Max Planck Institute Magdeburg
Preprints

Björn Baran Martin Köhler Nitin Prasad Jens Saak

Interfacing C-M.E.S.S. with Python

MPIMD/13-22 **November 19, 2013**

Abstract

The M.E.S.S. software suite is the successor of the obsolete LyaPack MATLAB® toolbox for solving large scale matrix equations and related problems. The software suite consists of a new MATLAB toolbox and a separate C library C-M.E.S.S. which works independent from MATLAB. Due to the fact that many scientists use Python with NumPy and SciPy for their everyday work we want to provide the key algorithm of M.E.S.S. for them, too. This report describes how we build an interface between Python and C-M.E.S.S. on top of the NumPy/SciPy Python libraries.

Impressum:

Max Planck Institute for Dynamics of Complex Technical Systems, Magdeburg

Publisher:
Max Planck Institute for
Dynamics of Complex Technical Systems

Address:
Max Planck Institute for
Dynamics of Complex Technical Systems
Sandtorstr. 1
39106 Magdeburg

http://www.mpi-magdeburg.mpg.de/preprints/

Contents

1 Introduction

M.E.S.S. (Matrix Equations Sparse Solver) is a toolbox intended to solve large sparse matrix equations like Lyapunov and Algebraic Riccati Equations. Currently, M.E.S.S. is available as a MATLAB toolbox and a separate C library. Besides C, Fortran and MATLAB the Python programming language together with the NumPy[3] and SciPy[6] packages becomes more and more popular in scientific computing.

The NumPy package provides a great and flexible implementation of n-dimensional arrays and basic linear algebra in Python. The SciPy package additionally provides a huge set of high level algorithms for scientific computations. Starting with advanced linear algebra operations and the ability to work with sparse matrices it also includes the Fast Fourier Transformations, ODE solvers and many other features that are known from MATLAB. Because of the increasing popularity we want to provide the M.E.S.S. functionality for Python, too. Obviously we can develop such a Python package in

two different ways. On the one hand, we can rewrite all important algorithms like the Low Rank Cholesky Factor ADI or the Low Rank Newton Method [13, 18] using pure NumPy and SciPy features. On the other hand, we can develop an interface between Python and C-M.E.S.S. and use the already implemented algorithms from the C library. The later of those two ways is focus of this paper.

In the following sections we describe how the basic C interface of NumPy works and how we convert the internal data structures of M.E.S.S. to the corresponding ones of NumPy and SciPy. Afterwards we derive a Python interface for the ADI algorithm which is similar to the one we provide in the MATLAB implementation one.

2 Python-C Module Initialization

Before we take a look at the conversion of data types between NumPy/SciPy and C-M.E.S.S. we give a brief overview how a C extension for Python looks like. The Python developers provide a good starting point for the development of an extension [2]. Basically, a Python-C extension consists of the following components:

1. Inclusion of the Python header `python.h`.
2. Inclusion of header files for additional functions and data structures, like the NumPy array API and C-M.E.S.S..
3. Implementation of the Python interface for all functions that should be available in Python because direct calls to C functions are only possible if the C function takes standard C data types as inputs. See [1] for more details about this.
4. A initialization function which registers the previously defined functions in the name space of the Python interpreter.

Additionally to these four basic parts of a Python-C extension we have to take care of some specialties that have changed in the Python API between the two major versions Python 2.x and Python 3.x [7]. Most of the changes do not affect our development but for the case where we have to distinguish between the old and the new Python APIs there exists a C macro which indicates the API version. Using the `PY_MAJOR_VERSION` preprocessor constant we use API calls that differ between both versions in the following way:

```
#if PY_MAJOR_VERSION < 3
// Code for the old API
#else
// Code for the new API
#endif
```

The first two parts of the component list for the Python-C extension result in the following head for the C code:

```
#include <Python.h>
#define PY_ARRAY_UNIQUE_SYMBOL MESS_VECTOR_MATRIX_PYTHON_C_API
```

```
#include <numpy/arrayobject.h>
#include "mess/mess.h"
```

Thereby, we have to define a unique identifier for the NumPy API. The preprocessor constant PY_ARRAY_UNIQUE_SYMBOL must be set to a unique name before we include numpy/arrayobject.h. This symbol is necessary for NumPy to keep track to which module the current NumPy instance belongs. If the module consists of more than one C source file every source file has to define this unique symbol before it includes NumPy.

Because of the unification of classic and UTF-8 strings in Python 3 [7] we have to define a macro to convert Python strings to C strings depending on the API version:

```
#if PY_MAJOR_VERSION < 3
#define ConvStringtoC(X) PyBytes_AsString(X)
#else
#define ConvStringtoC(X) PyBytes_AsString(PyUnicode_AsUTF8String(X))
#endif
```

This macro is necessary later to get information about data types or underlying classes from a Python object without dealing with the different APIs every time.

Every function we want to export from C to Python must be registered in the Python name space when the module is loaded using import. Therefore we list all functions that should be available from Python in a designated array. Each element of the array consists of four components. The first one is a string containing the name of the function from the Python point of view. The second one is the name of the function inside the C interface. More precisely this is a function pointer to the corresponding function in the Python-C interface. The third component is a flag which indicates how the function handles the given function arguments. Python knows two ways to identify the function arguments. On the one hand, the position of a function argument determines its meaning. This behavior is well known from almost all programming languages. If the flag is set to METH_VARARGS this behavior is used. On the other hand, Python can identify the function arguments by keywords, that means that the function arguments can appear in an arbitrary order as long as they are all marked with a keyword. This behavior is enabled by performing an or operation on METH_KEYWORDS and the flag. The last component of each function entry is a documentation string. The string can be retrieved from the function in Python by calling:

```
print(modulename.function_name.__doc__)
```

or in IPython:

```
modulename.function_name?
```

The last entry in the function list is a NULL entry to identify the end of the list without any external counter. The function list for our extension looks like:

```
static PyMethodDef interface_python_methods[] = {
 {"eps", Pymess_eps, METH_VARARGS,
      "Return the machine epsilon."},
```

```
 {"lradi", Pymess_lradi,
      METH_VARARGS | METH_KEYWORDS,
      "Interface to the C.M.E.S.S. LRCFADI function."},
 {NULL, NULL, 0, NULL}
};
```

If we want to add new functions to our Python-C extension we only have to write the interface function and simply create a new entry in this list of exported functions. The rest of the code is untouched.

The initialization function of the module is the largest difference between Python-2 and Python-3. Therefore we present both variants separately, although in the real implementation they are combined to one initialization function because most of the inner part is valid for both cases [5].

2.1 Python-2 C API Initialization

The initialization function for a Python-2 module looks basically like:

```
struct module_state {
    PyObject *error;
};

#define GETSTATE(m) (&_state)
static struct module_state _state;

#define INITERROR return
void initpycmess(void) {
 PyObject *module = Py_InitModule("pycmess", interface_python_methods
     );

 if (module == NULL)
  INITERROR;

 struct module_state *st = GETSTATE(module);
 st->error = PyErr_NewException("pycmess.Error", NULL, NULL);
 if (st->error == NULL) {
  Py_DECREF(module);
  INITERROR;
 }
 // Additional code
 import_array();
}
```

The most important thing in this case is the name of this function. It must be called `initTHEMODULENAME`. In our case the Python module should be called `pycmess` so the resulting name of the initialization function is `initpycmess`. If the function name is spelled differently the Python interpreter can not call the initialization and the module is not usable. The most important function is `Py_InitModule`. It

registers the previously defined function list with the corresponding module name in the Python name space. The first argument which represents the module name again must fit to the name of the initialization function without the `init` prefix. If the `Py_InitModule()` call is not able to register the module successfully, an error is raised. After a successful registration of our module at the Python interpreter we can perform some additional initializations like loading the NumPyAPI in our case.

