SMG2S Manual

For SMG2S Release 1.1

Version 1.1

Xinzhe Wu

Maison de la Simulation, Gif-sur-Yvette, France

May 9, 2019

Abstract

Iterative linear algebra methods are the important parts of the overall computing time of applications in various fields since decades. Recent research related to social networking, big data, machine learning and artificial intelligence has increased the necessity for non-hermitian solvers associated with much larger sparse matrices and graphs. The analysis of the iterative method behaviors for such problems is complex, and it is necessary to evaluate their convergence to solve extremely large non-Hermitian eigenvalue and linear problems on parallel and/or distributed machines. This convergence depends on the properties of spectra. Then, it is necessary to generate large matrices with known spectra to benchmark the methods. These matrices should be non-Hermitian and non-trivial, with very high dimension. A scalable parallel matrix generator SMG2S that uses the user-defined spectrum to construct large-scale sparse matrices and ensures their eigenvalues as the given ones with high accuracy is implemented based on MPI and C++11. This report gives the manual of SMG2S.

Contents

Chapter 1

Introduction

1.1 Getting Started

SMG2S (Scalable Matrix Generator with Given Spectrum) [1, 2] is a software which provides for generating the non-Hermitian Matrices with user-customized eigenvalues. SMG2S is implemented in parallel based on MPI (Message Passing Interface) and C++11 to support efficiently the generation of test matrices in parallel on distributed memory platforms.

Iterative linear algebra methods are essential for the applications in various fields. The analysis of the iterative method behaviors is complex, and it is necessary to evaluate their convergence to solve extremely large non-Hermitian eigenvalue and linear problems on parallel and/or distributed machines. This convergence depends on the properties of spectra. Thus, we propose SMG2S to generate large matrices with known spectra to benchmark these methods. The generated matrices are non-Hermitian and non-trivial, with very high dimension.

The functionality proposed inside SMG2S can verify the ability of SMG2S to keep the accuracy of a given spectrum. This function is based on the shift inverse power method. SMG2S also gives a graphic user interface to compare the given and final spectral distribution for the verification.

We will describe the following subset of the SMG2S.

- **Parallel Vector and Matrix:** this part presents the functions implemented in SMG2S to establish parallel vector and matrix over distributed memory platforms.
- **Nilpotent Matrix Object:** this part presents a special nilpotent matrix object for the matrix generation procedure in SMG2S.
- **Generating Matrix with prescribed eigenvalues:** this part gives the way to use SMG2S to generate required test matrices.
- **Interface to Other Languages/Libraries:** this part introduces the interface of SMG2S to other languages and existing scientific computational libraries such as PETSc and Trilinos.
- **Verification of Eigenvalues of Generated Matrix:** this part gives the way to verify the accuracy of eigenvalues of generated matrices comparing

with given spectrum. A graphic user interface is also provided to facilitate the comparison.

1.2 Installation

To obtain SMG2S, please follow the instructions at the SMG2S download page: https://smg2s.github.io/download.html.

Prerequisites:

- C++ Compiler with **c++11** support;
- Cmake (version minimum 3.6);
- (Optional) PETSc and SLPEc are necessary for the verification of the ability to keep the given spectrum.

In the main directory:

```
cmake . -DCMAKE_INSTALL_PREFIX=${INSTALL_DIRECTORY}
```

The main.cpp will generate an exectutable smg2s.exe to demonstrate a minimum sample :

```
make
```

The main part of SMG2S is a collection of header files. Install the header files into ${INSTALL_DIRECTORY}

```
make install
```

For testing the software in your platforms:

```
make test
```

The output of the test should be like:

```
Running tests...
Test project /User/name/SMG2S
Start 1: Test_Size_10000_w_proc1
1/4 Test #1: Test_Size_10000_w_proc1 .. Passed 1.20 sec
Start 2: Test_Size_20000_w_proc2
2/4 Test #2: Test_Size_20000_w_proc2 .. Passed 1.22 sec
Start 3: Test_Size_10000_s_proc1
3/4 Test #3: Test_Size_10000_s_proc1 .. Passed 1.20 sec
Start 4: Test_Size_10000_s_proc2
4/4 Test #4: Test_Size_10000_s_proc2 .. Passed 0.66 sec

100% tests passed, 0 tests failed out of 4

Total Test time (real) =   4.29 sec
```

