User's Guide
KMATHLIB API
Version 1.0

Large-scale Parallel Numerical Computing Technology Research Team
RIKEN Advanced Institute for Computational Scicence

April 11, 2016

Contents

Chapter 1

Introduction

KMATHLIB API is an application programming interface for using various numerical libraries through a common interface. It aims to reduce the burden of computational science software developers by proactively integrating the high-performance library into the software.

The development of computational science software requires the use of a high-performance numerical library, and often multiple numerical libraries are used simultaneously. However, in general these libraries have different data structures and usage methods. Developers are required to understand these differences and implement conversion functions of the data structures if necessary. Due to this, the developed program becomes cumbersome and more effort is required for development and maintenance of the computational science software. This might be a factor that deters computational science software developers from incorporating new libraries.

By using the KMATHLIB API, developers are able to use a variety of numerical libraries (PETSc[5, 6, 7], ScaLAPACK[4], FFTW[9], EigenExa[10], etc.) through a common interface. The API mitigates the aforementioned bottlenecks, as universal procedures can be accessed without concerns about the different libraries.

The KMATHLIB API was designed to be independent of any specific computer architecture. To date, the API has been confirmed to function on various high-performance computing platforms, such as the K computer, Fujitsu PRIMEHPC FX10, Fujitsu PRIMEHPC FX100, and cluster systems equipped with Intel x86-based processors. This document serves as a User's Guide for the KMATHLIB API 1.0.

The following chapters contain information about the basic mechanisms of the interface, the build procedure of the package, software development methodologies, and the specifications of KMATHLIB API.

Copyright

LICENCE.txt

Chapter 2

KMATHLIB API and Its Functions

The KMATHLIB API is an API for the simultaneous use of various numerical computation software (solvers). Auxiliary functions are provided with the API.

During software development using KMATHLIB API and its auxiliary functions (KMATHLIB API functions), the KMATHLIB API indirectly accesses solver functions, eliminating the task of directly accessing each solver interface. The KMATHLIB API is designed to run a variety of solvers through a universal method, thereby facilitating software development without concern for the different data formats and procedures of each solver.

This chapter will explain the KMATHLIB API and its auxiliary functions, as well as the precautions for each.

2.1 Hierarchy

The KMATHLIB API functions are separated into those that are directly accessed by users, functions for plugin management, and plugins to call individual solvers with a hierarchical structure as shown in Figure 2.1. Only KMATHLIB API and a portion of the PETSc functions that handle standard format data (described in Section 2.2), are colled directly from the software (application) using the KMATHLIB API functions. Plugin management and plugin calls are done internally from the KMATHLIB API so the application does not need to call them directly. The functions for plugin management consist of a function to initialize the individual plugins, an end function, a function to assess whether data conversion is needed, and a function for parameter configuration. Plugins consist of functions to use the solvers, including those for data format conversion, selection of a solver routine that fits the user parameters set through the KMATHLIB API, and the actual call to the solver routine.

2.2 Data Conversion Functions

The data formats (standard format) of the input and output variables are defined in the KMATHLIB API to enable a universal method that allows use of various solvers requiring different data formats.

When the solver is called through the plugin function, an automatic data conversion is performed on the input and output variables to the data format required by the solver (internal format). In the KMATHLIB API, the data conversion of input and output variables does not

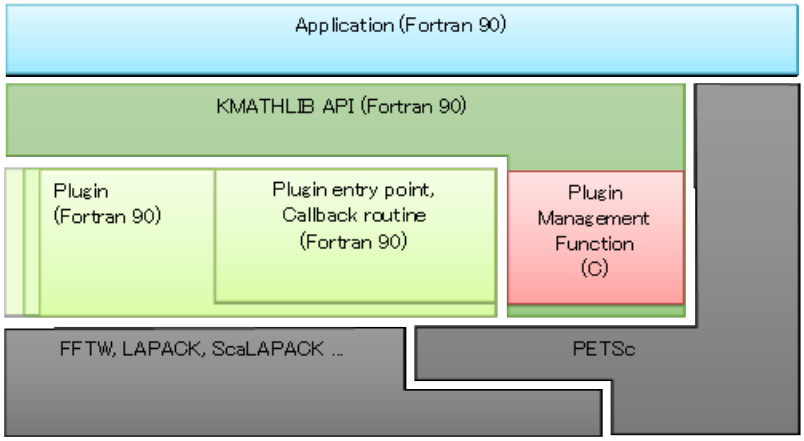

Figure 2.1: The module structure of KMATHLIB API functions.

necessarily take place always, as unnecessary conversions are suppressed through a validation process embedded in the plugin management function.

The data format used in PETSc has been accepted as the standard format of the KMATHLIB API functions. For this reason, PETSc functions are used in a portion of the functions in the KMATHLIB API. In addition, because the PETSc data type is used as the data format, attention needs to be given to how the standard-format scalar type will be affected by the PETSc scalar type (double-precision real or complex numbers, etc.) that is referenced to build the KMATHLIB API.

2.3 Plugins

As shown in Chapter 3, plugins (standard plugins) are included in the KMATHLIB API packages. Additional plugins may be added, and these are treated in exactly the same way; there is no mechanism to distinguish standard plugins from user-added ones. In addition, there are no restrictions on the solution type (linear system solving, eigenvalue solution, etc.) of the user-installed plugins. If the conversion of the internal and standard formats is explained within the plugin, any solution method can be incorporated as a plugin.

Chapter 3

The Build and Installation of KMATHLIB API

This chapter contains an overview of the KMATHLIB API package, as well as the instructions for build and installation.

3.1 An Overview of the KMATHLIB API Package

The KMATHLIB API package contains the source code of the KMATHLIB API (core API), including plugin management functions, the source code for standard plugins and test programs, and the `Makefile` to build these programs. The package directory structure is shown in Figure 3.1.

Requirements

The requirements to build the core API of KMATHLIB are the following:

- C/C++ compiler
- Fortran 90/95 compiler
- MPI
- PETSc

To date, it has been confirmed that the software listed in Table 3.1 can be used to successfully build the core API.

Table 3.1: Software with confirmed compatibility (core API).

Compiler	Fujitsu Compilers mpifrtpx, mpifccpx, mpiFCCpx (cross compiler for the K computer, FX10, FX100), Intel Compiler (ifort, icc 13.0.1)
MPI	Fujitsu MPI, OpenMPI 1.6.5
PETSc	PETSc 3.6.3

Table 3.2: Scalar type dependency and additional software required by plugins.

Plugin	Required additional software	Scalar type dependency
`template`	None	None
`dls_ssl2`	SSL II[1, 2]	Functions only if real
`dls_lapack`	LAPACK[3]	None
`dls_scalapack`	ScaLAPACK	None
`dlsm_scalapack`	ScaLAPACK	None
`deig_ssl2`	SSL II	Functions only if real
`deig_lapack`	LAPACK	None
`deig_scalapack`	ScaLAPACK	None
`deig_eigenea`	EigenExa	Functions only if real
`dsvd_scalapack`	ScaLAPACK	None
`sls_petsc`	None	None
`fft_fftw`	FFTW	Functions only if complex
`fft_kmath_fft3d`	FFTE[9] (source code)	Functions only if complex
`rnd_kmath_random`	KMATH RANDOM[11]	None

Table 3.3: Software with confirmed compatibility (standard plugin).