Reference Counting. The `Py_DECREF()` function is a special function for the memory management in Python. Python and its C-API do not directly rely on the C memory management system. On top of this Python implements a garbage collector and a reference counting system. Each object in the memory has a reference counter. Every time the object is used by a function the reference is increased by one. If a function does no longer need an object it decrements the reference by one. If the value of the reference counter is zero the object is no longer used by any function and can be freed. For our example that means if the `PyErr_NewException` returns `NULL` the error exception is not created and the module initialization has to fail. To this it decrements the reference on the `module` such that this will be thrown away by the garbage collector. Because of the reference counting concept we have to take care every time we retrieve a value from a Python object. Some functions will return new references to objects and values, that means their reference counter is incremented. Other functions only return a *borrowed* reference which means the reference counter is not incremented. If we got a new reference to an object we have to call `Py_DECREF` when we no longer need or access it. If we have to increase the reference manually we call `PY_INCREF` on the desired object. In order to remove an object completely from the memory we have to call `Py_DECREF` as many times as there is a reference on it.

2.2 Python-3 C API Initialization

In contrast to the Python-2 API a Python-3 module does no longer store its internal state in a global variable like the static `_state` structure in the Python-2 source code [19]. Python-3 keeps track of the state data of every module. That requires that not only the functions but also the internal state must be registered at the Python interpreter. Additionally we need two functions to iterate over the internal state and to clear it. All information about the module, like its name, the exported functions and the handling of the internal state are collected in the `PyModuleDef` structure. The following source code shows how this initialization looks in this case:

```
struct module_state {
    PyObject *error;
};
static int state_traverse(PyObject *m, visitproc visit, void *arg) {
    Py_VISIT(GETSTATE(m)->error);
    return 0;
}
static int state_clear(PyObject *m) {
    Py_CLEAR(GETSTATE(m)->error);
```

```
    return 0;
}
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "pycmess",
    NULL,
    sizeof(struct module_state),
    interface_python_methods,
    NULL,
    state_traverse,
    state_clear,
    NULL };

#define INITERROR return NULL
PyObject* PyInit_pycmess(void) {
    PyObject *module = PyModule_Create(&moduledef);

    if (module == NULL)
        INITERROR;
    struct module_state *st = ((struct module_state*)
                               PyModule_GetState(module));
    st->error = PyErr_NewException("interface_python.Error", NULL,
        NULL);
    if (st->error == NULL) {
        Py_DECREF(module);
        INITERROR;
    }
    // Additional Code
    init_array();
    return module;
}
```

Another difference between both API versions is the naming rule for the initialization function and its return type. For Python-3 this has to fit the following scheme: `PyInit_THEMODULENAME`. Additionally the function now returns the pointer to the initialized module instead of `void`. Using a set of `#if ... #else ... #endif` preprocessor statements we can combine both API variants to one single initialization part where we only have to modify our code once if we integrate new features.

In the following to sections we describe a set of helper functions that are necessary to move data between NumPy/SciPy and C-M.E.S.S..

3 Moving matrices and vectors to C-M.E.S.S.

In this section we discuss how the functions `matrix_to_c` and `vector_to_c` work and what the caveats during the implementation are. The function signature of these two functions are

```
mess_matrix matrix_to_c(PyObject *data);
mess_vector vector_to_c(PyObject *data);
```

Both functions should check if a corresponding matrix or vector type exists in C-M.E.S.S. and adjust the data types. For example if the matrix is a single precision one or an integer one in Python it is converted to double precision because C-M.E.S.S. only supports double precision values in real and complex arithmetics.

3.1 The matrix convert function

NumPy and SciPy support a huge set of dense and sparse matrix storage formats. For example the dense matrices can be stored row-major (C style) or column-major (Fortran style). Sparse matrices are available in Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Block Sparse Row (BSR), Coordinate (COO), Diagonal (DIA), Dictionary of Keys (DOK), and Link List (LIL) storage. For our interface we restrict to both dense storage schemes and the three sparse matrix formats supported by C-M.E.S.S.. These are the CSR, the CSC and the COO storage scheme. If we want to pass one of the other formats to our interface we have to convert them in Python before we call a function from our module. Therefore each sparse matrix in Python has some methods like `.tocsr()`, `.tocsc()` or `.tocoo()`.

The conversion function should work as follows: It takes a Python object as an input and inspects whether it is a matrix and a corresponding matrix type in C-M.E.S.S. exists. If the conversion is successful it returns a newly created `mess_matrix` object filled with the data from the Python matrix. In order to work with the data properly the original data is copied to new object instead of only referenced. If the matrix can not be converted the function returns `NULL` and an error is raised. The skeleton of the function is the following:

```
mess_matrix matrix_to_c(PyObject *data) {
 PyObject *module = NULL;
 if(PyObject_HasAttrString(data, "__module__")) {
   module = (PyObject*) PyObject_GetAttrString(data, "__module__");

   //Convert a CSR matrix to C
   if (strcmp(ConvStringtoC(module),"scipy.sparse.csr") == 0) {
      // ...
      Py_DECREF(module);  return csr_matrix;
   }

   //Convert a CSC  matrix to C
   if (strcmp(ConvStringtoC(module),"scipy.sparse.csc") == 0) {
      // ...
      Py_DECREF(module);  return csc_matrix;
   }

   //Convert a COO matrix to C
   if (strcmp(ConvStringtoC(module),"scipy.sparse.coo") == 0) {
```

```
      // ...
      Py_DECREF(module);  return coo_matrix;
   }
   Py_DECREF(module);
 }

 // Convert a Dense matrix to C
 if (PyArray_Check(data)) {
   mess_int_t i;
   PY_GET_LONG(i, data,"ndim");
   if (i != 2) {
      PyErr_SetString(PyExc_TypeError,
          "Dense matrix must be a two dimensional array.");
      return NULL;
   }
   // ...
   return dense_matrix;
 }
 PyErr_SetString(PyExc_TypeError,
      "Argument type should be a dense matrix,
       2d-array or a scipy.sparse matrix (coo, csr or csc)");
 return NULL;
}
```

First we check if the given Python object has a `__module__` property and if this fits to one of the three supported sparse storage formats from SciPy. Therefore we use the `ConvStringtoC` macro, which we introduced in Section 2. If none of these works we check if the given data is a two dimensional array which can be transfered to a dense matrix. If any other Python object is passed to the `matrix_to_c` function it results in an error and returns a `NULL` pointer. As already mentioned in Section 2, Python uses reference counting for the memory management. That is why we have to decrement the reference on the `module` variable after we accessed it the last time. Otherwise, the garbage collector will not recognize that this data set is not longer in use and will not free it.

