ORNL/TM-2019/1225

Documenting automated Fortran-C++ bindings with SWIG

Andrey Prokopenko
Matthew Bement
Seth Johnson
Katherine Evans

June 2019

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2019/1225

Computational Sciences and Engineering Division

DOCUMENTING AUTOMATED FORTRAN-C++ BINDINGS WITH SWIG

Andrey Prokopenko (PI) (Oak Ridge National Laboratory)
Matthew Bement (Los Alamos National Laboratory)
Seth Johnson (Oak Ridge National Laboratory)
Katherine Evans (Oak Ridge National Laboratory)

Date Published: June 2019

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

ABSTRACT

A new SWIG/Fortran tool introduced in Johnson et al. [to appear] automatically generates native Fortran 2003 interfaces to C and C++ libraries. This allows a seamless integration of existing C++ libraries into the Fortran applications. However, using the generated interfaces is complicated by the lack of appropriate documentation. In this report, we document a way to automatically port the existing Doxygen documentation of the C++ libraries to Fortran. We use ForTrilinos library as our target application, and discuss the scope and limitations of this approach.

1 INTRODUCTION

Doxygen is both a tool and a specification for writing documentation for software. The documentation is written within code comments, and thus is relatively easy to maintain. The documented code is run through the `doxygen` tool to produce output in one of the available formats, such as HTML, LaTeX, etc. Many large scientific projects, such as Trilinos Heroux et al. [2003], rely on Doxygen to document their application programming interface (API), hosting the generated HTML output on a website for easy access by the users.

Doxygen supports many programming languages. Of particular importance, it supports both C++ and Fortran. When running `doxygen`, it is possible to produce the output in language-agnostic form, i.e., XML format. Thus, if one is able to write a tool to convert Doxygen XML output back to a specific language, and to insert it in the right place of the generated code, it would be possible to translate comments from one language to the other.

In this report, we discuss the steps required to insert Doxygen documentation into the automatically generated Fortran interface wrappers for an existing C++ library.

2 GENERAL PROCEDURE

In this Section, we describe the steps to automatically port documentation from a C++ header to an automatically generated Fortran interface wrapper.

Step 1. Produce language-agnostic XML output

The first step is to use Doxygen to generate XML output. This is done by runnning the `doxygen` tool on the relevant C++ library headers. As an input, `doxygen` takes in a Doxygen configuration file (Doxyfile). A default Doxyfile could be generated by running

```
$ doxygen -g
```

In order to produce XML output, the option `GENERATE_XML` in the Doxyfile must be set to `YES`.

Step 2. Convert XML output to SWIG docstrings

In order for the documentation be attached in the right places in the generated output later, it must first be converted to a SWIG-readable format called docstrings. A docstring is a SWIG code snippet containing the function name and its documentation, including the input arguments. For example,

```
%feature("docstring")  Teuchos::ParameterList::set "ParameterList &
Teuchos::ParameterList::set(std::string const &name, T const &value)

Set a parameter whose value has type T.

Parameters:
-----------

name:  [in] The parameter's name.

value:  [in] The parameter's value. This determines the template parameter T.
   In most cases, you will not need to specify the type T explicitly; the
   compiler will infer it from this argument.
```

This is done by running a Python script `doxy2swig.py` (available as part of the Trilinos' package PyTrilinos) as follows:

```
$ doxy2swig.py doxygen_output.xml swig_docstrings.i
```

Here, `doxygen_output.xml` is the Doxygen XML output produced by the first step, and `swig_docstrings.i` is the resulting SWIG file containing docstrings for all the functions in the `index.xml`.

Step 3. Include SWIG docstrings file into SWIG interface file

This file of docstrings must then be included in the SWIG interface file like this:

```
%include "swig_docstrings.i"
... rest of SWIG file ...
```

When SWIG is run, the documentation strings will be automatically inserted in the appropriate places in the wrapper code in the form of native language comments.

Step 4. Run Doxygen on the produced Fortran wrapper files

The last step is to run `doxygen` again on the produced wrapper file to generate the documentation in the desired format.

3 FORTRILINOS ADAPTATION

In this Section, we describe the adaptation of the general approach described in Section 2 to ForTrilinos. For reasons documented in Section 4, we require doxygen version 1.8.15 or above.