SSL II	Fujitsu SSL II (those provided by the K computer, FX10, FX100)
LAPACK	Fujitsu SSL II (those provided by the K computer, FX10, FX100), Intel Math Kernel Library 11.0.1.117
ScaLAPACK	ScaLAPACK (the K computer, FX10, FX100), Intel Math Kernel Library 11.0.1.117
EigenExa	EigenExa 2.3c
FFTW	FFTW 3.3.3
FFTE	FFTE 6.0.0
KMATH RANDOM	KMATH RANDOM 1.1

Additional software is required to build the plugin. Some plugins are buildable and/or operable only when the scalar type of PETSc is a real number or a complex number. Table 3.2 shows the necessary software and the scalar type dependency for each additional plugin.

Individual build and installation is possible for each plugin, and users need only install the software for the plugins that they require. To date, it has been confirmed that the software in Table 3.3 can be used to successfully build the plugins.

3.2 Build Procedure

3.2.1 The Configuration of Environment Variables

This section describes the environment variables required by the core API of the KMATHLIB API and during the build process of the plugin. In addition, this subsection shows a sample configuration using the `export` command based on the assumption that the execution shell is bash or zsh. When csh or tcsh is used, an appropriately substituted command should be called.

PETSC_INC, PETSC_LIB

The library path and include path of PETSc are configured as PETSC_INC, PETSC_LIB using the following commands:

Sample configuration of the PETSc environment variables

```
$ export PETSC_INC\
  ="$HOME"/opt/petsc-3.6.3_complex/arch-fujitsu-sparc64fx-opt-z/include
$ export PETSC_LIB\
  ="$HOME"/opt/petsc-3.6.3_complex/arch-fujitsu-sparc64fx-opt-z/lib
```

The environment variable PETSC_INC is referenced during the build of all plugins, with the exception of the core API build and template. In addition, the environment variable PETSC_LIB is referenced during the build of the KMATHLIB API test programs. The PETSc path must be configured for the environment variables in accordance with the scalar type used.

EIGENEXA_INC, EIGENEXA_LIB

The library path and include path of EigenExa are configured as EIGENEXA_INC, EIGENEXA_LIB using the following commands:

Sample configuration of the EigenExa environment variables

```
$ export EIGENEXA_INC="$HOME"/opt/EigenExa-2.3c/include
$ export EIGENEXA_LIB="$HOME"/opt/EigenExa-2.3c/lib
```

The environment variable EIGENEXA_INC is referenced during the build of the deig_eigenexa plugin. In addition, the environment variable EIGENEXA_LIB is referenced during the build of the test program (test/04) of the deig_eigenexa plugin.

FFTW_INC, FFTW_LIB

The library path and include path of FFTW are configured as FFTW_INC, FFTW_LIB using the following commands:

Sample configuration of the FFTW environment variables

```
$ export FFTW_INC=/home/apps/fftw/3.3.3/include
$ export FFTW_LIB=/home/apps/fftw/3.3.3/lib64
```

The environment variable FFTW_INC is referenced during the build of the fft_fftw plugin. In addition, the environment variable FFTW_LIB is referenced during the build of the test program (test/09, test/10) of the fft_fftw plugin and the test program (test/11) of the fft_fftw and fft_kmath_fft3d plugins.

FFTE_SRC

The directory path to the FFTE source code is configured using the following command:

Sample configuration of the FFTE environment variable

```
$ export FFTE_SRC="$HOME"/opt/ffte-6.0
```

This environment variable is used during the build of the fft_kmath_fft3d plugin.

KMATH_RANDOM_INC, KMATH_RANDOM_LIB

The library path and include path of KMATH_RANDOM are configured as KMATH_RANDOM_INC, KMATH_RANDOM_LIB using the following commands:

Sample configuration of the KMATH RANDOM environment variables

```
$ export KMATH_RANDOM_INC="$HOME"/opt/KMATH_RANDOM-1.1/random/f90
$ export KMATH_RANDOM_LIB="$HOME"/opt/KMATH_RANDOM-1.1/random/f90
```

The environment variable KMATH_RANDOM_INC is referenced during the build of the rnd_kmath_random plugin. In addition, the environment variable KMATH_RANDOM_LIB is referenced during the build of the test program (test/12) of the rnd_kmath_random plugin.

3.2.2 The Build and Installation of KMATHLIB API

This subsection explains the build and installation procedures of the KMATHLIB API using the K computer, Fujitsu PRIMEHPC FX10, or the x86 cluster. Even in other computing environments, a similar build is possible with slight modifications.

Change to the directory where the KMATHLIB API package is located and extract the contents from the package using the following command. Open the directory containing the unpacked content.

Extraction of the KMATHLIB API Package

```
$ tar zxf KMATHLIB_API-1.0.tgz
$ cd KMATHLIB_API-1.0
```

If building with the x86 cluster, link and edit the computer information configuration file using the following command. It should be noted that the following configuration is not required for the K computer and Fujitsu PRIMEHPC FX10.

Configuration of the computer information file

```
$ cd src/
$ rm Makefile.machine
$ ln -s arch/Makefile.machine.Intel Makefile.machine
$ vim Makefile.machine    (Change the configuration as necessary)
$ cd ../
```

Perform the build and installation of the core API using the following command:

Build and installation of the core API

```
$ cd src/core
$ make install
$ cd ../..
```

The command will automatically install the intermediate files in module format (*.mod files) and library files (*.a file) of the core API into the include/ and lib/ directories.

The following section explains the build and installation of standard plugins. For instance, perform the installation of the dls_scalapack plugin using the following command:

Build and installation of plugins

```
$ cd src/plugins/dls_scalapack
$ make install
$ cd ../../..
```

This command installs the intermediate module file and library files of the `dls_scalapack` plugin into the `include/` and `lib/` directories. This procedure is repeated for all standard plugins requiring installation. To install all plugins, the following command can be used to automatically build all standard plugins.

Build and installation of all plugins

```
$ cd src/plugins
$ make install
$ cd ../..
```

3.2.3 The Build and Installation of Test Programs

This section describes how to build, install, and run the test programs included in this package. The following procedure can be used to build and run the test program (`test/06`) of the `dls_scalapack` plugin.

Example Build of the Test Program (`test/06`)

```
$ cd test/06
$ make
(Create the executable test)
$ vim run.sh
(Create the job script file)
$ pjsub run.sh
(The job isrun and the results are outputted in run.sh.o[job number].)
$ cd ../..
```

For reference, the following is a sample creation of a job script file for the K computer.