The `PY_GET_LONG` macro in the previous code snippet helps us to get the `ndim` attribute out of the `data` object easily. It is defined as

```
#define PY_GET_LONG(dest, obj, name ) { \
 PyObject *pv = PyObject_GetAttrString(obj, name); \
 if (pv != NULL) {
  dest = PyLong_AsLong(pv);\
  Py_DECREF(pv); \
 }}
```

and works follows: It fetches the attribute called `name` from the object `obj` which creates a new reference on this attribute. If this was successful it tries to convert the value of the attribute to a long integer and writes it to `dest`. Afterwards, the reference on the attribute is decremented. If it fails it leaves the value of `dest` as it is.

3.1.1 Dealing with sparse matrices

Because all three supported matrix storage formats rely on the same internal structure, i.e. one array for the values and two integer arrays to store the corresponding positions in the matrix, we restrict to the Compressed Sparse Row storage here. The CSC and the COO storage schemes work analogously. The whole procedure of converting a sparse matrix to C-M.E.S.S. consists of three steps:

1. Get the number of rows, number of columns and the number of non zero elements from the matrix,
2. Get the data type of the values from the object and convert it either to `double` or to `double complex` values.
3. Fetch both index vectors and convert the integer type to `mess_int_t`.

The first step is realized using the following piece of code:

```
PyObject *temp = NULL, *temp2 = NULL;
mess_matrix csr_matrix;
mess_matrix_init(&csr_matrix);
PY_GET_LONG(csr_matrix->nnz, data, "nnz");
temp = (PyObject*) PyObject_GetAttrString(data, "shape");
temp2 = (PyObject*) PyTuple_GetItem(temp, 0);
csr_matrix->rows = (mess_int_t) PyLong_AsLong(temp2);
temp2 = (PyObject*) PyTuple_GET_ITEM(temp, 1);
csr_matrix->cols = (mess_int_t) PyLong_AsLong(temp2);
Py_DECREF(temp);

csr_matrix->store_type = MESS_CSR;
csr_matrix->symmetry = MESS_GENERAL;
```

The number of non zero elements is extracted directly out of the object from the `nnz` property using the `PY_GET_LONG` macro. The dimension of the matrix is stored in the `shape` attribute as a tuple `(rows,cols)`. The extraction of this works in two steps. First we get the tuple from the `shape` attribute and save it in a temporary variable. Afterwards we can pick the first and the second value in the tuple via `PyTuple_GET_ITEM` and convert them to proper integer values. We only need to decrement the reference on the `shape` variable because `PyTuple_GET_ITEM` only returns a borrowed reference which is not incremented before. Because SciPy does not distinguish between symmetric or non-symmetric matrices we set the `symmetry` property of the matrix to `MESS_GENERAL`.

The second step is to extract the values of the matrix from the Python object. The two Python data types `float64` and `complex128` correspond directly to the `double` and `double complex` types in C-M.E.S.S.. They can be copied directly to C using `memcpy`. All other types which present a non complex value, such as `float32` or one of the various integer types, must be converted to `float64` before. If the values are of the single precision complex type `complex64` they must be converted to `complex128`. The data type can be extracted out of the object by reading the

dtype attribute. Additionally integer data can be checked using PyArray_ISSIGNED and PyArray_ISUNSIGNED function. This avoids to check for every of the available integer types. The desired matrix data is contained in an array hidden behind the data attribute. The whole procedure to get the data copied from Python to C is shown in the following code snippet:

```
temp = (PyObject*) PyObject_GetAttrString(data, "dtype");
PyObject *dtype = (PyObject*) PyObject_GetAttrString(temp,"name");
char *dtype_str = ConvStringtoC(dtype);
Py_DECREF(temp);

PyObject *value_data =(PyObject*)PyObject_GetAttrString(data,"data");
if ( strcmp(dtype_str,"float64") == 0
     || strcmp(dtype_str,"float32") == 0
     || PyArray_ISSIGNED(value_data)
     || PyArray_ISUNSIGNED(value_data)) {
      PyObject *values = NULL;
     csr_matrix->values = malloc(sizeof(double)*csr_matrix->nnz);
     if ( strcmp(dtype_str,"float64") != 0 ) {
       values = PyArray_FROM_OT(value_data,NPY_DOUBLE);
     } else {
       values = value_data;
     }
     memcpy(csr_matrix->values, PyArray_DATA(values),
            sizeof(double)*csr_matrix->nnz);
     csr_matrix->data_type = MESS_REAL;
     if ( values != value_data) { Py_DECREF(values); }
}
else if (strcmp(dtype_str,"complex128") == 0
          ||strcmp(dtype_str,"complex64") == 0) {
     PyObject *values = NULL;
     csr_matrix->values_cpx = malloc(sizeof(double complex)
                                     *csr_matrix->nnz);
     if ( strcmp(dtype_str,"complex64") == 0 ) {
       values = PyArray_FROM_OT(value_data,NPY_COMPLEX128);
     } else {
       vlues = value_data;
     }
     memcpy(csr_matrix->values_cpx,PyArray_DATA(values),
            sizeof(double complex)*csr_matrix->nnz);
     csr_matrix->data_type = MESS_COMPLEX;
     if ( values != value_data) { Py_DECREF(values); }
}
else {
     Py_DECREF(dtype); Py_DECREF(value_data); Py_DECREF(module);
     PyErr_SetString(PyExc_TypeError, "Argument Matrices should
        have either  integer, float32/64 or complex64/128 entries");
     return NULL;
}
```

```
Py_DECREF(dtype);
Py_DECREF(value_data);
```

The key ingredients in this part are the `PyArray_FROM_OT` and the `PyArray_DATA` function. The first one allows us to convert an arbitrary array to an array with a desired data type. We use this function to ensure that we have either one of the two supported data types in C-M.E.S.S.. If the data type is already `float64` or `complex128` we do not call this function. As of NumPy 1.6.1 each `PyArray_FROM_OT` should be preceded by a `PyErr_Clear()` call because it internally checks the error state of the runtime system without ensuring a proper error state before. The second important function `PyArray_DATA` returns a pointer to the data of the array. This pointer can be cast to the corresponding C data type or can be used as input for `memcpy` to copy the array data to a new previously allocated array. Depending on the data type we allocate the `values` or the `values_cpx` component of the C-M.E.S.S. matrix and copy the data to this location. The data copy is necessary because otherwise an access to the C-M.E.S.S. matrix can damage the linked Python object. The reason behind this is that we cannot ensure that there are no pointer arithmetic and no memory management related operations performed on the matrix. In such a case the Python interpreter would crash.

Moving the index vectors from Python to C results in some platform specific problems regarding the integer sizes. NumPy uses an integer called `npy_int` as default. Depending on the operating system and the hardware platform this can be a 32bit or a 64bit integer. Inside C-M.E.S.S. we have the same problem. The type `mess_int_t` can be 32bit or 64 bit, as well. Obviously, we can only copy the values directly in two cases. Otherwise the integer sizes differ and the result after the copy procedure only contains unusable data. The problem can be solve by introducing a temporary array. This array contains the 64bit integer representation of the values and can be cast to any other possibly smaller integer type. This temporary array is created by using the already known `PyArray_FROM_OT` function again. The pointer which is extracted from the temporary array afterwards has the type `npy_int64*`. If we now iterate over the array we can cast each element correctly to `mess_int_t`. The procedure for the row indices contained in the `indptr` attribute of a CSR matrix looks like:

```
temp = (PyObject*) PyObject_GetAttrString(data, "indptr");
PyObject * indptr = PyArray_FROM_OT(temp, NPY_INT64);
array = (npy_int64 *) PyArray_DATA(indptr);
csr_matrix->rowptr = malloc(sizeof(mess_int_t)*(csr_matrix->rows+1));
for (i = 0; i < csr_matrix->rows+1; i++) {
  csr_matrix->rowptr[i] = (mess_int_t) array[i];
}
Py_DECREF(temp);
Py_DECREF(indptr);
```

The column indices are extracted in the same way out of the `indices` attribute. After all indices are converted we have to decrement the reference of all temporarily created Python objects to tell the garbage collector that they can be removed from memory.

