

# The MSI User's Guide

– VERSION 0.1 –

Scientific Computation Research Center  
Rensselaer Polytechnic Institute

August 18, 2017

\*\*\*\*\* SCOREC 3-CLAUSE BSD LICENSE \*\*\*\*\*

Copyright (c) 2017, Scientific Computation Research Center (SCOREC), Rensselaer Polytechnic Institute (RPI). All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the SCOREC, RPI nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Questions to [seols@rpi.edu](mailto:seols@rpi.edu)

Derived from the OSI 3-clause BSD

# Contents

<b>1</b>	<b>API Overview</b>	<b>4</b>
1.1	Naming Convention . . . . .	4
1.2	Abbreviation . . . . .	4
1.3	Data Types and Classes . . . . .	4
1.4	Enumeration Types . . . . .	5
<b>2</b>	<b>API Functions</b>	<b>5</b>
2.1	Initialization and Finalization . . . . .	5
2.2	Mesh Entity . . . . .	6
2.3	Field . . . . .	7
2.4	PETSc Matrix and Solver . . . . .	7
<b>3</b>	<b>Installation</b>	<b>10</b>
3.1	S/W Requirements . . . . .	10
3.2	Compilation . . . . .	10
3.3	Test Program . . . . .	11

# 1 API Overview

The MSI API's are provided in the file “`msi.h`”. Throughout this section, unless specified, DOF, matrix row and column ID's are specified by a local ID. A word “*global*” in a function name indicates that the function involves global operation or global data.

## 1.1 Naming Convention

MSI API function name consists of three words connected with ‘\_’.

- the first word is “msi”
- the second word is an operation target. If the operation target is system-wide, the operation target is omitted. For instance, the function name which initializes the MSI service consists of two words: *msi\_start*.
- the third word is the operation description starting with a verb. For example, the function *msi\_matrix\_getNumIter* returns the number of iterations in the solve operation.

The following are operation targets used in the second word.

- *field*: the api is performed on a field object
- *matrix*: the api is performed on a Petsc matrix

## 1.2 Abbreviation

Abbreviations may be used in API naming. See <http://scorec.rpi.edu/wiki/Abbreviations> for more information.

## 1.3 Data Types and Classes

For a geometry, partition and mesh model, the term *instance* is used to indicate the model data existing on each process. For example, a mesh instance on process *i* means a pointer to a mesh data structure on process *i*, in which all mesh entities on process *i* are contained and from which they are accessible. For all other data such as field and matrix, the term *handle* is used to indicate the pointer to the data. For example, a matrix handle means a pointer to the matrix. The predefined data type has a prefix *p* to indicate the pointer data type.

The following are predefined data types used in the interface function parameters.

pMesh	a mesh instance (defined in “pumi.h”)
pMeshEnt	a mesh entity handle (defined in “pumi.h”)
pField	a field handle (defined in “pumi.h”)
pOwnership	a user-defined ownership handle (defined in “pumi.h”)
pMatrix	a matrix handle

## 1.4 Enumeration Types

The enumeration type for matrix type is:

```
msi_matrix_type {
    MSI_MULTIPLY = 0 /* matrix for multiplication */,
    MSI_SOLVE      /* 1 - matrix for solve */
}
```

The enumeration type for matrix status is:

```
msi_matrix_status {
    MSI_NOT_FIXED = 0 /* matrix is modifiable */,
    MSI_FIXED      /* 1 - matrix is not modifiable */
}
```

## 2 API Functions

### 2.1 Initialization and Finalization

The functions initialize/finalize the MSI operations.

```
void msi_start(
    pMesh /* in */ m,
    pOwnership /* in */ o=NULL)
```

Given a mesh and ownership handle, initialize MSI services for solver-PUMI interactions. If the ownership is not provided, the default is set to *NULL*. If the ownership is *NULL*, the PUMI’s default ownership rule is used (a part with the minimum process rank is the owning part of duplicate copies).

Note that the following operations should be performed prior to this function.

- MPI initialization
- Solver initialization (e.g. PetscInitialize)

- PUMI initialization
- model and mesh loading

```
int msi_scorec_finalize()
```

Finalize the MSI services and clears all internal data. Note that the following operations should follow to complete further finalizations.

