydtk2::Mapper::knn_g (C++ function), 45
 parpydtk2::Mapper::leaf_b (C++ function), 46
 parpydtk2::Mapper::leaf_g (C++ function), 46
 parpydtk2::Mapper::Mapper (C++ function), 46
 parpydtk2::Mapper::operators_ (C++ member), 48
 parpydtk2::Mapper::optrs_ (C++ member), 48
 parpydtk2::Mapper::opts_ (C++ member), 48
 parpydtk2::Mapper::parse_list_ (C++ function), 50
 parpydtk2::Mapper::profiling_ (C++ member), 48
 parpydtk2::Mapper::radius_b (C++ function), 45
 parpydtk2::Mapper::radius_g (C++ function), 45
 parpydtk2::Mapper::rank (C++ function), 44
 parpydtk2::Mapper::ranks (C++ function), 44
 parpydtk2::Mapper::ready_ (C++ member), 48
 parpydtk2::Mapper::register_coupling_fields (C++ function), 47
 parpydtk2::Mapper::rho (C++ function), 46
 parpydtk2::Mapper::set_basis (C++ function), 44
 parpydtk2::Mapper::set_dimension (C++ function), 44
 parpydtk2::Mapper::set_leaf_b (C++ function), 45
 parpydtk2::Mapper::set_leaf_g (C++ function), 46
 parpydtk2::Mapper::set_rho (C++ function), 46
 parpydtk2::Mapper::set_search (C++ function), 49
 parpydtk2::Mapper::stat_ (C++ member), 48
 parpydtk2::Mapper::stat_write_freq_counter_ (C++ member), 48
 parpydtk2::Mapper::StatInfo (C++ class), 50
 parpydtk2::Mapper::title_ (C++ member), 49
 parpydtk2::Mapper::transfer_data (C++ function), 47
 parpydtk2::Mapper::use_awls (C++ function), 44
 parpydtk2::Mapper::use_knn_b (C++ function), 45
 parpydtk2::Mapper::use_knn_g (C++ function), 45
 parpydtk2::Mapper::use_mmls (C++ function), 44
 parpydtk2::Mapper::use_n2n (C++ function), 44
 parpydtk2::Mapper::use_radius_b (C++ function), 45
 parpydtk2::Mapper::use_radius_g (C++ function), 45
 parpydtk2::Mapper::use_spline (C++ function), 44
 parpydtk2::Mapper::verbose_ (C++ member), 48

R

radius_b (parpydtk2.Mapper attribute), 29, 33
 radius_g (parpydtk2.Mapper attribute), 29, 33
 rank (parpydtk2.IMeshDB attribute), 24, 28
 rank (parpydtk2.Mapper attribute), 29, 34
 ranks (parpydtk2.IMeshDB attribute), 24, 28
 ranks (parpydtk2.Mapper attribute), 29, 34
 register_coupling_fields() (parpydtk2.Mapper method), 34
 resolve_empty_partitions() (parpydtk2.IMeshDB method), 28
 rho (parpydtk2.Mapper attribute), 29, 34

S

set_matching_flag_n2n() (parpydtk2.Mapper method), 34
 show_experimental (C macro), 35
 show_experimental_if (C macro), 35
 show_info (C macro), 35
 show_info_master (C macro), 35
 show_warning (C macro), 35
 show_warning_if (C macro), 35
 size (parpydtk2.IMeshDB attribute), 24, 29
 SPLINE (in module parpydtk2), 22
 streamer (C macro), 35
 streamer_master (C macro), 35

T

throw_error (C macro), 35
 throw_error_if (C macro), 35
 throw_noimpl (C macro), 35
 throw_noimpl_if (C macro), 35
 transfer_data() (parpydtk2.Mapper method), 34

W

WENDLAND2 (in module parpydtk2), 22
 WENDLAND21 (in module parpydtk2), 22
 WENDLAND4 (in module parpydtk2), 22
 WENDLAND6 (in module parpydtk2), 22
 WU2 (in module parpydtk2), 22
 WU4 (in module parpydtk2), 22
 WU6 (in module parpydtk2), 22