Finally, we only need to return the created `csr_matrix`. This matrix can be used like any other matrix inside the C-M.E.S.S. library.

3.1.2 Dealing with dense matrices

Dense matrices are represented as NumPy nd-arrays with two dimensions. Internally the values are stored in C style or Fortran style arrays. Additionally, NumPy allows to define so called slices on arrays. A slice in the Python context is a view on a sub-array (defined by a continuous or equally strided index set) which can be used like a ordinary array but shares the same data. That means if we access an element in the sliced matrix we access the corresponding element in the original matrix. The following example creates a slice on the the sub-matrix $A(1:2, 4:5)$ of a matrix A:

```
Asub = A[1:3,4:6]
```

The different upper indices are cause by the Python `:`-operator which does not include the upper bound like MATLAB does. Each access to `Asub` is redirected to the corresponding part in A. Even slices that only access every second element of an array are possible. The slices work for C style storage, as well as, for the Fortran style storage. If we want to implement this by a distinction of cases we result in four complicated cases each for real and complex arithmetics. If the matrices are stored in C style we might have to transpose them again to get the corresponding Fortran storage scheme because C-M.E.S.S. only supports dense matrices in Fortran style. The reason behind this is that C-M.E.S.S. strongly relies on the FORTRAN77 interface to BLAS and LAPACK.

Fortunately, the NumPy API provides some helper functions which enable us to handle all four cases in only one step per data type. Each array has a stride information for each dimension. This property contains the information how many positions in memory the next element in the same dimension is away from the current one. Consider we have a stride s_{row} for the rows and a stride s_{col} for the columns then the element (i, j) of a matrix, with zero based indexing, is at position

$$p := i \cdot s_{\text{row}} + j \cdot s_{\text{col}} \tag{1}$$

If we have a ordinary matrix $A \in \mathbb{R}^{m \times n}$ which is no slice then the following conditions are true: If the matrix is stored in C storage $s_{\text{row}} := n$ and $s_{\text{col}} = 1$. If the matrix is stored like a Fortran array then $s_{\text{row}} := 1$ and $s_{\text{col}} = m$. In the case of sliced matrices the row and the column stride are set to values that corresponds to the slice.

If we now take a look on the implementation we first create our new dense matrix and extract the size and the stride information out of the Python array:

```
mess_matrix dense_matrix;
mess_matrix_init(&dense_matrix);
dense_matrix->rowptr = NULL;
dense_matrix->colptr = NULL;

rows = (mess_int_t) PyArray_DIM(data,0);
cols = (mess_int_t) PyArray_DIM(data,1);
```

```
stride_rows = PyArray_STRIDE(data,0);
stride_cols = PyArray_STRIDE(data,1);
```

In contrast to the sparse matrices, the size information can be extracted directly from the array by calling `PyArray_DIM` for each dimension. The `PyArray_STRIDE` function returns the stride for every dimension in bytes. In order to get the stride counted in elements we have to divide by the size of element later on.

The detection of the data types works as for the sparse matrices. The handling of other types than `float64` and `complex128` can be realized in the same way as for the sparse matrices or by implementing it for every allowed data type. First we have to figure out the size of one element to adjust the stride information. Afterwards we can copy each element to a Fortran style array by iterating over the whole matrix. Therefore we compute the position in the NumPy array using (1). Hence, this access strategy works for C and Fortran style storage we do no longer need to transpose a C style matrix after we copied the values. The following source code shows this procedure:

```
temp = (PyObject*) PyObject_GetAttrString(data, "dtype");
dtype = (PyObject*) PyObject_GetAttrString(temp, "name");
dtype_str = strdup(ConvStringtoC(dtype));
Py_DECREF(temp);
Py_DECREF(dtype);
if ( strcmp(dtype_str, "float64") == 0  ) {
  type_size = sizeof(double);
  data_type = MESS_REAL;
} else if ( strcmp(dtype_str, "complex128") == 0 ) {
  type_size = sizeof(double complex);
  data_type = MESS_COMPLEX;
} else {
  // Throw error
}

mess_matrix_alloc(dense_matrix, rows, cols,
                  rows*cols,MESS_DENSE, data_type);
stride_rows/= type_size;
stride_cols/= type_size;

if ( strcmp(dtype_str, "float64") == 0  ) {
  double *values = PyArray_DATA(data);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      dense_matrix->values[i+j*dense_matrix->ld] =
                  values[i*stride_rows+j*stride_cols];
    }
  }
} else if ( strcmp(dtype_str, "complex128") == 0 ) {
  double complex *values = PyArray_DATA(data);
  for (i = 0; i < rows; i++) {
```

```
    for (j = 0; j < cols; j++) {
      dense_matrix->values_cpx[i+j*dense_matrix->ld] =
                    values[i*stride_rows+j*stride_cols];
    }
  }
}
free(dtype_str);
```

The vector convert function is implemented as the dense matrix convert function restricted to a one dimensional NumPy nd-array.

4 Building a Python Matrix and Vector objects from C

After performing our computations in C we want to pass the corresponding results back to Python, or more precisely to the corresponding NumPy objects. Therefore we provide two functions similar to the ones from Section 3:

```
PyObject* matrix_to_python(mess_matrix c_matrix);
PyObject* vector_to_python(mess_vector c_vector);
```

Both functions take a C-M.E.S.S. object as an input and produce double precision real `float64` or complex `complex128` NumPy/SciPy object from it. If other data types are desired the objects must be converted afterwards in Python to the correct data type. Another difference is that both functions try to move the data directly to the Python object to reduce the memory usage and the copy operations. For this reason the input objects are reset at the end of the function and will be empty. If we need the data afterwards in C again we have to create a copy before.

4.1 Converting a matrix from C-M.E.S.S. to Python

Similar to the function presented in Subsection 3.1 we have to distinguish all possible input storage types of the `mess_matrix` object. This results in the following skeleton for the matrix convert function:

```
PyObject* matrix_to_python(mess_matrix c_matrix) {
  if ( MESS_IS_CSR(c_matrix) ) {
    ...
    return csr_matrix;
  } else if ( MESS_IS_CSC(c_matrix) ) {
    ...
    return csc_matrix;
  } else if ( MESS_IS_COORD(c_matrix) ) {
    ...
    return coo_matrix;
  } else if ( MESS_IS_DENSE(c_matrix) ) {
    ...
    return dense_matrix;
  } else {
```

```
    PyErr_SetString(PyExc_TypeError,
      "C-MESS Matrices must be CSR, CSC, COORD,  or DENSE ones");
    return NULL;
  }
};
```

For the further explanations we restrict again to CSR and dense matrices because the two remaining sparse storages formats again work analogously.