A sample creation of the job script file (`run.sh`) (the K computer)

```
#!/bin/bash -x
#
#PJM --rsc-list "node=6"
#PJM --rsc-list "elapse=00:05:00"
#PJM --stg-transfiles all
#PJM --stgin "./test ./"
#PJM -s
#
. /work/system/Env_base

mpiexec -n 6 ./test
```

It is possible to build and run the other test programs using the same method.

```
KMATHLIB_API_v1.0
|-- include                   Module files (*.mod files) will be installed.
|-- lib                       Library files (*.a files) will be installed.
|-- src
|   |-- arch                  Stores computer-dependent information needed
|   |                         during the build.
|   |-- core                  Stores the core API.
|   `-- plugins               Stores standard plugins.
|       |-- deig_eigenexa         Plugin to solve the real standard symmetric
|       |                         eigenvalue problem with EigenExa.
|       |-- deig_lapack           Plugin to solve the standard eigenvalue problem
|       |                         with LAPACK.
|       |-- deig_scalapack        Plugin to solve the standard eigenvalue problem
|       |                         with ScaLAPACK.
|       |-- deig_ssl2             Plugin to solve the standard eigenvalue problem
|       |                         with SSL II.
|       |-- dsvd_scalapack        Plugin to compute the singular value decomposition
|       |                         with ScaLAPACK.
|       |-- dls_lapack            Plugin to solve the linear equation with LAPACK.
|       |-- dls_scalapack         Plugin to solve the linear equation with ScaLAPACK.
|       |-- dls_ssl2              Plugin to solve the linear equation with SSL II.
|       |-- dlsm_scalapack        Plugin to solve the linear equation with multiple
|       |                         right-hand sides with ScaLAPACK.
|       |-- fft_fftw              Plugin to run FFT with FFTW.
|       |-- fft_kmath_fft3d       Plugin to run 3D FFT with KMATH FFT3D.
|       |-- rnd_kmath_random      Plugin to run random number generation
|       |                         with KMATH RANDOM.
|       |-- sls_petsc             Plugin to solve systems of sparse linear equations
|       |                         with PETSc.
|       `-- template              Plugin template.
`-- test                      Stores test programs for the standard plugin.
    |-- 01                        Test program of template.
    |-- 02                        Test program of template.
    |-- 03                        Test program of sls_petsc.
    |-- 04                        Test program of deig_scalapack, deig_eigenexa.
    |-- 05                        Test program of deig_ssl2, deig_lapack.
    |-- 06                        Test program of dls_scalapack.
    |-- 07                        Test program of dls_lapack.
    |-- 08                        Test program of dls_ssl2.
    |-- 09                        Test program of fft_fftw.
    |-- 10                        Test program of fft_fftw, fft_kmath_fft3d.
    |-- 11                        Test program of fft_fftw, fft_kmath_fft3d.
    |-- 12                        Test program of rnd_kmath_random.
    |-- 13                        Test program of dsvd_scalapack.
    `-- 14                        Test program of dlsm_scalapack.
```

Figure 3.1: The directory structure of the KMATHLIB API package.

Chapter 4

Software Development using KMATHLIB API

The following section explains how to use the KMATHLIB API. The example that is used is a test program (Figure 4.1) that solves a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$ using the ScaLAPACK routine.

Defining the Module and Header Inclusion

In software that uses KMATHLIB API, the `use` statement defines both the `kmath_lib_mod` module for utilizing the core API functions of KMATHLIB API, and the module (in this example, the `kmath_plugin_dls_scalapack_mod` module) for utilizing the plugins called by the program. The header file inclusion is performed to use the PETSc functions. In this example, `finclude/petscsys.h`, `finclude/petscvec.h`, and `finclude/petscmat.h` are included in order to construct matrices and vectors

Variable Declaration and Value Assignment

In the variable declaration area, the standard format (i.e. PETSc data type) is used to declare a variable that stores data to pass to the solver. In the present example, the coefficient matrix A has been declared as `Mat :: A`, and the right-hand side vector $\boldsymbol{b}$ and solution vector $\boldsymbol{x}$ as `Vec :: B`, `Vec :: X`, respectively. In addition, declaration is needed for a variable to store the context handle of KMATHLIB API (`type(s_context) :: h`) and variables to receive the error information returned by each routine (`PetscInt :: ierr` in this example).

The program execution segment performs the value assignment of the declared variable and calls the solver through KMATHLIB API. PETSc functions are used as necessary to assign values to the variables and set them to the data to be passed to the solver. It should be noted that to use the PETSc functions, it is necessary to initialize PETSc.

The Usage of KMATHLIB API

The following section describes the method by which the solver is called through KMATHLIB API. First, the MPI communicator is specified by `KMATH_Create`, KMATHLIB API is initialized, and the context handle is created. The context handle manages the state (setting parameters, etc.) and is maintained until destroyed by `KMATH_Destroy`. Then, the standard format variable is registered as a solver variable through `KMATH\_Set\_Parameter(...)`. In this example, the coefficient matrix variable `Mat :: A` is registered as a solver variable `'A'`; the variables of the

right-hand side vector and solution vector are registered in the same manner. Then, a call to `KMATH_Solve(h,ierr)` automatically calls a solver routine to solve the linear equation. It should be noted that it is necessary to first initialize MPI in order to use KMATHLIB API. Since the initialization of PETSc also initializes MPI, an additional initialization is not required after the initialization of PETSc.

The conversion of the standard and internal formats is done automatically, and users of the KMATHLIB API functions do not need to write extra commands. However, it must be noted that even after calling `KMATH_Solve(h, ierr)`, the calculation results will not be reflected in the registered variable (in this example, `X`) until `KMATH_Destroy(h, ierr)` or `KMATH_Flush(h, ierr)` is called. In the event, the output results are reused to call the same solver again (for example, to additionally solve $A\boldsymbol{y} = \boldsymbol{x}$); the conversion of the results data (stored in internal format) back to standard format is unnecessary. Thus, the KMATHLIB API will not send the results to the registered variables unless explicitly instructed to do so.

```
program test
  use kmath_lib_mod
  use kmath_plugin_dls_scalapack_mod
  implicit none
#include <finclude/petscsys.h>
#include <finclude/petscvec.h>
#include <finclude/petscmat.h>

  Mat :: A
  Vec :: B
  Vec :: X
  ...
  PetscInt             :: ierr
  ...
  type(s_context)      :: h

  call PetscInitialize(PETSC_NULL_CHARACTER, ierr)

  ... (The matrix using the PETSc functions,
       generation of the right-hand side vector, etc.) ...

  ! Solve the linear system A X = B
  call KMATH_Create(h, PETSC_COMM_WORLD, ierr)

  call KMATH_Set_Plugin(h, KMATH_Plugin_Setup_DLS_ScaLAPACK, ierr)
  call KMATH_Set_Parameter(h, 'A', A, ierr)
  call KMATH_Set_Parameter(h, 'B', B, ierr)
  call KMATH_Set_Parameter(h, 'X', X, ierr)
  call KMATH_Solve(h, ierr)

  call KMATH_Destroy(h, ierr)

  ... (Processes that reference calculation results, etc.) ...

  call PetscFinalize(ierr)
end program test
```

Figure 4.1: The main code of `test/06/test.fpp`.

Chapter 5

Specifications of the Interfaces and Standard Plugins

5.1 KMATHLIB API

The following section describes each interface of the KMATHLIB API. All are processed through collective operation, and must be simultaneously called by all MPI ranks of the MPI communicator registered to the handle.

5.1.1 `KMATH_Create` Interface

The KMATHLIB API is initialized by specifying the MPI communicator. The processing necessary for initialization is performed, and returns the context handle.

```
use kmath_lib_mod
subroutine KMATH_Create(handle, comm, ierr)
```

Argument	Type	Input/Output	Description
`handle`	`type(s_context)`	Input	Context handle.
`comm`	`integer`	Input	MPI communicator.
`ierr`	`integer`	Output	Error value. Successful completion if zero.

5.1.2 `KMATH_Destroy` Interface

Ends the KMATHLIB API. The context linked to the handle is released. If the calculation results are pending in internal format, this call will write the contents into the user-side output variable in standard format.

```
use kmath_lib_mod
subroutine KMATH_Destroy(handle, ierr)
```

Argument	Type	Input/Output	Description
`handle`	`type(s_context)`	Input	Context handle.
`ierr`	`integer`	Output	Error value. Successful completion if zero.