ForTrilinos provides interfaces for several Trilinos packages (Belos, Teuchos, Tpetra) and a general linear and nonlinear solver interfaces. We will use Tpetra to describe the documentation generation.

As discussed in Section 2, the first step is to generate Doxygen XML documentation. In order to automatically pick up the location of the original Trilinos Tpetra headers, ForTrilinos carries `Doxyfile_tpetra.in` file with the following code:

```
INPUT = @PROJECT_SOURCE_DIR@/packages/tpetra/classic/src \
        @PROJECT_SOURCE_DIR@/packages/tpetra/classic/NodeAPI \
        @PROJECT_SOURCE_DIR@/packages/tpetra/core/src \
        @PROJECT_SOURCE_DIR@/packages/tpetra/kernels/src \
        @PROJECT_SOURCE_DIR@/packages/tpetra/tsqr/src
```

During ForTrilinos configuration, the `@PROJECT_SOURCE_DIR@` macro is automatically replaced by the correct location of the Trilinos source directory. We note here that we used this approach instead of specifying a relative (to ForTrilinos) path as the later does not properly work with symbolic links.

The `Doxyfile_tpetra.in` is also modified to have

```
EXTRACT_PRIVATE = YES
EXTRACT_ALL = YES
```

This step produces a file `tpetra_dox.i` containing the appropriate docstrings, and places it in the `ForTrilinos/src/tpetra/src` directory, containing other SWIG files used to generate Tpetra interfaces. The main file, `fortpetra.i`, contains the following line

```
%include "tpetra_dox.i"
```

in order to automatically pick up docstrings with SWIG.

The ForTrilinos wrapper files are generated only in developer mode (enabled with `-DForTrilinos_ENABLE_DeveloperMode=ON`). The generated file, `fortpetra.F90` in this case, will contain the required documentation, a snippet of which is shown below:

```
module fortpetra
 use, intrinsic :: ISO_C_BINDING
 implicit none
 private

 ! DECLARATION CONSTRUCTS

 ! class Tpetra::Map< LO,GO,NO >
!> A parallel distribution of indices over processes.
!>
!> Parameters:
!> -----------
!>
!> LocalOrdinal:  The type of local indices. Currently, this must be int.
!> (In Epetra, this is always just int.)
!>
!> <snip>
!>
!> C++ includes: Tpetra_Map_decl.hpp
 type, public :: TpetraMap
  type(SwigClassWrapper), public :: swigdata
 contains
  procedure :: getGlobalNumElements => swigf_TpetraMap_getGlobalNumElements
  ...
```

```
  procedure, private :: swigf_TpetraMap_getRemoteIndexList__SWIG_0
  procedure, private :: swigf_TpetraMap_getRemoteIndexList__SWIG_1
  generic :: getRemoteIndexList => swigf_TpetraMap_getRemoteIndexList__SWIG_0,
       swigf_TpetraMap_getRemoteIndexList__SWIG_1
 end type TpetraMap
 interface TpetraMap
  module procedure swigf_new_TpetraMap__SWIG_1
  module procedure swigf_new_TpetraMap__SWIG_2
  ...
 end interface
 <snip>
contains
 ! MODULE SUBPROGRAMS

  !> Tpetra::Map< LocalOrdinal,
  !> GlobalOrdinal, Node >::Map()
  !>
  !> Default constructor (that does nothing).
  !>
  !> This creates an empty Map, with 0 (zero) indices total. The Map's
  !> communicator only includes the calling process; in MPI terms, it
  !> behaves like MPI_COMM_SELF.
  !>
  !> This constructor exists mainly to support view semantics of Map. That
  !> is, we can create an empty Map, and then assign a nonempty Map to it
  !> using operator=. This constructor is also useful in methods like
  !> clone() and removeEmptyProcesses(), where we have the information to
  !> initialize the Map more efficiently ourselves, without going through
  !> one of the three usual Map construction paths.
  function swigf_new_TpetraMap__SWIG_1(numglobalelements, comm, lg) &
     result(self)
   ...
  end function
  !> Tpetra::Map< LocalOrdinal,
  !> GlobalOrdinal, Node >::Map()
  !>
  !> Default constructor (that does nothing).
  !>
  !> This creates an empty Map, with 0 (zero) indices total. The Map's
  !> communicator only includes the calling process; in MPI terms, it
  !> behaves like MPI_COMM_SELF.
  !>
  !> This constructor exists mainly to support view semantics of Map. That
  !> is, we can create an empty Map, and then assign a nonempty Map to it
  !> using operator=. This constructor is also useful in methods like
  !> clone() and removeEmptyProcesses(), where we have the information to
  !> initialize the Map more efficiently ourselves, without going through
  !> one of the three usual Map construction paths.
  function swigf_new_TpetraMap__SWIG_2(numglobalelements, comm) &
      result(self)
   ...
```

```
end function
!> global_size_t Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node
!> >::getGlobalNumElements() const
!>
!> The number of elements in this Map.
!>
!> This function should be thread safe and thread scalable, assuming that
!> you refer to the Map by value or reference, not by Teuchos::RCP.
function swigf_TpetraMap_getGlobalNumElements(self) &
     result(swig_result)
 ...
end function
```