1.3 CMake Options

We use CMake to build, test and package SMG2S. If you do not have PETSc and SLEPc in your platform, please make sure the option below is OFF in CMakeLists.txt.

```
option(INSTALL_TO_USE "Install SMG2S include files?" OFF)
```

1.4 Copyright and Licensing of SMG2S

SMG2S is an open source software published under the GNU Lesser General Public License v3.0. SMG2S can be redistributed and modified under the terms of this license.

SMG2S is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. SMG2S is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with SMG2S. If not, see ¡http://www.gnu.org/licenses/¿.

1.5 Programming Language in SMG2S

SMG2S is a collection of templated header files written in C++. The wrappers to C and Python codes are provided. The users of PETSc or Trilinos can directly use SMG2S with the interfaces implemented inside.

1.6 Referencing SMG2S

Place cite these papers if you want to reference SMG2S.

- @article{galichergenerate, title={Generate Very Large Sparse Matrices Starting from a Given Spectrum}, author={Galicher, Hervé and Boillod-Cerneux, France and Petiton, Serge and Calvin, Christophe} }
- @inproceedings{wu2018parallel, title={A Parallel Generator of Non-Hermitian Matrices computed from Given Spectra}, author={Wu, Xinzhe and Petiton, Serge and Lu, Yutong}, booktitle={VECPAR 2018: 13th International Meeting on High Performance Computing for Computational Science}, year={2018}}

1.7 Directory Structure

The directory structure of SMG2S is given as follows:

```
SMG2S
└─parVector
```

```
│
│   ├── parVectorMap.h ◇implementation of distributed vector map
│   └── parVector.h ◇implementation of distributed vector
├── parMatrix
│   ├── MatrixCSR.h ◇implementation of serial CSR Matrix
│   └── parMatrixSparse.h ◇distributed sparse matrix
├── smg2s
│   ├── specGen.h ◇Function to provide given spectrum
│   └── smg2s.h ◇ implementation of smg2s generator
├── utlis
│   ├── MPI_DataType.h
│   ├── utlis.h
│   └── logo.h
├── verification
│   ├── powerIverse.cpp ◇verfication impl based on SLEPc
│   ├── UI ◇GUI for comparison based on SLEPc
│   └── tests
├── interface
│   ├── C ◇interface to C
│   ├── PETSc ◇interface to PETSc
│   ├── Python ◇interface to Python
│   └── Trilinos ◇interface to Trilinos
├── example
│   ├── arnoldi
│   ├── gmres
│   ├── krylov
│   └── teptra
└── config
    └── config.h
```

1.8 List of SMG2S Contributors

This is the list of SMG2S contributors:

Xinzhe Wu	main constributor	xinzhe.wu@ed.univ-lille1.fr
Serge Petiton	Supervisor	serge.petiton@univ-lille1.fr
Quentin Petit	GUI Implementation (Intern)	quentin.petit@polyetch-lille.net

Chapter 2

Templated SMG2S Parallel Matrix and Vector

2.1 Parallel Vector

The distributed vector inside SMG2S is implemented using the C++ programming language and MPI. The parallel vector implementation is composed of two main classes: *parVectorMap* and *parVector*. *parVectorMap* class is a vector index map which controls the partitioning and distribution over the processes, and *parVector* is the parallel vector itself, which contains the actual distributed data and the corresponding functions controlling the data.