5.1.3 `KMATH_Set_Plugin` Interface

Performs plugin registration by specifying a plugin entry point. See Section 5.2 for the names of the plugin entry points of standard plugins. If the plugins are already configured and the calculation results are pending in internal format, this call will write the contents into the user-side output variables in standard format.

```
use kmath_lib_mod
subroutine KMATH_Set_Plugin(handle, comm, ierr)
```

Argument	Type	Input/Output	Description
`handle`	`type(s_context)`	Input	Context handle.
`plugin_setup`	`interface`	Input	Plugin entry point.
`ierr`	`integer`	Output	Error value. Successful completion if zero.

5.1.4 `KMATH_Solve` Interface

Performs the calculation. User-side input variables in standard format are converted to internal format if necessary, and the actual calculation is executed by the plugin. At the end of the call, the calculation result is pending in internal format. If a solution is not reached, returns -1 (`E_SolveFail`) as error value.

```
use kmath_lib_mod
subroutine KMATH_Solve(handle, ierr)
```

Argument	Type	Input/Output	Description
`handle`	`type(s_context)`	Input	Context handle.
`ierr`	`integer`	Output	Error value. Successful completion if zero.

5.1.5 `KMATH_Flush` Interface

If the calculation results are pending in internal format, this call will write the contents to the user-side output variable in standard format.

```
use kmath_lib_mod
subroutine KMATH_Flush(handle, ierr)
```

Argument	Type	Input/Output	Description
`handle`	`type(s_context)`	Input	Context handle.
`ierr`	`integer`	Output	Error value. Successful completion if zero.

5.1.6 `KMATH_Set_Parameter` Interface

Sets the values of the specified parameters. The input/output data variables are also passed to KMATHLIB API through this interface.

```
use kmath_lib_mod
subroutine KMATH_Set_Parameter(handle, key, val, ierr)
```

Argument	Type	Input/Output	Description
`handle`	`type(s_context)`	Input	Context handle.
`key`	`character(*)`	Input	Parameter name.

`val`	`integer`
	`character(*)`
	`double precision`
	`PetscFortranAddr`
		Input	Set value.

`ierr`	`integer`	Output	Error value. Successful completion if zero.

5.1.7 `KMATH_Set_Parameters` Interface

Sets the values of the specified parameters. The input/output data variables are also passed to KMATHLIB API through this interface.

```
use kmath_lib_mod
subroutine KMATH_Set_Parameters(handle, key, val, ierr)
```

Argument	Type	Input/Output	Description
`handle`	`type(s_context)`	Input	Context handle.
`key`	`character(*)(:)`	Input	Parameter name.

`val`	`integer(:)`
	`character(*)(:)`
	`double precision(:)`
	`PetscFortranAddr(:)`
		Input	Set values.

`ierr`	`integer`	Output	Error value. Successful completion if zero.

5.1.8 `s_context` Structure

The structure of the KMATHLIB API context.

```
use kmath_lib_mod
type(s_context) :: context
```

Member	Type	Description
`c`	`integer(8)`	The context entitiy. Stores the pointer value to the memory area allocated from C.

5.2 Standard Plugins

This section describes the specifications of standard plugins.

5.2.1 `dls_ssl2` Plugin

Solves a dense matrix linear system using SSL II.

Plugin module name	`use kmath_plugin_dls_ssl2_mod`
Entry point name	`KMATH_Plugin_Setup_DLS_SSL2`

Parameter	Type	Default value	Input/Output	Description
`A`	`Mat`	None	Input	Coefficient matrix.
`B`	`Vec`	None	Input	Right-hand side vector.
`X`	`Vec`	None	Output	Solution vector.
`matrix_type`	`character(*)`	`"general"`	Input	Matrix type of the coefficient matrix. `"general"` if general matrix. If real symmetric matrix designate as `"symmetric"`.

5.2.2 `dls_lapack` Plugin

Solves a dense matrix linear system using LAPACK.

Plugin module name	`use kmath_plugin_dls_lapack_mod`
Entry point name	`KMATH_Plugin_Setup_DLS_LAPACK`

Parameter	Type	Default value	Input/Output	Description
`A`	`Mat`	None	Input	Coefficient matrix.
`B`	`Vec`	None	Input	Right-hand side vector.
`X`	`Vec`	None	Output	Solution vector.
`matrix_type`	`character(*)`	`"general"`	Input	Matrix type of the coefficient matrix. `"general"` if general matrix. If real symmetric matrix designate as `"symmetric"`.

5.2.3 `dls_scalapack` Plugin

Solves a dense matrix linear system using ScaLAPACK.

Plugin module name	`use kmath_plugin_dls_scalapack_mod`
Entry point name	`KMATH_Plugin_Setup_DLS_ScaLAPACK`

Parameter	Type	Default value	Input/Output	Description
`A`	`Mat`	None	Input	Coefficient matrix.
`B`	`Vec`	None	Input	Right-hand side vector.
`X`	`Vec`	None	Output	Solution vector.
`matrix_type`	`character(*)`	`"general"`	Input	Matrix type of the coefficient matrix. `"general"` if general matrix. If real symmetric matrix designate as `"symmetric"`.

5.2.4 `dlsm_scalapack` Plugin

Solves a dense matrix linear system with multiple right-hand sides using ScaLAPACK.

Plugin module name	`use kmath_plugin_dlsm_scalapack_mod`
Entry point name	`KMATH_Plugin_Setup_DLSM_ScaLAPACK`

Parameter	Type	Default value	Input/Output	Description
`A`	`Mat`	None	Input	Coefficient matrix.
`B`	`Mat`	None	Input	Matrix with right-hand side vectors in each column.
`X`	`Mat`	None	Output	Matrix with solution vectors in each column.
`matrix_type`	`character(*)`	`"general"`	Input	Matrix type of the coefficient matrix. `"general"` if general matrix. If real symmetric matrix designate as `"symmetric"`.

5.2.5 `deig_ssl2` Plugin

Solves the eigenvalue problem of real symmetric dense matrices using SSL II. Operates only if scalar type is a double-precision real number.

Plugin module name	`use kmath_plugin_deig_ssl2_mod`
Entry point name	`KMATH_Plugin_Setup_DEIG_SSL2`

Parameter	Type	Default value	Input/Output	Description
`A`	`Mat`	None	Input	Matrix to calculate eigenpairs.
`X`	`Vec`	None	Output	Vector containing eigenvalues.
`Z`	`Mat`	None	Output	Matrix with the eigenvectors in each column.

5.2.6 `deig_lapack` Plugin

Solves the eigenvalue problem of a matrix using LAPACK. Treated as a real symmetric matrix if build is done using a scalar type as a double-precision real number, and as a Hermitian matrix if build is done as a double-precision complex number.

Plugin module name	`use kmath_plugin_deig_lapack_mod`
Entry point name	`KMATH_Plugin_Setup_DEIG_LAPACK`

Parameter	Type	Default value	Input/Output	Description
`A`	`Mat`	None	Input	Matrix to calculate eigenpairs.
`X`	`Vec`	None	Output	Vector containing eigenvalues.
`Z`	`Mat`	None	Output	Matrix with the eigenvectors in each column.

5.2.7 `deig_scalapack` Plugin

Solves the eigenvalue problem of a matrix using ScaLAPACK. Treated as a real symmetric matrix if build is done using a scalar type as a double-precision real number, and as a Hermitian matrix if build is done as a double-precision complex number.