As we can see, the generated documentation is inserted into the Fortran wrapper file in correct places. However, due to limitations described in Section 4, the documentation for the constructors is repeated.

The final run of `doxygen` on the generated `F90` file produces the results as shown in Figures 1 and 2.

4 LIMITATIONS

The described procedure works well in many situations. However, it also has significant drawbacks that are hard to overcome.

Specifically, for C++ to Fortran documentation translation, we encountered:

- **The docstrings are simply copied from the XML documentation**

  Often, the C++ documentation uses C++ syntax and terminology, references C++ classes and methods. Ideally, such syntax should be translated into the wrapper language syntax, so that all of the documentation visible to the user of a package would be in native format.

- **Generic procedures are handled incorrectly**

  Fortran generic procedures pose a challenge on multiple fronts. First, there are challenges related to Doxygen itself. For example, note the `getremoteindexlist` generic of the `Tpetramap` interface in Figure 1. If one follows the link to `swigf_tpetramap_getremoteinexlist_swig_0`, there is essentially no documentation, as shown in Figure 3.

  To access the documentation for this procedure, one has to refer back to the top level `fortpetra` interface documentation, navigate to the Functions/Subroutines section and then follow the link to `swigf_tpetramap_getremoteinexlist_swig_0`, where one finds the expected documentation, as is shown in three images shown in Figure 4.

  When one does access the documentation for the generic procedures in this manner, the documentation is identical for different function signatures, as is also seen in Figure 2. This is due to how `doxy2swig` handles the different argument lists. It is not currently known if modifications to `doxy2swig` could address this limitation, and if so, if SWIG allows docstrings to be associated with function signatures in addition to function names.

We also note that only recent versions of Doxygen seem to work. The recommended version is 1.8.15 or above. For example, we found that Doxygen 1.8.5 does not link the documentation of the binding name function to the procedure name.

5 REFERENCES

Michael Heroux et al. An overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

Seth R. Johnson, Andrey Prokopenko, and Katherine J. Evans. Automated Fortran-C++ bindings for large-scale scientific applications. *Computing in Science & Engineering*, to appear.

ForTrilinos

Main Page | Modules | Data Types List | Files | Search

fortpetra | tpetramap

Public Member Functions | Public Attributes | Private Member Functions | List of all members

fortpetra::tpetramap Interface Reference

A parallel distribution of indices over processes. More...