4.2 Handling of sparse matrices

Depending on the storage type of the input we convert the C-M.E.S.S. matrix to an instance of the following SciPy objects: `scipy.sparse.csr_matrix`, `scipy.sparse.csc_matrix` or `scipy.sparse.coo_matrix`. This works similar to the already presented direction to C in a three step scheme:

1. Create two Python arrays containing the index information of the sparse matrix. Due to compatibility issues we copy them to a NumPy `NPY_INT` integer array such that we fit all different platform specifications without an additional distinction of cases.

2. Move the value data directly to a Python array without coping or reallocating a memory location.

3. Call the constructor routine of the corresponding NumPy or SciPy object with the previously created arrays.

The Python arrays for the first step are created using the `PyArray_SimpleNew` function. This function allocates a new n-dimensional Python array of a desired size and data type. The C compatible memory location for the array is retrieved using the PyArray_DATA function afterwards. This location can be used as a standard C style array and is filled with the corresponding data using standard C operations. In the case of our two index vectors we allocate two arrays of the type `NPY_INT` and fill them with a `for`-loop afterwards casting the `mess_int_t` integers to the correct Python integer type `npy_int`:

```
npy_intp dim = (npy_intp) c_matrix->rows +1 ;
PyObject *ReturnIndptrArray = PyArray_SimpleNew(1, &dim, NPY_INT);
npy_int* array = (npy_int *) PyArray_DATA(ReturnIndptrArray);
for ( i = 0; i < dim; i++) {
  array[i] = (npy_int ) c_matrix->rowptr[i];
}

dim = (npy_intp) c_matrix->nnz;
PyObject *ReturnIndicesArray = PyArray_SimpleNew(1, &dim, NPY_INT);
array = (npy_int *) PyArray_DATA(ReturnIndicesArray);
for (i = 0; i < dime; i++) {
  array[i] = (npy_int) c_matrix->colptr[i];
}
```

In order to save some memory we do not want to copy the values of the matrix too. In this case we directly use the memory location where the data reside in C for the Python matrix too. Therefore we create the Python array containing the values using the `PyArray_SimpleNewFromData`. This function works similar to the `PyArray_SimpleNew` function, but instead of allocating a new memory location it take an already allocated one as a pointer and refers to that one. This results in the following piece of code to transfer the values of the matrix to Python:

```
PyObject *ReturnDataArray = NULL;
if(c_matrix->data_type == MESS_REAL){
  npy_intp nnz = (npy_intp) c_matrix->nnz;
  ReturnDataArray = PyArray_SimpleNewFromData(1, &nnz,
                    NPY_FLOAT64, (void *) c_matrix->values);
} else if(c_matrix->data_type == MESS_COMPLEX) {
  npy_intp nnz = (npy_intp) c_matrix->nnz;
  ReturnDataArray = PyArray_SimpleNewFromData(1, &nnz,
                    NPY_COMPLEX128, (void *) c_matrix->values_cpx);
} else {
  PyErr_SetString(PyExc_TypeError,
    "C-MESS Matrices can only be Real or Complex");
}
```

After all data has been moved to Python we call the constructor method of the `scipy.sparse.csr_matrix` type. For our case we use the following variant:

```
csr_matrix = scipy.sparse.csr_matrix((data, indices, indptr),
                                     shape=(M, N))
```

To this end we have to package all three arrays that contain the data for a CSR matrix to one tuple of three elements and the dimension of the matrix to a second tuple of two integers:

```
PyObject* ArrayReturn = Py_BuildValue("(OOO)", ReturnDataArray,
              ReturnIndicesArray, ReturnIndptrArray);
PyObject* Dim_Return = Py_BuildValue("(ii)", c_matrix->rows,
              c_matrix->cols);
```

These tuples are now passed to the constructor which sets up the SciPy instance of our sparse matrix:

```
PyObject *ScipySparse = PyImport_ImportModule((char*)"scipy.sparse");
PyObject *Create_CSRmatrix = PyObject_GetAttrString(ScipySparse,
                             (char*)"csr_matrix");
PyObject *csr_matrix =  PyObject_CallObject(Create_CSRmatrix,
                    Py_BuildValue("(OO)", ArrayReturn, Dim_Return));
```

As final step we have to decrement the reference count of all newly generated Python objects by one and reset the `c_matrix` because we move the data to Python and the former `values` memory location is now controlled by the Python memory management. The last step is done setting `c_matrix->values` or respectively `c_matrix->values_cpx`

to NULL and calling MESS_MATRIX_RESET on the matrix. This does, however, not mean that the matrix must not be cleared at the end of the Python-C extension via mess_matrix_clear.

In case of CSC and Coordinate matrices we have to change the dimensions of the different arrays slightly and adjust the constructor calling sequence.

4.3 Moving dense matrices to Python

Since C-M.E.S.S. supports only the Fortran compatible matrix storage and NumPy uses the C-style storage by default we either have to transpose our matrix before or transfer it using a similar technique as presented in Subsection 3.1.2. We choose the second way because of it does not need the intermediate transposed copy of the matrix and it allows to transfer matrices where the leading dimension is not equal to the number of rows. This is for example the case if the given matrix is only a view to another matrix (similar to the concept of slices in NumPy). Additionally, implementing the transfer in the same way as the transfer from Python to C works allows to handle a change in the default storage scheme behavior of NumPy without changing our code. The work flow in this case is as follows:

1. Allocate the 2 dimensional NumPy array for the matrix values.
2. Get the row and column stride from the previously allocated NumPy array.
3. Copy the matrix values from C to Python. The corresponding position in the Python array is computed from the stride properties using 1 from Page 12.
4. Call the constructor numpy.matrix to create a dense matrix out the more general two dimensional object.

The two dimensional array is allocated using the PyArray_SimpleNew function as for the sparse matrices and the stride is extracted using PyArray_STRIDE which was already mentioned in Subsection 3.1.2:

```
PyObject* ReturnArray = NULL;
npy_intp  dim[2] = {c_matrix->rows, c_matrix->cols};
npy_intp stride_rows, stride_cols;
if ( MESS_IS_REAL(c_matrix)) {
  ReturnArray = PyArray_SimpleNew(2, dim , NPY_FLOAT64);
} else if (MESS_IS_COMPLEX(c_matrix)) {
  eturnArray = PyArray_SimpleNew(2, dim , NPY_COMPLEX128);
stride_rows = PyArray_STRIDE(ReturnArray,0);
stride_cols = PyArray_STRIDE(ReturnArray,1);
```

As before, depending on the data type we have to adjust the stride variables because they are counted as byte offsets and not in terms of floating point numbers. Afterwards we iterate over all matrix elements and copy them to the corresponding position in the ReturnArray:

```
if(MESS_IS_REAL(c_matrix)){
  double *values = PyArray_DATA(ReturnArray);
  stride_cols /= sizeof(double);
  stride_rows /= sizeof(double);
  for (i = 0; i < c_matrix->rows; i++)
    for (j = 0; j < c_matrix->cols; j++)
      values[i*stride_rows+j*stride_cols] = c_matrix->values[i+j*
          c_matrix->ld];
} else if(MESS_IS_COMPLEX(c_matrix)){
  double complex *values = PyArray_DATA(ReturnArray);
  stride_cols /= sizeof(double complex );
  stride_rows /= sizeof(double complex );
  for (i = 0; i < c_matrix->rows; i++) {
    for (j = 0; j < c_matrix->cols; j++) {
      values[i*stride_rows+j*stride_cols] =
        c_matrix->values_cpx[i+j*c_matrix->ld];
}
```