Plugin module name	`use kmath_plugin_deig_scalapack_mod`
Entry point name	`KMATH_Plugin_Setup_DEIG_ScaLAPACK`

Parameter	Type	Default value	Input/Output	Description
`A`	`Mat`	None	Input	Matrix to calculate eigenpairs.
`X`	`Vec`	None	Output	Vector containing eigenvalues.
`Z`	`Mat`	None	Output	Matrix with the eigenvectors in each column.

5.2.8 `deig_eigenexa` Plugin

Solves the eigenvalue problem of a real symmetric dense matrix using EigenExa. Works correctly only if scalar type is a double-precision real number.

Plugin module name	`use kmath_plugin_deig_eigenexa_mod`
Entry point name	`KMATH_Plugin_Setup_DEIG_EigenExa`

Parameter	Type	Default value	Input/Output	Description
`A`	`Mat`	None	Input	Matrix to calculate eigenpairs.
`X`	`Vec`	None	Output	Vector containing eigenvalues.
`Z`	`Mat`	None	Output	Matrix with the eigenvectors in each column.

5.2.9 `dsvd_scalapack` Plugin

Performs the singular value decomposition (SVD) of a dense matrix using ScaLAPACK.

Plugin module name	`use kmath_plugin_dsvd_scalapack_mod`
Entry point name	`KMATH_Plugin_Setup_DSVD_ScaLAPACK`

Parameter	Type	Default value	Input/Output	Description
`A`	`Mat`	None	Input	Matrix to calculate the SVD.
`S`	`Vec`	None	Output	Vector containing singular values.
`U`	`Mat`	None	Output	Matrix with the left singular vectors in each column.
`VT`	`Mat`	None	Output	Transpose of a matrix with the right singular vectors in each column.

5.2.10 `sls_petsc` Plugin

Solves a sparse linear system using PETSc.

Plugin module name	`use kmath_plugin_sls_petsc_mod`
Entry point name	`KMATH_Plugin_Setup_SLS_PETSc`

Parameter	Type	Default value	Input/Output	Description
`A`	`Mat`	None	Input	Coefficient matrix.
`B`	`Vec`	None	Input	Right-hand side vector.
`X`	`Vec`	None	Output	Solution vector.
`matrix_type`	`character(*)`	`"general"`	Input	Matrix type of the coefficient matrix. `"general"` if general matrix. If real symmetric matrix designate as `"symmetric"`.

5.2.11 `fft_fftw` Plugin

Carries out a discrete Fourier transform using FFTW.

Plugin module name	`use kmath_plugin_fft_fftw_mod`
Entry point name	`KMATH_Plugin_Setup_FFT_FFTW`

Parameter	Type	Default value	Input/Output	Description
`dimension`	`integer`	1	Input	(Described below)
`direction`	`charactr(*)`	`"forward"`	Input	Direction of the Fourier. transformation. Designate `"forward"` if a forward transformation, `"backward"` if an inverse transformation.
`size_x`	`integer`	0	Input	(Described below)
`size_y`	`integer`	0	Input	(Described below)
`size_z`	`integer`	0	Input	(Described below)
`A`	`Mat`	None	Input	Matrix used as input data.
`X`	`Mat`	None	Output	Matrix used as Output data.

Parameter `dimension`

Specify the number of dimensions in the target data of the Fourier transform, in the range 1 (one-dimensional), 2 (two-dimensional), or 3 (three-dimensional). `Mat` type is utilized for input and output data. The relationship between the specified number of dimensions and the storage method of values into the matrix is as follows:

1-dimensional(n)	Create an n-by-1 matrix.
2-dimensional(n, m)	Create an n-by-m matrix.
3-dimensional(n, m, o)	Create an $(n \times m)$-by$o$ matrix.

Parameters `size_x`, `size_y`, `size_z`

Specify the target data size of the Fourier transform. It is necessary to specify only `size_x` if one-dimensional, `size_x` and `size_y` if two-dimensional, and all parameters if three-dimensional.

5.2.12 `fft_kmath_fft3d` Plugin

Carries out a three-dimensional discrete Fourier transform using KMATH FFT3D.

Plugin module name `use kmath_plugin_fft_kmath_fft3d_mod`
Entry point name `KMATH_Plugin_Setup_FFT_KMATH_FFT3D`

Parameter	Type	Default value	Input/Output	Description
`direction`	`charactr(*)`	`"forward"`	Input	Direction of the Fourier. transformation. Designate `"forward"` if a forward transformation, `"backward"` if an inverse transformation.
`size_x`	`integer`	0	Input	Refer to `fft_fftw` plugin.
`size_y`	`integer`	0	Input	Refer to `fft_fftw` plugin.
`size_z`	`integer`	0	Input	Refer to `fft_fftw` plugin.
`A`	`Mat`	None	Input	Matrix used as input data.
`X`	`Mat`	None	Output	Matrix used as Output data.

5.2.13 `rnd_kmath_random` Plugin

Carries out pseudo-random number generation, uniformly-distributed in $[1, 2)$ using KMATH Random.

Plugin module name `use kmath_plugin_fft_rnd_kmath_random_mod`
Entry point name `KMATH_Plugin_Setup_Rnd_KMATH_Random`

Parameter	Type	Default value	Input/Output	Description
`X`	`Vec`	None	Output	The generated pseudo-random number sequence..

Chapter 6

KMATHLIB API — Now and Future

The current KMATHLIB API contains the constraints described in the following list and some functions are planned to be added to the KMATHLIB API. In future roll-outs, improvements and additional functionality may be seen with respect to these points.

1. A large number of numerical libraries (solvers) are used in today's computational science software. However, in the current version of KMATHLIB API, most functions — with the exception of a handful — have not been implemented as standard plugins. To provide additional convenience to KMATHLIB API users, it is necessary to add new functionality to standard plugins, covering even more solvers.

2. In the current KMATHLIB API, the reuse of variables (converted into internal format) occurs only when the same plugins are used consecutively and the output variable of a calculation is used as the input variable in the calculation immediately after. However, it is possible, even with input variables, for the internal format data to be reused in a subsequent calculation if the input variable performs calculations via a non-destructive solver, or if the input variables are separately stored as copies converted into internal format. Furthermore, even when switching to a different plugin, it is possible to reuse the data if keeping the same internal format. However, under the current KMATHLIB API functionality, this would be written back into standard format. It is believed that a mechanism to determine reusability will reduce the overhead associated with data format conversion and lead to performance enhancement of software that uses the KMATHLIB API.

3. During the solver call using KMATHLIB API, there is a mechanism to secure storage space within the plugin for internal-format data. However, in the current KMATHLIB API function group there is no way to know the size of the secured storage space, and error processing is incomplete in the event that necessary memory cannot be secured within the plugin. There is an urgent need for an inquiry function for the secured storage size, and for the implementation of appropriate error handling.

4. Currently, the standard plugin libraries are fixed according to the plugin. For this reason, KMATHLIB API users must themselves determine what library to use, and difficulties may arise for users without a deep understanding of the numerical libraries. As a means to solve this problem, consideration will be given to a plugin that automatically selects an appropriate library in accordance with the set parameters (symmetry and degree of the

matrix, etc.) and the computing environment (SSL II, LAPACK, ScaLAPACK, etc.) for each problem class (dense matrix linear system, dense matrix eigenvalue problem, etc.).