- mesh deletion
- PUMI finalization
- Solver finalization (e.g. PetscFinalize)
- MPI finalization

## 2.2 Mesh Entity

In terms of mesh entity operation, MSI provides the operations only related to field ID of individual mesh entity as those are not supported in PUMI. For the rest of mesh entity operations including setting/getting field data (DOF) over nodes, use the API's in `PUMI.h`.

```
void msi_ment_getFieldID (
    pMeshEnt /* in */ e,
    pField /* in */ f,
    int /* in */ i,
    int* /* out */ start_dof_id,
    int* /* out */ end_dof_id_plus_one)
```

Given a mesh entity handle, field handle and the index of node  $i$ , return the starting local ID and the ending local ID plus one for field data (DOF) of  $i^{th}$  node of the entity.

```
void msi_ment_getGlobalFieldID (
    pMeshEnt /* in */ e,
    pField /* in */ f,
    int /* in */ i,
    int* /* out */ start_dof_id,
    int* /* out */ end_dof_id_plus_one)
```

Given a mesh entity handle, field handle and the index of node  $i$ , return the starting global ID and the ending global ID plus one for field data (DOF) of  $i^{th}$  node of the mesh entity.

## 2.3 Field

In terms of field operation, MSI provides the field creation only as PUMI field creation routine, `pumi_field_create`, does not support *multiple values*. For the rest of field operations including field deletion, use the API's in `PUMI.h`.

```
pField msi_field_create (  
    pMesh /* in */ m,  
    const char* /* in */ field_name,  
    int /* in */ nv,  
    int /* in */ nd)
```

Given field name, the number of values (*nv*), and the number of DOF's per value (*nd*), create a field for all nodes (owned, non-owned part boundary and ghost). Note that PUMI allocates contiguous memory for the array of field data. The size of field data is  $nv*st*nd*nn$ , where *st* is scalar type (1 for real, 2 for complex), and *nn* is the number of local nodes on each process.

If *nv* is 1, `msi_field_create (m, field_name, nv, nd)` is equivalent to `pumi_field_create (m, field_name, nd)`.

## 2.4 PETSc Matrix and Solver

```
pMatrix msi_matrix_create (  
    int /* in */ matrix_type,  
    pField /* in */ field)
```

Given a matrix type and a field handle, create a matrix and return its handle. The matrix type indicates the purpose of the matrix: 0 for matrix-vector multiplication and 1 for solver. The input field handle is used to retrieve the numbering (row/column ID) for matrix manipulation. The status of matrix is *MSI\_NOT\_FIXED* so the matrix values can be modified.

```
void msi_matrix_delete (pMatrix /* in */ matrix)
```

Given a matrix handle, delete the matrix.

```
void msi_matrix_assemble (pMatrix /* in */ matrix)
```

Given a matrix handle, perform matrix assembly and set the status of matrix to *MSI\_FIXED*. The matrix values cannot be modified any further.

```
void msi_matrix_insert (  
    pMatrix /* in */ matrix,  
    int /* in */ r,  
    int /* in */ c,  
    int /* in */ scalar_type,  
    double* /* in */ value)
```

Given a matrix handle, local DOF ID  $r$  and  $c$ , scalar type (0 for real, 1 for complex) and an array of double values, insert or overwrite *value* to the matrix at  $(r,c)$ . If *value* is a real number, *scalar\_type* is 0. Otherwise, *scalar\_type* is 1. A real type value can be inserted into a complex matrix but a complex type value cannot be inserted into a real matrix.

If the number of DOF's associated with the field is  $d$ ,  $r$  or  $c$  divided by  $d$  is a local ID of corresponding mesh entity. Use `msi_ment_getFieldID` to get local DOF ID of a mesh entity.

```
void msi_matrix_add (
    pMatrix /* in */ matrix,
    int /* in */ r,
    int /* in */ c,
    int /* in */ scalar_type,
    double* /* in */ value)
```

Given a matrix handle, local DOF ID  $r$  and  $c$ , scalar type (0 for real, 1 for complex) and an array of double values, add *value* to the existing value of matrix at  $(r,c)$ . If *value* is a real number, *scalar\_type* is 0. Otherwise, *scalar\_type* is 1. A real type value can be added into a complex matrix but a complex type value cannot be added into a real matrix.

If the number of DOF's associated with the field is  $d$ ,  $r$  or  $c$  divided by  $d$  is a local ID of corresponding mesh entity. Use `msi_ment_getFieldID` to get local DOF ID of a mesh entity.

```
void msi_matrix_addBlock (
    pMatrix /* in */ matrix,
    int /* in */ elem_id,
    int /* in */ r_value_index,
    int /* in */ c_value_index,
    double* /* in */ values)
```

Given a matrix handle, a local element ID, row value index, column value index, and a double array containing element matrix values, add values in the element matrix to the global matrix. The the number of the values associated with the field is 1, accepted value for *r\_value\_index* and *c\_value\_index* is 0.

```
void msi_matrix_setBC (
    pMatrix /* in */ matrix,
    int /* in */ local_row_index)
```