Public Member Functions

procedure	release => swigf_release_tpetramap
procedure	isonetoone => swigf_tpetramap_isonetoone
procedure	getglobalnumelements => swigf_tpetramap_getglobalnumelements
procedure	getnodenumelements => swigf_tpetramap_getnodenumelements
procedure	getminlocalindex => swigf_tpetramap_getminlocalindex
procedure	getmaxlocalindex => swigf_tpetramap_getmaxlocalindex
procedure	getminglobalindex => swigf_tpetramap_getminglobalindex
procedure	getmaxglobalindex => swigf_tpetramap_getmaxglobalindex
procedure	getminallglobalindex => swigf_tpetramap_getminallglobalindex
procedure	getmaxallglobalindex => swigf_tpetramap_getmaxallglobalindex
procedure	getlocalelement => swigf_tpetramap_getlocalelement
procedure	getglobalelement => swigf_tpetramap_getglobalelement
procedure	getnodeelementlist => swigf_tpetramap_getnodeelementlist
procedure	isnodelocalelement => swigf_tpetramap_isnodelocalelement
procedure	isnodeglobalelement => swigf_tpetramap_isnodeglobalelement
procedure	isuniform => swigf_tpetramap_isuniform
procedure	iscontiguous => swigf_tpetramap_iscontiguous
procedure	isdistributed => swigf_tpetramap_isdistributed
procedure	iscompatible => swigf_tpetramap_iscompatible
procedure	issameas => swigf_tpetramap_issameas
procedure	locallysameas => swigf_tpetramap_locallysameas
procedure	islocallyfitted => swigf_tpetramap_islocallyfitted
procedure	getcomm => swigf_tpetramap_getcomm
procedure	description => swigf_tpetramap_description
procedure	removeemptyprocesses => swigf_tpetramap_removeemptyprocesses
procedure	replacecommwithsubset => swigf_tpetramap_replacecommwithsubset
generic	assignment => swigf_tpetramap_op_assign__
generic	getremoteindexlist => swigf_tpetramap_getremoteindexlist__swig_0, swigf_tpetramap_getremoteindexlist__swig_1
type(tpetramap) function	swigf_new_tpetramap__swig_1 (numglobalelements, comm, lg) Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map() More...
type(tpetramap) function	swigf_new_tpetramap__swig_2 (numglobalelements, comm) Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map() More...
type(tpetramap) function	swigf_new_tpetramap__swig_4 (numglobalelements, numlocalelements, comm) Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map() More...
type(tpetramap) function	swigf_new_tpetramap__swig_7 (numglobalelements, indexlist, comm) Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map() More...
type(tpetramap) function	swigf_new_tpetramap__swig_8 () Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map() More...
integer(tpetralookupstatus) function	swigf_tpetramap_getremoteindexlist__swig_0 (self, gidlist, nodeidlist, lidlist) LookupStatus Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::getRemoteIndexList(const Teuchos::ArrayView< const GlobalOrdinal > &GIDList, const Teuchos::ArrayView< int > &nodeIDList) const. More...

◆ swigf_new_tpetramap__swig_1()

```
type(tpetramap) function fortpetra::swigf_new_tpetramap__swig_1 ( integer(c_long), intent(in)          numglobalelements,
                                                                  class(teuchoscomm), intent(in)     comm,
                                                                  integer(tpetralocalglobal), intent(in) lg
                                                                )
```

Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map()

Default constructor (that does nothing).

This creates an empty Map, with 0 (zero) indices total. The Map's communicator only includes the calling process; in MPI terms, it behaves like MPI_COMM_SELF.

This constructor exists mainly to support view semantics of Map. That is, we can create an empty Map, and then assign a nonempty Map to it using operator=. This constructor is also useful in methods like clone() and removeEmptyProcesses(), where we have the information to initialize the Map more efficiently ourselves, without going through one of the three usual Map construction paths.

Figure 1. Generated Doxygen documentation for `TpetraMap`.