2.1.1 Vector Map

The distribution of a set of integer labels (or elements) across the processes is here called a *parVectorMap*. In SMG2S, it is implemented with the help of *std* :: *map*, which maps the proc number and related integers. Basically, *parVectorMap* handles the definition of global and local indices for the mapping across the processes. Here we give some methods of $parVectorMap < S >$:

```
S Loc2Glob(S local_index);
//convert local index to global index;

S Glob2Loc(S global_index);
//convert local index to global index;

int GetRank();
//Get proc index;

S GetLowerBound();
//Get lower bound index for each proc;

S GetUpperBound();
//Get upper bound index for each proc;

S GetLocalSize();
```

```
    //Get index number for each proc;

    S GetGlobalSize();
    //Get total interger number for all procs.
```

2.1.2 Creating a Distributed Vector

A parallel vector *parVector* across the processes can be created with the help of *parVectorMap*. The entries of *parVector* are distributed over different processes referring to the *parVectorMap*. *parVector* $< T, S >$ is implemented with the C++ template, with T the data type of entry, and S the data type of index.

This is the constructor of *parVector*, with *MPI_Comm ncomm* the working MPI communicator, *Slbound* and *Subound* respectively the lower and upper bound of vector global indices on each proc.

```
    /*constructor*/
    parVector(MPI_Comm ncomm, S lbound, S ubound);
```

Here we give some methods of *parVector*:

```
    parVectorMap<S> *GetVecMap();
    //return the related parVectorMap of parVector;

    S Loc2Glob(S local_index);
  //convert local index to global index;

    S Glob2Loc(S global_index);
    //convert local index to global index;

    int GetRank();
    //Get proc index;

    S GetLowerBound();
    //Get lower bound index for each proc;

    S GetUpperBound();
    //Get upper bound index for each proc;

    S GetLocalSize();
    //Get index number for each proc;

    S GetGlobalSize();
    //Get total interger number for all procs;

    T *GetArray();
    //Get the array containg the entries on each proc;

    void SetValueLocal(S row, T value);
  //insert value in the local index named row;
```

```
   void SetValuesLocal(S nindex, S *rows, T *values);
   //insert array in the local indices named rows;

   void SetValueGocal(S row, T value);
   //insert value in the gocal index named row;

   void SetValuesGocal(S nindex, S *rows, T *values);
   //insert array in the gocal indices named rows;

   void SetTovalue(T value);
  //Set the entries of parVector all to same given value;

  void VecAdd(parVector *v);
  //Add another vector v with same mapping;

  void VecScale(T scale);
  //Scaling the vector;

  T VecDot(parVector *v);
  //vector dot product operation;

  void ReadExtVec(std::string spectrum);
  //read vector from local file;

  void VecView();
  //display the vector;

  void specGen(std::string spectrum);
//Generate/loal a special vector containing given spectrum
```

Here we give an example to generate a *parVector* by SMG2S:

```
   int world_size;
   int world_rank;
   int span, lower_b, upper_b;
   MPI_Comm_size(comm, &world_size);
   MPI_Comm_rank(comm, &world_rank);
   span = int(ceil(double(probSize)/double(world_size)));

   if(world_rank == world_size - 1){
       lower_b = world_rank * span;
       upper_b = probSize - 1 + 1;
   }else{
       lower_b = world_rank * span;
       upper_b = (world_rank + 1) * span - 1 + 1;
   }

   parVector<T,S> *vec = new parVector<T,S>(      \
   comm,lower_b, upper_b);
```

```
    vec->SetTovalue(T val);
```

2.1.3 Parallel Matrix

The one-dimensional row-major parallel matrix *parMatrixSparse* in SMG2S is distributed with the same *parVectorMap* of a given *parVector*. On each process, the columns indices and the entries values are stored by a $std :: map < T, S >$ with S data type of indices and T data type of entries.