Although, we already created a two dimensional NumPy array we call the `numpy.matrix` constructor as well. This is more or less done for compatibility reasons and easier handling of the matrix in scientific computing. As difference between a two dimensional array and a matrix the NumPy documentation says "A matrix is a specialized 2-D array that retains its 2-D nature through operations."[3] The call of the constructor is done like in the case of the sparse matrices: dense matrix from Python to C:

```
PyObject *Numpy = PyImport_ImportModule((char*)"numpy");
PyObject *Create_dense_matrix =
            PyObject_GetAttrString(Numpy,(char*)"matrix");
PyObject *ReturnMatrix = PyObject_CallObject(Create_dense_matrix,
            Py_BuildValue("(O)", ReturnArray));
//clean up
Py_DECREF(Numpy);
Py_DECREF(Create_dense_matrix);
Py_DECREF(ReturnArray);
MESS_MATRIX_RESET(c_matrix);
return ReturnMatrix;
```

4.4 Converting vectors to Python

Like in the previous Section the vectors are regarded as special matrices. But from the data structure point of view this is only a continuous array of values from the same type. This allows us an easy implementation of the convert routine. We only need to call PyArray_SimpleNewFromData with the dimension of the vector and the right pointer to the entries:

```
PyObject* vector_to_python(mess_vector c_vector){
```

```
  PyObject* ReturnArray = NULL;
  npy_intp dim;
  if(MESS_IS_REAL(c_vector)) {
    dim = (npy_intp ) c_vector -> dim;
    ReturnArray = PyArray_SimpleNewFromData(1, &dim, NPY_FLOAT64,
                  c_vector->values);
    c_vector->values = NULL;
    c_vector->dim = 0;
  } else if (MESS_IS_COMPLEX(c_vector)) {
    dim = (npy_intp ) c_vector -> dim;
    ReturnArray = PyArray_SimpleNewFromData(1, &dim, NPY_COMPLEX128,
                  c_vector->values_cpx);
    c_vector->values_cpx = NULL;
    c_vector->dim = 0;
  }
  return ReturnArray;
}
```

5 Interfacing the LRCF-ADI algorithm

After we discussed the data transfer of the two main objects in the numerical linear algebra we focus on an easy to use and easy to extend interface to various large scale matrix equation solvers from C-M.E.S.S.. As most important representative of this family we use the continuous time Lyapunov equation and its generalization

$$AX + XA^T + BB^T = 0, \tag{2}$$

$$AXE^T + EXA^T + BB^T = 0. \tag{3}$$

For sake of completeness we mention the transposed case of these two equations

$$A^TX + XA + C^TC = 0, \tag{4}$$

$$A^TXE + E^TXA + C^TC = 0 \tag{5}$$

as well. These Lyapunov equations play an important role in system and control theory. They are the key ingredients for the analysis of linear time invariant systems [8]. Furthermore the solution is necessary for the Balanced Truncation(BT) model order reduction [17, 18] or the solution of algebraic Riccati equations using a Newton-Scheme [18].

In the small and dense case there exist the Bartels-Stewart Algorithm [9] and Hammarlings-Method [15] to solve these equations. In the sparse and large scale case the Alternating-Directions-Implicit (ADI) algorithm was shown to be one of most efficient solvers. Caused by the problem of storing the full solution $X \in \mathbb{R}^{n\times n}$ of a large scale Lyapunov equation the Low-Rank-Cholesky-Factor ADI (LRCF-ADI) algorithm restricts to a low rank solution $Z \in \mathbb{C}^{n\times p}$, $X \approx ZZ^H$ with $p \ll n$. Details about the current state of the art implementations in C-M.E.S.S. can be found in [18, 10, 11, 12, 14] and will not be discussed here.

5.1 Transition of the options class from Python to C

The ADI algorithm is controlled by a parameter set which defines for example the maximum iteration number, stopping criteria, and information about the shift parameter computation. These parameters are combined as a `class` in Python which corresponds to a structure in C. For compatibility reasons the organization of the Python class is similar to the options structure in MATLAB-M.E.S.S.. Table 1 shows the transition between the components in the Python class and the `mess_lrcfadi_options` structure in C. For further extensions like the Low-Rank-Netwon-Method (LRNM) we include the necessary information already in the options class. Additional information that exists in the C-M.E.S.S. `mess_lrcfadi_options` structure are not affected by the transition.

Because the entries in the options class are only simple values or already handled data types we can transfer them to C easily. In addition to the `PY_GET_LONG` macro, presented in Section 3 we define two macros to transfer floating point numbers and vectors to C:

```
#define PY_GET_DOUBLE(dest, from, name ) { \
 PyObject *pv = PyObject_GetAttrString(from, name); \
 if (pv != NULL) { \
  dest = PyFloat_AsDouble(pv); \
  Py_DECREF(pv); } \
 }
#define PY_GET_VECTOR(dest,from,name) { \
 PyObject *pv = PyObject_GetAttrString(from,name); \
 if (pv !=NULL && pv != Py_None) { \
   dest = vector_to_c(pv); \
   Py_DECREF(pv); \
 } else { Py_XDECREF(pv); dest = NULL; }}
```

The whole transition procedure will work as follows: First we extract the `adi`, the `adi.shifts` and the `nm` sub-class out the options class and then copy each value from Python to the corresponding one in C using the previously presented macros. Only the `opt.adi.type` parameter needs a separate handling. This parameter is character in MATLAB and Python but an enumeration in C. The underlying string is moved to C via the already presented `ConvStringtoC` macro. Afterwards a comparison selects the equivalent entry from the enumeration. The following source code is a sketch how the transition procedure is implemented:

```
mess_lrcfadi_options parse_lrcfadi_options(PyObject *options) {
 mess_lrcfadi_options opt = NULL ;
 PyObject *adi_opt = NULL , *adi_shift_opt =NULL, *nm_opt = NULL;
 mess_lrcfadi_options_init(&opt));

 adi_opt         = PyObject_GetAttrString(options,"adi");
 adi_shift_opt   = PyObject_GetAttrString(adi_opt, "shifts");
 nm_opt          = PyObject_GetAttrString(options,"nm");
```

```
 // Extract values form .adi.
 PY_GET_LONG(opt->maxit, adi_opt, "maxit");
 PY_GET_DOUBLE(opt->res2_tol,adi_opt,"res2_tol");
 ...
 // Extract values from .adi.shitfs.
 PY_GET_VECTOR(opt->p, adi_shift_opt, "p");
 PY_GET_LONG(opt->arp_p, adi_shift_opt, "arp_p");
 ...
 // Extract values from .nm.
 PY_GET_LONG(opt->nm_maxit, nm_opt, "maxit");
 ...

 Py_DECREF(adi_opt);
 Py_DECREF(adi_shift_opt);
 Py_DECREF(nm_opt);
 return opt;
}
```

5.2 Wrapper around `mess_lrcfadi_lradi`

Finally we have to provide a wrapper function around the `mess_lrcfadi_adi` function. This wrapper is included in the `interface_python_methods` structure from Section 2 to register our extension in the Python name space. This function implements the LRCF-ADI algorithm for the standard and the generalized Lyapunov equation with many C-specific implementation improvements as well as threading capabilities [18, 11, 16]. The wrapper has to perform the following steps:

1. Parse the calling sequence and identify the arguments.
2. Unpack the `equation` object and extract the matrices A, B and if existing also E using the conversion functions from Section 3.
3. Transcribe the `options` class from Python to C using the parse_lrcfadi_options from Subsection 5.1.
4. Setup the mess_lrcadi_eqn object from the given matrices.
5. Call `mess_lrcfadi_adi` to solve one of the Equations (2), (3), (4), or (5).
6. Convert the solution matrix Z and the 2-norm residual history to Python objects and build a return value from them.