Acknowledgments

The KMATHLIB API has been developed using RIKEN's supercomputer, "K computer" ("K computer" utilization enhancement: case number ra000005 (2013-)). Many thanks to the Advanced Institute for Computational Science at RIKEN for its support.

To test the KMATHLIB API on the Fujitsu PRIMEHPC FX10, we referenced the "PETSc Construction Procedure" published on the University of Tokyo's Oakleaf-FX support portal. We express gratitude to the parties involved in the preparation of this procedure manual.

Bibliography

[1] Fujitsu SSL II User's Guide, 2014 (in Japanese).

[2] Fujitsu SSL II/MPI User's Guide, 2014 (in Japanese).

[3] LAPACK — Linear Algebra PACKage, `http://www.netlib.org/lapack/` (verified 2016-02-15).

[4] ScaLAPACK — Scalable Linear Algebra PACKage, `http://www.netlib.org/scalapack/` (verified 2016-02-15).

[5] S. Balay et al., PETSc Web page, `http://www.mcs.anl.gov/petsc` (verified 2016-02-15).

[6] S. Balay et al., "PETSc Users Manual", 2015.

[7] S. Balay et al., "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries", Modern Software Tools in Scientific Computing, pp. 163–202, Birkhäuser Press, 1997.

[8] F. Matteo and J. G. Steven, "The Design and Implementation of FFTW3", Proceedings of the IEEE, Vol. 93, No. 2, pp. 216–331, 2005.

[9] D. Takahashi, FFTE: A Fast Fourier Transform Package, `http://www.ffte.jp/` (verified 2016-02-15).

[10] EigenExa (a High Performance Eigen-Solver), `http://www.aics.riken.jp/labs/lpnctrt/EigenExa_e.html` (verified 2016-02-15).

[11] KMATH_RANDOM (a High Performance Pseudorandom Number Generator), `http://www.aics.riken.jp/labs/lpnctrt/KMATH_RANDOM_e.html` (verified 2015-02-15).

Appendix A

Plugin Development Tutorial

As mentioned in Section 2.3, users are free to add plugins to the KMATHLIB API. The plugin management function calls in a timely manner the callback routine responsible for the following processes: performing computation, converting standard format data to internal format, and converting the internal format into standard format (Figure A). The plugin developers must properly implement these callback routines and create plugin entry points to register the callback routine with the KMATHLIB API.

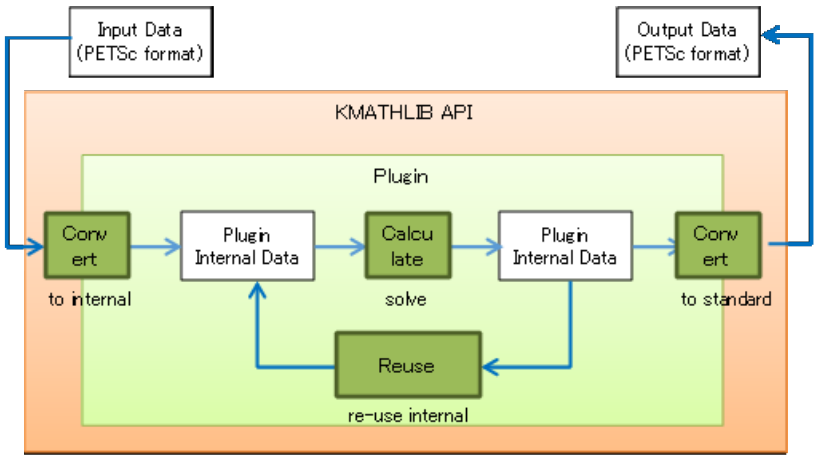

Figure A.1: The data conversion flow and the callback routine within the plugin.

A.1 Plugin Entry Points and Callback Routines

This section explains the methodology for plugin development using as example the included `template` plugin.

Plugin Entry Points

Plugin developers must develop subroutines that become entry points to the plugin with the following specifications. In the user code, the plugin becomes usable by passing this subroutine to the plugin registration interface `KMATH_Set_plugin()` of the KMATHLIB API. The entry point name is optional but must be unique within the program. If the same entry point name exists, a symbol duplication error will occur when linked.

In the template plugin, the entry point name is `KMATH_Plugin_Setup_Template` as shown in Figure A.2.

`subroutine` *plugin_entry_point_name*`(context)`
The subroutine is called by the plugin management function when the `KMATH_Set_plugin()` interface is called. During the call, the context data (equivalent to the member `c` of the `s_context` structure, refer Section 5.1.8) created during the KMATHLIB API initialization, is passed.

Argument	Type	Input/Output	Description
`context`	`integer(8)`	Input	Context data.

Plugin-registered routine `kml_regist_plugin_routines`

Registers the plugin callback routine by calling the current routine once within the subroutine of the plugin entry point. See Figure A.2 for usage examples of the `template` plugin.

Declaration `src/core/kml_pluin_if.h`

```
subroutine kml_regist_plugin_routines(context,
                                      pcontext,
                                      in_parameters,
                                      out_parameters,
                                      mem_parameters,
                                      func_initialize,
                                      func_shutdown,
                                      func_realloc,
                                      func_solve,
                                      func_to_internal,
                                      func_to_standard,
                                      func_reuse_internal)
```

```
...
subroutine KMATH_Plugin_Setup_Template(context)

  ! formal arguments
  integer(8),              intent(in)    :: context

  ! local variables
  type(s_pcontext), pointer :: pcontext


  write(6,*) "KMATH_Plugin_Setup_Template> called."

  allocate(pcontext)
  pcontext%self => pcontext

  call kml_regist_plugin_routines(context,           &
                                  pcontext,          &
                                  "A,B",             &
                                  "X",               &
                                  "P1,P2",           &
                                  kmp_initialize,    &
                                  kmp_shutdown,      &
                                  kmp_realloc,       &
                                  kmp_solve,         &
                                  kmp_to_internal,   &
                                  kmp_to_standard,   &
                                  kmp_reuse_internal)

  return

end subroutine KMATH_Plugin_Setup_Template
...
```

Figure A.2: The plugin entry point of `src/plugins/template/kmath_plugin_template.fpp`.

Argument	Type	Input/Output	Description
`context`	`integer(8)`	Input	Context data.
`pcontext`	`integer(8)`	Input	Plugin context data.
`in_parameters`	`character(*)`	Input	Specify input parameters of the calculation. If multiple parameters exist, separate with commas. The order of the parameters become "0th input parameter," "1st input parameter," etc.
`out_parameters`	`character(*)`	Input	Specify output parameters of the calculation. If multiple parameters exist, separate with commas. The order of the parameters become "0th output parameter," "1st output parameter," etc.
`mem_parameters`	`character(*)`	Input	Specify parameter names to allocate/reallocate memory that stores the internal format. If multiple parameters exist, separate with commas.
`func_initialize`	`subroutine`	Input	Specify callback routine for initialization.
`func_shutdown`	`subroutine`	Input	Specify callback routine for shutdown.
`func_realloc`	`subroutine`	Input	Specify callback routine for reallocation.
`func_solve`	`subroutine`	Input	Specify callback routine for solve.
`func_to_internal`	`subroutine`	Input	Specify callback routine for conversion to internal format.
`func_to_standard`	`subroutine`	Input	Specify callback routine for conversion to standard format.
`func_reuse_internal`	`subroutine`	Input	Specify callback routine for reuse of internal format data.