Given a matrix handle of type 1 (*MSL\_SOLVE*) and a local row index, zero out all off-diagonal values in the row of the matrix and set the diagonal value to one. The operation is carried out during finalizing the matrix. It will overwrite other insertion operations to the local row of the matrix. For complex-valued matrix, the real part of the diagonal is set to one and the imaginary part is set to zero. This function should be called on all processes that use the DOF numbering associated with the matrix row.

```
void msi_matrix_setLaplaceBC (
```



```

pMatrix /* in */ matrix,
int /* in */ row,
int /* in */ size,
int* /* in */ columns,
double* /* in */ values)

```

Given a matrix handle of type 1 (*MSLSOLVE*), a local row index, the number of values to be inserted (*size*), the columns to set the values (*columns*), and the values to be set in the order of the *columns*, set multiple values for the row of the matrix. If real values are inserted into a complex matrix, the corresponding imaginary parts are set to zero. The operation is carried out during finalizing the matrix. This function will overwrite other insertion operations to the row. This function should be called on all processes that use the DOF numbering associated with the matrix row.

```

void msi_matrix_multiply (
    pMatrix /* in */ A,
    pField /* in */ x,
    pField /* out */ b)

```

Given a matrix handle (*A*) of type 0 (*MSLMULTIPLY*) and an input field handle (*x*), perform the matrix-vector multiplication “*Ax*” and write the result in the field *b*. If the input matrix or the input field is complex-valued, the output field must be complex-valued.

```

void msi_matrix_solve (
    pMatrix /* in */ A,
    pField /* in */ b,
    pField /* out */ x)

```

Given a matrix handle (*A*) of type 1 (*MSLSOLVE*) and a RHS field handle (*b*), solve the global discrete equation “*Ax=b*” and write the solution into the field *x*.

```

int msi_matrix_getNumIter (pMatrix /* in */ matrix)

```

Given a matrix handle of type 1 (*MSLSOLVE*), return the number of iterations of solve operation.

```

void msi_matrix_print(pMatrix /* in */ matrix)

```

Given a matrix handle, print the *non-zero* matrix value along with global row/column index. The row/column ID starts with 0. PUMI provides an equivalent API for fields and nodes.

```

void msi_matrix_write (
    pMatrix /* in */ matrix,
    const char* /* in */ file_name,
    int /* in */ start_index=0)

```

Given a matrix handle, file name and a starting local ID for nodes (default is 0), write the *non-zero* matrix values in file(s). For each process *i*, the matrix information is written in “*filename-i*”. If *file\_name* is *NULL*, *msi\_matrix\_print* is performed. PUMI provides an equivalent API for fields and nodes.

## 3 Installation

MSI is a free open source software available in <https://github.com/SCOREC/msi>. This section discuss the S/W requirements and compilation briefly.

### 3.1 S/W Requirements

At a minumum, the following softwares are required to install MSI.

- `cmake` - v3.0 or higher
- MPI
- Zoltan [?]
- PUMI [?, ?]
- METIS/ParMETIS [?]

The detailed discussion on how to build PUMI can be found in <https://github.com/SCOREC/core/wiki/General-Build-instructions>.

### 3.2 Compilation

To build MSI, run a cmake configuration file and do “*make install*”. Two example cmake configuration files are available in the top source folder; “*openmpi-gcc4.4.5-real-config.sh*” for real type and “*openmpi-gcc4.4.5-complex-config.sh*” for complex type.

The essential configuration options include:

- `ZOLTAN_LIBRARY`: path and file name of Zoltan library
- `PARMETIS_LIBRARY`: path and file name of ParMETIS library
- `METIS_LIBRARY`: path and file name of METIS library
- `SCOREC_INCLUDE_DIR`: path to PUMI header files
- `SCOREC_LIB_DIR`: path and file name of PUMI libraries
- `ENABLE_PETSC`: set ON to link MSI with PETSc solver
- `PETSC_INCLUDE_DIR`: path to PETSc header files
- `PETSC_LIB_DIR`: path to PETSc libraries
- `ENABLE_TRILINOS`: set ON to link MSI with Trilinos solver

- `TRILINOS_INCLUDE_DIR`: path to Trilinos header files
- `TRILINOS_LIB_DIR`: path to Trilinos libraries
- `DENABLE_COMPLEX`: set ON to build MSI complex value
- `CMAKE_INSTALL_PREFIX`: path to install MSI header file and library

In the current version, Trilinos is not supported.

For a complete list of configuration options, see “*CMakeLists.txt*” in the top source folder.

### 3.3 Test Program

A test program with PETSc is available in “*test/petsc/main.cc*” in the top source folder. The test program is a good start to learn how to use MSI API. The input arguments of the test program are the following:

- `argv[1]` - input model file (.dmg)
- `argv[2]` - input mesh file (.smb)

How to generate PUMI-readable model and mesh files and how to use PUMI are beyond the scope of this document. For such a topic, see PUMI User’s Guide available in <http://www.scorec.rpi.edu/pumi>.