The equation structure contains the system matrix A as `eqn.A_`, the mass matrix E as `eqn.E_` and the right hand side B as `eqn.B`. If the mass matrix does not exist the standard Lyapunov equation is solved otherwise the generalized one is solved. The additional underscores exist only to get an equivalent structure as in the MATLAB-M.E.S.S.. Depending on the `opt.adi.type` the right hand side must have the correct shape. If the type is set to `'B'` the right hand side `B` must fulfill $BB^T \in \mathbb{R}^{n\times n}$. If the

Component in Python	Component in C	Description
opt.adi.maxit	opt->maxit	Maximum iteration number in the ADI iteration
opt.adi.type	opt->type	Choose between Equation (2)/(3) and (4)/(5)
opt.adi.res2_tol	opt->res2_tol	2-norm cancellation criterion in the ADI
opt.adi.res2c_tol	opt->res2c_tol	2-norm stagnation criterion in the ADI
opt.adi.rel_change_tol	opt->rel_change_tol	Relative Frobenius change in the solution factor criterion
opt.adi.ccStep	opt->ccStep	Frequency of the column compression
opt.adi.ccTol	opt->ccTol	Tolerance for the column compression
opt.adi.gpStep	opt->gpStep	Frequency for the Galerkin projection in the ADI
opt.adi.output	opt->output	Verbosity level in the ADI
opt.adi.shifts.p	opt->p	Predefine shift vector
opt.adi.shifts.arp_p	opt->arp_p	Number of Arnoldi step w.r.t. A in the shift computation
opt.adi.shifts.arp_m	opt->arp_m	Number of Arnoldi step w.r.t. A^{-1} in the shift computation
opt.adi.shifts.l0	opt->l0	Number of desired shifts
opt.adi.shifts.paratype	opt->paratype	Type of the shifts
opt.adi.shifts.b0	opt->b0	Starting vector for the Arnoldi process
opt.nm.maxit	opt->nm_maxit	Maximum number of iterations in the LRNM
opt.nm.res2_tol	opt->nm_res2_tol	2-norm cancellation criterion in the LRNM
opt.nm.res2c_tol	opt->nm_res2c_tol	2-norm stagnation criterion in the LRNM
opt.nm.rel_change_tol	opt->nm_rel_change_tol	Frobenius norm relative change in the LRNM
opt.nm.rel2_change_tol	opt->nm_rel2_change_tol	2-norm relative change in the LRNM
opt.nm.gpStep	opt->nm_gpStep	Frequency for the Galerkin projection in the LRNM
opt.nm.singleshifts	opt->nm_singleshifts	Use the same ADI shifts for the whole LRNM
opt.nm.output	opt->nm_output	Verbosity level in the LRNM

Table 1: Transition between Python and C

type is 'C' the right hand side B has to fulfill $B^T B \in \mathbb{R}^{n \times n}$. This corresponds to the two different Lyapunov equation types (2) and (4).

The first step in the implementation of the wrapper is the identification and the extraction of the function arguments. Depending on the flags set in the PyMethodDef structure the arguments are identified by their order or keywords. The by-order behavior is know from C, where the position of the parameter in the calling sequence determines its meaning. If this behavior is specified in the PyModuleDef structure we extract the parameters using

```
int PyArg_ParseTuple(PyObject *args, const char *format, ...)
```

where the format string identifies the type of arguments. The destination variables of the arguments are passed as pointers in the variable argument list in the order they occur in the format string. More details about the format string and the corresponding arguments are available in [4]. A more flexible variant to pass arguments to a function is to identify them using keywords. The function call in Python looks like:

```
Z = Py_mess_lrcfadi(equation=myequation, options=myoptions)
```

This type of a argument lists is handled by `PyArg_ParseTupleAndKeywords` which gets an additional keyword array to match the allowed keywords. We use the keyword argument for our extension because it supports both by-order and by-keyword identification of the arguments. As counterpart to the `Py_Parse*` functions we can use the `Py_BuildValue` function to combine different data types and data structures to one return value. The behavior of this function is similar to the parser function. It takes a format string which contains the order of the elements in the return value and the variable argument list. More details about this are also available in [4].

Employing all previously discussed functions and macros we end up with the following source code for our LRCF-ADI wrapper:

```
PyObject* Pymess_lradi(PyObject *self, PyObject *args, PyObject *
    kwrds){
 PyObject *equation = NULL, *options = NULL, *temp = NULL;
 mess_matrix  A=NULL,B=NULL,E=NULL,Z=NULL;
 mess_lrcfadi_options opt = NULL ;
 mess_lrcfadi_eqn lyapeqn = NULL ;
 mess_lrcfadi_status stat = NULL ;

 static char *kwlist[] = {"equation","options",NULL};
 if (!PyArg_ParseTupleAndKeywords(args, kwrds, "OO", kwlist,
      &equation, &options)){
   PyErr_SetString(PyExc_TypeError, "The call sequence is wrong");
   return NULL; }

 opt = parse_lrcfadi_options(options);
 mess_lrcfadi_status_init(&stat);
 PY_GET_MATRIX(A,equation,"A_");
 PY_GET_MATRIX(B,equation,"B");
 PY_GET_MATRIX(E,equation,"E_");
```

```
 if ( A==NULL || B == NULL ) {
  PyErr_SetString(PyExc_RuntimeError, "Matrix A or B is missing");
  return NULL;
 }
 mess_lrcfadi_eqn_init(&lyapeqn);
 if (E == NULL) {
  mess_lrcfadi_eqn_lyap(lyapeqn, opt, A, B);
 } else {
  mess_lrcfadi_eqn_glyap(lyapeqn, opt, A, E, B);
 }

 mess_lrcfadi_parameter(lyapeqn, opt, stat);
 mess_matrix_init(&Z);
 mess_lrcfadi_adi(lyapeqn,opt,stat,Z);
 temp = Py_BuildValue("OO",matrix_to_python(Z),
         vector_to_python(stat->res2_norms));

 mess_lrcfadi_eqn_clear(&lyapeqn);
 mess_lrcfadi_options_clear(&opt);
 mess_lrcfadi_status_clear(&stat);
 if (A!=NULL) mess_matrix_clear(&A);
 if (B!=NULL) mess_matrix_clear(&B);
 if (Z!=NULL) mess_matrix_clear(&Z);
 if (E!=NULL) mess_matrix_clear(&E);
 return temp;
}
```

The PY_GET_MATRIX macro shown above works exactly like the PY_GET_VECTOR macro but with matrices instead of vectors. The real implementation includes additional sanity checks and error handling for each critical step. Other wrappers like for the Low-Rank-Newton-Method can be implemented easily in the same way.

6 Examples

We have seen how to create a Python interface for one of the C-M.E.S.S. functions. We now want to give a small example how this interacts with NumPy and SciPy.