Plugin Callback Routine

The plugin management function calls the plugin callback routine if necessary during KMATHLIB API processes. The routines that plugin developers need to implement are listed below. The routine names are optional. The errors that are returned are defined in `src/core/kml_error_code.h`.

```
...
subroutine kmp_initialize(context, pcontext, error)

  ! formal arguments
  integer(8),              intent(in)    :: context
  type(s_pcontext),        intent(inout) :: pcontext
  integer,                 intent(out)   :: error

  ! local variables
  integer                  :: comm, rank, nproc, ierr


  call kml_get_mpi_communicator(context, comm)

  call MPI_Comm_rank(comm, rank, ierr)
  call MPI_Comm_size(comm, nproc, ierr)

  write(6,*) "Kmp_Initialize> called. Rank:", rank, " / ", nproc

  call kml_set_error(error, EP_None)
  return

end subroutine kmp_initialize
...
```

Figure A.3: The callback routine for initialization of `src/plugins/template/kmath_plugin_template.fpp`.

`subroutine` ***callback_initialize_name*** `(context, pctxt, error)`

The callback routine for initialization. Implements the initialization of the plugin. See Figure A.3 for an example using the `template` plugin.

Argument	Type	Input/Output	Description
`context`	`integer(8)`	Input	Context data.
`pctxt`	`<any type>`	Input/Output	Plugin context data.
`error`	`integer`	Output	Error information.
			Successful completion `EP_None`
			Abnormal completion `EP_FatalInit`

`subroutine` ***callback_shutdown_name*** `(context, pctxt, error)`

The callback routine for shutdown. Implements the shutdown of the plugin. See Figure A.4 for an example using the `template` plugin.

```
...
subroutine kmp_shutdown(context, pcontext, error)

  ! formal arguments
  integer(8),              intent(in)    :: context
  type(s_pcontext),        intent(inout) :: pcontext
  integer,                 intent(out)   :: error


  write(6,*) "Kmp_Shutdown> called."

  deallocate(pcontext%self)

  call kml_set_error(error, EP_None)
  return

end subroutine kmp_shutdown
...
```

Figure A.4: The callback routine for shutdown of `src/plugins/template/kmath_plugin_template.fpp`.

Argument	Type	Input/Output	Description
`context`	`integer(8)`	Input	Context data.
`pctxt`	`<any type>`	Input/Output	Plugin context data.
`error`	`integer`	Output	Error information.
			Successful completion `EP_None`
			Abnormal completion `EP_FatalShutdown`

`subroutine` ***callback_realloc_name***`(context, pctxt, error)`

The callback routine for reallocation. Implements the reallocation of memory to store internal format data. See Figure A.5 for an example using the `template` plugin.

Argument	Type	Input/Output	Description
`context`	`integer(8)`	Input	Context data.
`pctxt`	`<any type>`	Input/Output	Plugin context data.
`error`	`integer`	Output	Error information.
			Successful completion `EP_None`
			Abnormal completion `EP_FatalRealloc`

`subroutine` ***callback_solve_name***`(context, pctxt, error)`

The callback routine for calculation. Implements calculation processing. See Figure A.6 for an example using the `template` plugin.

```
...
subroutine kmp_realloc(context, pcontext, error)

  ! formal arguments
  integer(8),              intent(in)    :: context
  type(s_pcontext),        intent(inout) :: pcontext
  integer,                 intent(out)   :: error


  write(6,*) "Kmp_Realloc> called. "

  call kml_set_error(error, EP_None)
  return

end subroutine kmp_realloc
...
```

Figure A.5: The callback routine for reallocation of `src/plugins/template/kmath_plugin_template.fpp`.

Argument	Type	Input/Output	Description
`context`	`integer(8)`	Input	Context data.
`pctxt`	`<any type>`	Input/Output	Plugin context data.
`error`	`integer`	Output	Error information.
			Successful completion `EP_None`
			Abnormal completion `EP_FatalRealloc`

```
...
subroutine kmp_solve(context, pcontext, error)

  ! formal arguments
  integer(8),              intent(in)    :: context
  type(s_pcontext),        intent(inout) :: pcontext
  integer,                 intent(out)   :: error


  write(6,*) "Kmp_Solve> called."

  call kml_set_error(error, EP_None)
  return

end subroutine kmp_solve
...
```

Figure A.6: The callback routine to solve `src/plugins/template/kmath_plugin_template.fpp`.

```
...
subroutine kmp_to_internal(context, pcontext, error, in_param, in_petsc)

  ! formal arguments
  integer(8),              intent(in)    :: context
  type(s_pcontext),        intent(inout) :: pcontext
  integer,                 intent(out)   :: error
  integer,                 intent(in)    :: in_param
  PetscFortranAddr,        intent(in)    :: in_petsc


  write(6,*) "Kmp_To_Internal> called.  IN:", in_param, &
             " (", in_petsc, ")"

  call kml_set_error(error, EP_None)
  return

end subroutine kmp_to_internal
...
```

Figure A.7: The callback routine that converts the data to internal format in `src/plugins/template/kmath_plugin_template.fpp`.

`subroutine` ***callback_to_internal_name***`(context, pctxt, error, in_param, in_petsc)`

The callback routine for conversion of the data format to internal format. Implements the conversion of KMATHLIB API input data from standard format to internal format. See Figure A.7 for an example using the `template` plugin.

Argument	Type	Input/Output	Description
`context`	`integer(8)`	Input	Context data.
`pctxt`	`<any type>`	Input/Output	Plugin context data.
`error`	`integer`	Output	Error information. Successful completion `EP_None`, Abnormal completion `EP_FatalToInternal`.
`in_param`	`integer`	Input	A number greater or equal to 0 indicating which input parameter is converted into internal format.
`in_petsc`	`PetscFortranAddr`	Input	The actual standard format data to be converted.

```
...
subroutine kmp_to_standard(context, pcontext, error, out_param, out_petsc)

  ! formal arguments
  integer(8),              intent(in)    :: context
  type(s_pcontext),        intent(inout) :: pcontext
  integer,                 intent(out)   :: error
  integer,                 intent(in)    :: out_param
  PetscFortranAddr,        intent(in)    :: out_petsc


  write(6,*) "Kmp_To_Standard> called. OUT:", out_param, &
             " (", out_petsc, ")"

  call kml_set_error(error, EP_None)
  return

end subroutine kmp_to_standard
...
```

Figure A.8: The callback routine to convert the data to standard format in `src/plugins/template/kmath_plugin_template.fpp`.

`subroutine` ***callback_to_standard_name***`(context, pctxt, error, out_param, out_petsc)`

The callback routine for conversion of the data format to standard format. Implements the conversion of KMATHLIB API output data from internal format to standard format. See Figure A.8 for an example using the `template` plugin.

Argument	Type	Input/Output	Description
`context`	`integer(8)`	Input	Context data.
`pctxt`	`<any type>`	Input/Output	Plugin context data.
`error`	`integer`	Output	Error information. Successful completion `EP_None` Abnormal completion `EP_FatalToStandard`
`out_param`	`integer`	Input	A number greater or equal to 0 indicating which output parameter is converted into standard format.
`out_petsc`	`PetscFortranAddr`	Input	The actual internal format data to be converted.