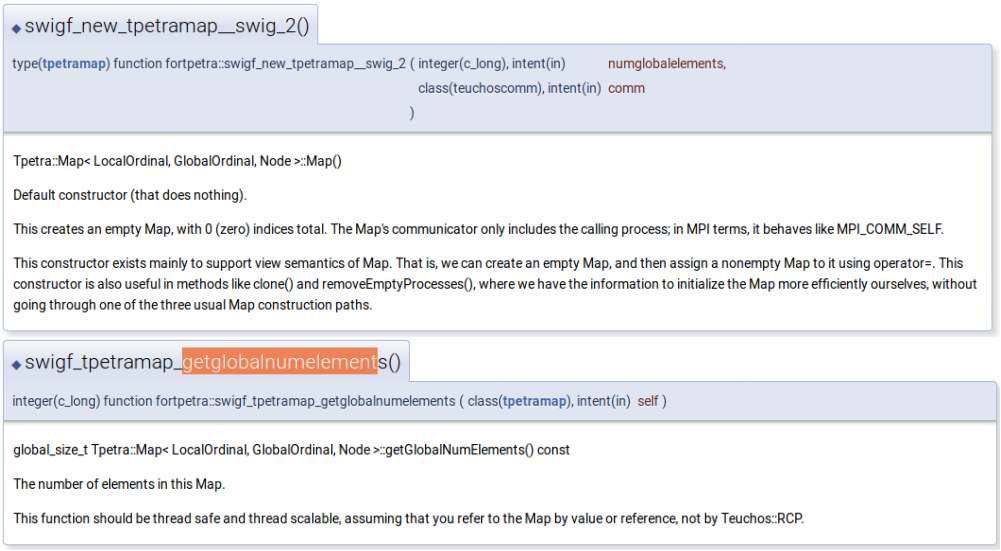


Figure 2. Generated Doxygen documentation for `TpetraMap`.

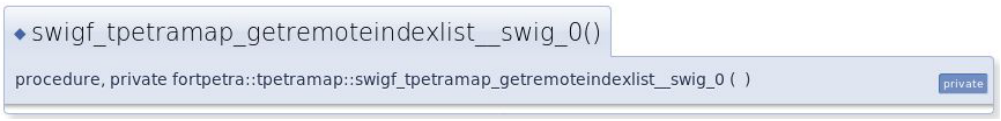
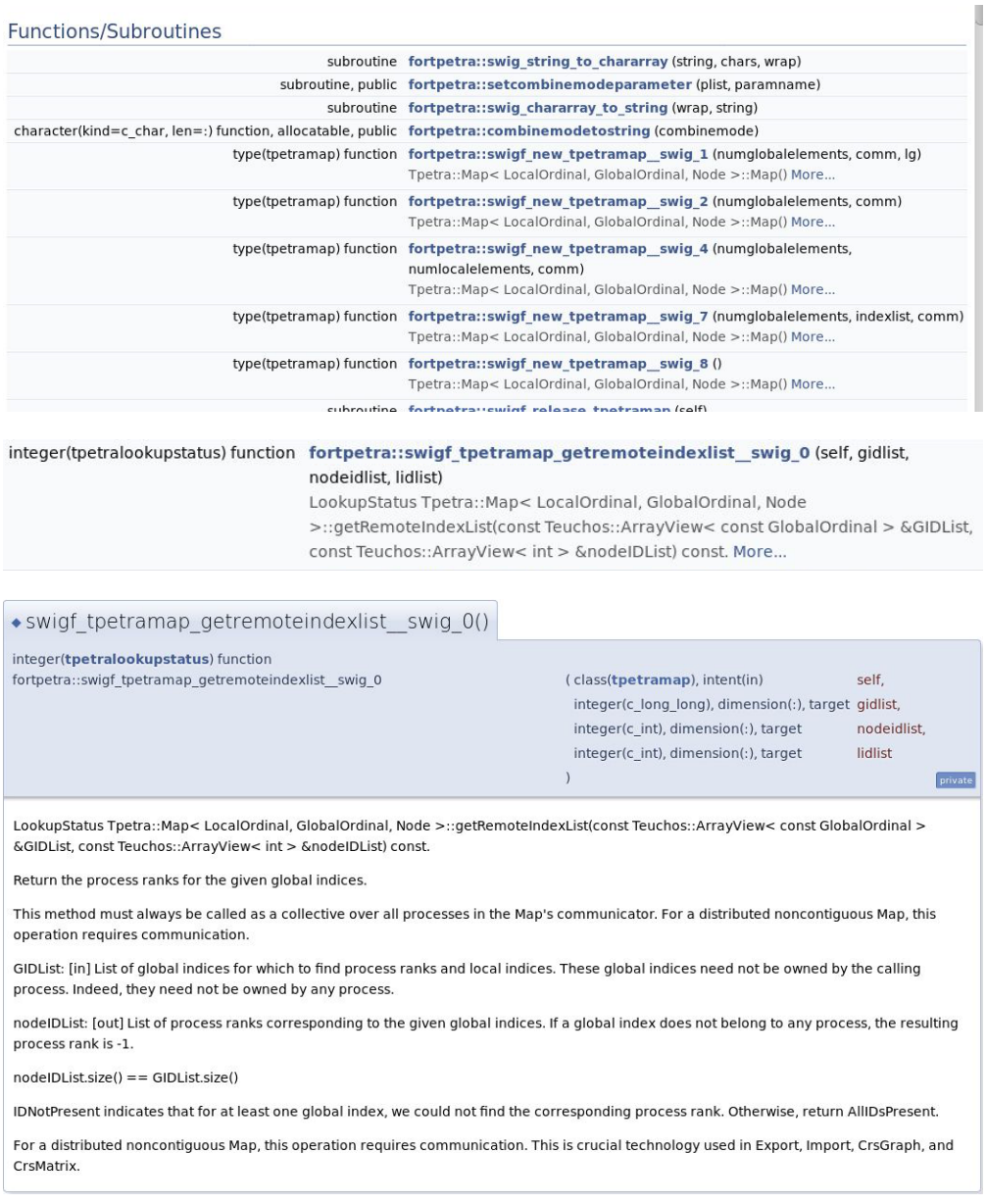