6.1 Solving a standard Lyapunov equation

As first example we want to solve a standard Lyapunov equation

$$AX + XA^T + BB^T = 0$$

defined by two matrices A and B. The matrices are read form MatrixMarket files and stored in an instance of `pycmess.equation`. The options for the ADI are left to their default values given by `pycmess.options()`. After setting up the solver our

program should call the solver and verify the result again. Beside our C extension the program needs to import NumPy and some packages from SciPy. The whole concise Python program for this purpose looks like:

```
from numpy import *
from scipy.sparse import *
from scipy.linalg import norm
from scipy.io import mmread
from pycmess import *
import sys

if len(sys.argv) != 3:
    print("Usage: ",sys.argv[0] , " A.mtx B.mtx")
    raise SyntaxError

opt = options()
eqn = equation()
eqn.A_= mmread(sys.argv[1]).tocsr()
eqn.B = mmread(sys.argv[2])
Z, _ =  lradi(eqn,opt)
X = Z.dot(Z.T)
RES= eqn.A_.todense().dot(X)
RES= RES + RES.T + eqn.B.dot(eqn.B.T)
res = norm(RES,1)
relres = res / norm(eqn.B.dot(eqn.B.T),1)
print "Rel. Residual: ", relres
```

We can see that the main part for solving the equation consists of only 5 lines of code. Two of them are the initialization of the data structure, additional two lines setup the equation and finally the equation is solved with one call of the solver routine. The additional return values "_" is necessary because the ADI-wrapper returns two values and we are only interested in the first one which is the solution of our equation. The second one which contains a vector with the convergence history is passed to the dummy variable "_". It is not possible either in Python-2 or in Python-3 to declare a return value as optional. Due to a slow sparse-dense matrix-matrix product in SciPy we have to convert the system matrix A to a dense one if we want to compute the residual. Otherwise computing AX takes a long time.

6.2 Solving a generalized Lyapunov equation

For this example we want to solve the transposed generalized Lyapunov equation

$$A^T X E + E^T X A + C^T C = 0$$

with enabled Galerkin projection [18] and residual tolerances adjusted. All necessary information can be set in the `opt` object generated by `pycmess.options()`. The transposed equation is selected by setting `opt.adi.type` to `'C'`. The Galerkin projection for the Lyapunov equation is enabled by setting `opt.adi.gpStep` positive

integer. The integer defines after how many ADI steps the projection is performed. The residual tolerances are adjusted by various components in the options structure as they are listed in Table 1. We assume that we want to have a 2-norm residual of 10^{-12}. As in the previous example the matrices are read again from MatrixMarket files. The overall program looks like:

```
from numpy import *
from scipy.sparse import *
from scipy.linalg import norm
from scipy.io import mmread
from pycmess import *
import sys

if len(sys.argv) != 4:
    print("Usage: ",sys.argv[0] , " A.mtx E.mtx B.mtx")
    raise SyntaxError

opt = options()
eqn = equation()
eqn.A_= mmread(sys.argv[1]).tocsr()
eqn.E_= mmread(sys.argv[2]).tocsr()
eqn.B = mmread(sys.argv[3]).todense()
opt.adi.gpStep=5
opt.adi.res2_tol = 1e-12
opt.adi.type='C'
Z, _ =  lradi(eqn,opt)
X = Z.dot(Z.T)
RES= eqn.A_.todense().T.dot(X).dot(eqn.E_.todense())
RES= RES + RES.T + eqn.B.T.dot(eqn.B)
res = norm(RES,1)
relres = res / norm(eqn.B.T.dot(eqn.B),1)
print "Rel. Residual: ", relres
```

We can easily see that we only have to include the `E_` matrix in the equation structure and modify three components of the options structure so satisfy our new problem setup.

7 Conclusions

In the previous sections we develop a set of helper routines and a unified Python-2 and Python-3 interface for the C-M.E.S.S. library. Upon those helper functions we are able to write concise wrapper functions for the algorithms provided by C-M.E.S.S. which interact perfectly with the NumPy/SciPy software packages. The comparison of the wrapper approach and the plain Python NumPy/SciPy implementation approach is the topic of the Bachelor studies of Björn Baran.

Acknowledgements

The third author acknowledges the financial support for his two-month summer-internship received from the MPI Magdeburg inside the DAAD IAESTE Program.

References

[1] *ctypes A foreign function library for Python*, `http://docs.python.org/3/library/ctypes.html`.

[2] *Extending and Embedding the Python Interpreter*, `http://docs.python.org/3/extending/`.

[3] *NumPy*, `http://www.numpy.org`.

[4] *Parsing arguments and building values in Python 3*, `http://docs.python.org/3/c-api/arg.html`.

[5] *Porting Extension Modules to Python 3*, `http://docs.python.org/3/howto/cporting.html`.

[6] *SciPy*, `http://www.scipy.org/`.

[7] *Whats New in Python*, `http://docs.python.org/3/whatsnew/`.

[8] A. C. Antoulas, *Approximation of Large-Scale Dynamical Systems*, SIAM Publications, Philadelphia, PA, 2005.

[9] R. H. Bartels and G. W. Stewart, *Solution of the matrix equation $AX + XB = C$*, Communications of the ACM, 15 (1972), pp. 820–826.

[10] P. Benner, P. Kürschner, and J. Saak, *Avoiding complex arithmetic in the low-rank adi method efficiently*, vol. 12, WILEY-VCH Verlag, 2012, pp. 639–640. DOI: 10.1002/pamm.201210308.

[11] P. Benner, P. Kürschner, and J. Saak, *Efficient Handling of Complex Shift Parameters in the Low-Rank Cholesky Factor ADI Method*, Numerical Algorithms, 62 (2013), pp. 225–251. 10.1007/s11075-012-9569-7.

[12] P. Benner, P. Kürschner, and J. Saak, *Self-generating and efficient shift parameters in ADI methods for large Lyapunov and Sylvester equations*, Preprint MPIMD/13-18, Max Planck Institute Magdeburg, October 2013. Available from `http://www.mpi-magdeburg.mpg.de/preprints/`.

[13] P. Benner and J. Saak, *Efficient solution of large scale Lyapunov and Riccati equations arising in model order reduction problems*, Proc. Appl. Math. Mech., 8 (2008), pp. 10085 – 10088.

[14] ——, *Numerical solution of large and sparse continuous time algebraic matrix Riccati and Lyapunov equations: a state of the art survey*, GAMM Mitteilungen, 36 (2013), pp. 32–52.

[15] S. Hammarling, *Numerical solution of the stable, non-negative definite Lyapunov equation*, IMA J. Numer. Anal., 2 (1982), pp. 303–323.

[16] M. Köhler and J. Saak, *Efficiency improving implementation techniques for large scale matrix equation solvers*, Chemnitz Scientific Computing Prep. 09-10, TU Chemnitz, 2009.

[17] B. C. Moore, *Principal component analysis in linear systems: controllability, observability, and model reduction*, IEEE Trans. Automat. Control, AC-26 (1981), pp. 17–32.

[18] J. Saak, *Efficient Numerical Solution of Large Scale Algebraic Matrix Equations in PDE Control and Model Order Reduction*, PhD thesis, TU Chemnitz, July 2009. Available from `http://nbn-resolving.de/urn:nbn:de:bsz:ch1-200901642`.

[19] M. von Löwis, *Python Enhancement Proposal 3121: Extension Module Initialization and Finalization*, tech. rep., `http://www.python.org/dev/peps/pep-3121/`, 2007.