`subroutine` ***callback_to_reuse_name***`(context, pctxt, error, out_param, in_param)`

The callback routine for reuse of the internal format data. Implements the reuse of internal format data. This routine is called in the event of a user code such as the following:

```
call KMATH_Set_Parameter(h, 'InputParam', A, ierr)
call KMATH_Set_Parameter(h, 'OutputParam', X, ierr)
call KMATH_Solve(h, ierr)

call KMATH_Set_Parameter(h, 'InputParam', X, ierr)
call KMATH_Set_Parameter(h, 'OutputParam', Y, ierr)
call KMATH_Solve(h, ierr)
```

After the first call to `KMATH_Solve`, the data in variable `X` has not yet been converted from the internal format of `'OutputParam'`. This callback routine copies the internal format data of `'OutputParam'` to the internal format data of `'InputParam'` (a copy process is required to be implemented by the developer). See Figure A.9 for an example using the `template` plugin.

Argument	Type	Input/Output	Description
`context`	`integer(8)`	Input	Context data.
`pctxt`	`<any type>`	Input/Output	Plugin context data.
`error`	`integer`	Output	Error information. Successful completion `EP_None` Abnormal completion `EP_FatalToReuse`
`out_param`	`integer`	Input	A number greater or equal to 0 indicating which output parameter will become reuse destination.
`in_param`	`integer`	Input	A number greater or equal to 0 indicating which input parameter will become reuse destination.

```
...
subroutine kmp_reuse_internal &
           (context, pcontext, error, out_param, in_param)

  ! formal arguments
  integer(8),              intent(in)    :: context
  type(s_pcontext),        intent(inout) :: pcontext
  integer,                 intent(out)   :: error
  integer,                 intent(in)    :: out_param
  integer,                 intent(in)    :: in_param


  write(6,*) "Kmp_Reuse_Internal> called. OUT:", out_param, &
             " => IN:", in_param

  call kml_set_error(error, EP_None)
  return

end subroutine kmp_reuse_internal
...
```

Figure A.9: The callback routine to reuse the internal format data in `src/plugins/template/kmath_plugin_template.fpp`.

A.2 Description of Subroutines Available to the Plugin

This section describes the subroutines that are available to the plugins.

Subroutine `kml_get_mpi_communicator`

Returns the MPI communicator linked to the specified context.

Declaration `src/core/kml_pluin_if.h`

```
subroutine kml_get_mpi_communicator(handle, comm)
```

Argument	Type	Input/Output	Description
`handle`	`integer(8)`	Input	Context handle.
`comm`	`integer`	Output	MPI communicator.

Subroutine `kml_get_parameter_int`

Gets the parameter value of the specified name as an integer.

Declaration `src/core/kml_pluin_if.h`

```
subroutine kml_get_parameter_int(handle, key, val, err)
```

Argument	Type	Input/Output	Description
`handle`	`integer(8)`	Input	Context handle.
`key`	`character(*)`	Input	Parameter name.
`val`	`integer`	Output	Acquired parameter value as an integer.
`err`	`integer`	Output	Error value. `EI_None`(= 0) Successful completion. `EI_ParamNotFound`(= 200) The parameter of the specified name does not exist. `EI_FatalGetParam`(= 201) Type mismatch with set value.

Subroutine `kml_get_parameter_real`

Gets the parameter value of the specified name as a real number.

Declaration `src/core/kml_pluin_if.h`

```
subroutine kml_get_parameter_real(handle, key, val, err)
```

Argument	Type	Input/Output	Description
`handle`	`integer(8)`	Input	Context handle.
`key`	`character(*)`	Input	Parameter name.
`val`	`double precision`	Output	Acquired parameter value as real.
`err`	`integer`	Output	Error value. `EI_None`(= 0) Successful completion. `EI_ParamNotFound`(= 200) The parameter of the specified name does not exist. `EI_FatalGetParam`(= 201) Type mismatch with set value.

Subroutine `kml_get_parameter_string`

Gets the parameter value of the specified name as a string.

Declaration `src/core/kml_pluin_if.h`

```
subroutine kml_get_parameter_string(handle, key, val, err)
```

Argument	Type	Input/Output	Description
`handle`	`integer(8)`	Input	Context handle.
`key`	`character(*)`	Input	Parameter name.
`val`	`character(*)`	Output	Acquired parameter value as a string.
`err`	`integer`	Output	Error value. `EI_None`(= 0) Successful completion. `EI_ParamNotFound`(= 200) The parameter of the specified name doest not exist. `EI_FatalGetParam`(= 201) Type mismatch with set value.

Subroutine kml_get_parameter_petsc

Gets the parameter value of the specified name as type PetscFortranAddr.

Declaration src/core/kml_pluin_if.h

```
subroutine kml_get_parameter_petsc(handle, key, val, err)
```

Argument	Type	Input/Output	Description
handle	integer(8)	Input	Context handle.
key	character(*)	Input	Parameter name.
val	PetscFortranAddr	Output	Acquired parameter value as type PetscFortranAddr.
err	integer	Output	Error value. EI_None(= 0) Successful completion. EI_ParamNotFound(= 200) The parameter of the specified name doest not exist. EI_FatalGetParam(= 201) Type mismatch with set value.

Subroutine kml_get_row_col

Queries the number of process rows and process columns from the specified process number.

Declaration src/core/kml_pluin_if.h

```
subroutine kml_get_row_col(mprocs, nrow, ncol)
```

Argument	Type	Input/Output	Description
nprocs	integer	Input	Process number.
nrow	integer	Output	Queried process row count.
ncol	integer	Output	Queried process column count.

Subroutine kml_get_prime_factor

Resolves the specified value into prime factors.

Declaration src/core/kml_pluin_if.h

```
subroutine kml_get_prime_factor(val, factors)
```

Argument	Type	Input/Output	Description
val	integer	Input	The input value for prime factorization.
factors	integer(:)	Output	Array of prime factors.

Subroutine `kml_tolower_case`

Converts the specified string to lowercase.

Declaration `src/core/kml_pluin_if.h`

```
subroutine kml_tolower_case(str)
```

Argument	Type	Input/Output	Description
`str`	`character(*)`	Input/Output	String to be converted.

Subroutine `kml_set_error`

Sets the error code. If the preprocessor macro `DEBUG` is enabled, the subroutine outputs an error string that corresponds to the error code, in response to a standard input.

Declaration `src/core/kml_pluin_if.h`

```
subroutine kml_set_error(error, code)
```

Argument	Type	Input/Output	Description
`error`	`integer`	Input/Output	Input variable for error code setting.
`code`	`integer`	Input	Error code.

Appendix B

Data Conversion and Reuse during Interface Calls

When the interface `KMATH_Solve()` is executed, the variables given as output parameters are registered to the hash table, which stores information about whether a calculation result has been obtained. The next time `KMATH_Solve()` is executed, the subroutine searches for the input variables in this hash table. If found, the procedure assumes that the internal format data of the previous output result is valid and a callback routine is invoked for reuse. If not found, a callback routine is called to convert from standard to internal format.

The following table shows the processes that occur after each interface call:

Interface	Processing order	Description
`KMATH_Create`		None.
`KMATH_Destroy`	1	Convert from internal to standard format.
	2	Execute shutdown procedure.
`KMATH_Set_Plugin`	1	Convert from internal to standard format.
	2	Switch plugin (new registration) and initialize.
`KMATH_Solve`	1	Convert from standard to internal format, Otherwise reuse.
	2	Convert from internal to standard format.
	3	Execute calculation. New output results (internal format) are stored into the hash table.
`KMATH_Flush`	1	Convert from internal to standard format.
`KMATH_Set_Parameter`	1	Convert from internal to standard format.
	2	Reallocate.