Figure 3. Incorrect Doxygen output for `TpetraMap` generic.



Functions/Subroutines

subroutine	**fortpetra::swig_string_to_chararray** (string, chars, wrap)
subroutine, public	**fortpetra::setcombinemodeparameter** (plist, paramname)
subroutine	**fortpetra::swig_chararray_to_string** (wrap, string)
character(kind=c_char, len=:) function, allocatable, public	**fortpetra::combinemodetostring** (combinemode)
type(tpetramap) function	**fortpetra::swigf_new_tpetramap__swig_1** (numglobalelements, comm, lg) Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map() More...
type(tpetramap) function	**fortpetra::swigf_new_tpetramap__swig_2** (numglobalelements, comm) Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map() More...
type(tpetramap) function	**fortpetra::swigf_new_tpetramap__swig_4** (numglobalelements, numlocalelements, comm) Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map() More...
type(tpetramap) function	**fortpetra::swigf_new_tpetramap__swig_7** (numglobalelements, indexlist, comm) Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map() More...
type(tpetramap) function	**fortpetra::swigf_new_tpetramap__swig_8** () Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::Map() More...
subroutine	**fortpetra::swigf_release_tpetramap** (self)

integer(tpetralookupstatus) function **fortpetra::swigf_tpetramap_getremoteindexlist__swig_0** (self, gidlist, nodeidlist, lidlist)

LookupStatus Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::getRemoteIndexList(const Teuchos::ArrayView< const GlobalOrdinal > &GIDList, const Teuchos::ArrayView< int > &nodeIDList) const. More...

◆ swigf_tpetramap_getremoteindexlist__swig_0()

integer(**tpetralookupstatus**) function
fortpetra::swigf_tpetramap_getremoteindexlist__swig_0 (class(**tpetramap**), intent(in) self,
integer(c_long_long), dimension(:), target gidlist,
integer(c_int), dimension(:), target nodeidlist,
integer(c_int), dimension(:), target lidlist
)

private

LookupStatus Tpetra::Map< LocalOrdinal, GlobalOrdinal, Node >::getRemoteIndexList(const Teuchos::ArrayView< const GlobalOrdinal > &GIDList, const Teuchos::ArrayView< int > &nodeIDList) const.

Return the process ranks for the given global indices.

This method must always be called as a collective over all processes in the Map's communicator. For a distributed noncontiguous Map, this operation requires communication.

GIDList: [in] List of global indices for which to find process ranks and local indices. These global indices need not be owned by the calling process. Indeed, they need not be owned by any process.

nodeIDList: [out] List of process ranks corresponding to the given global indices. If a global index does not belong to any process, the resulting process rank is -1.

nodeIDList.size() == GIDList.size()

IDNotPresent indicates that for at least one global index, we could not find the corresponding process rank. Otherwise, return AllIDsPresent.

For a distributed noncontiguous Map, this operation requires communication. This is crucial technology used in Export, Import, CrsGraph, and CrsMatrix.

Figure 4. Correct Doxygen output for `TpetraMap` generic.