This is the constructor of *parMatrixSparse*:

```
/*constructor*/
parMatrixSparse(parVector<T,S> *Vec, parVector<T,S> *Vec);
```

Here we given some methods of the *parMatrixSparse* object:

```
parVectorMap<S> *GetYMap();
//return the matrix mapping of the columns;

MPI_Comm GetComm();
//Get current working MPI communicator;

std::map<S,T> *GetDynMatLoc();
//Get the map storing cols and entries on each proc;

void LOC_MatView();
//display the parallel matrix

void Loc_SetValueLocal( S row, S col, T value);
//insert value in the local index row;

void Loc_SetValuesLocal( S nindex, S *rows, S *cols, \
 T *values);
//insert array  in the local index rows;

void SetValueGocal(S row, T value);
//insert value in the gocal index named row;

void SetValuesGocal(S nindex, S *rows, T *values);
//insert array in the gocal indices named rows;

void Loc_SetValue(S row, S col, T value);
//Set the entriy (row, col) of parMatrix with value;

void Loc_SetDiagonal(parVector<T,S> *diag);
//set the diagonal of matrix to a given vector;

void Loc_MatScale(T scale);
//Scaling the matrix;
```

```
void Loc_MatAXPY(parMatrixSparse<T,S> *X, T scale);
//AXPY operation;

void Loc_MatAYPX(parMatrixSparse<T,S> *X, T scale);
//AYPX operation;

void Loc_ZeroEntries();
//Zeros all entries and keep the previous matrix pattern;

void MA(Nilpotency<S> nilp, parMatrixSparse<T,S> *prod);
//matrix multiple a special nilpotent matrix;

void AM(Nilpotency<S> nilp, parMatrixSparse<T,S> *prod);
//special nilpotent matrix multiple another matrix;
```

2.1.4 Creating a Distributed Matrix

The is an example of creating a distributed matrix with the mapping of parallel vector:

```
/*lower_b and upper_b of each proc is given*/

parVector<T,S> *vec = new parVector<T,S>(     \
comm,lower_b, upper_b);

parMatrixSparse<T,S> *A=new parMatrixSparse<T,S>(vec,vec);
```

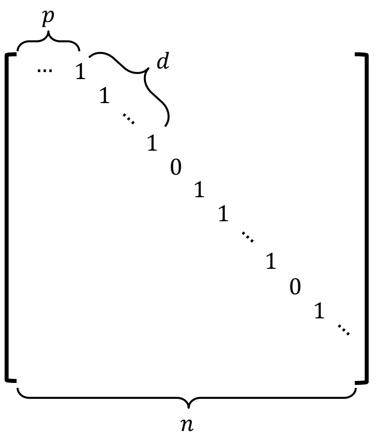
Chapter 3

Templated Nilpotent Matrix Object

3.1 Introduction

The nilpotent matrix is very important for the generation of test matrices with given spectrum. Il can be defined by several parameters, the explicit implementation is not necessary.

Figure 3.1: Nilpotent Matrix



The three parameters defined a nilpotent matrix is listed as:

- **dIagPostion**: the distance of the off-diagonal to the diagonal, refering to p in Fig. 3.1;
- **length**: the continuous one on the off-diagonal of nilpotent matrix, refering to d in Fig. 3.1;
- **probSize**: the number of row/column of nilpotent matrix, refering to n in Fig. 3.1.

3.2 Different Types of Nilpotent Matrix

The different nilpotent matrix will influence the sparsity pattern of the final generated matrix.

- NilpType1: diagPostion = 2
- NilpType2: diagPostion = 3
- NilpType3: diagPostion > 3

3.3 Creating a Nilpotent Matrix Object

```
Nilpotency<int> nilp;
nilp.NilpType1(length,probSize);
//
nilp.NilpType2(length,probSize);
//
nilp.NilpType3(diagPostion, length,probSize);
```

3.4 Parameter Validation for Nilpotent Matrix

- NilpType1: parameter length can be any integer value > 0;
- NilpType2: parameter length sould be even;
- NilpType3: validation of parameter length is complex. *length* should be divisible by p.

Chapter 4

Generating Matrix with SMG2S

4.1 SMG2S Class

The header file ./smg2s/smg2s.h implement the matrix generation method. It is defined as:

```
template<typename T, typename S>
parMatrixSparse<T,S> *smg2s(
S probSize,
Nilpotency<S> nilp,
S lbandwidth,
std::string spectrum,
MPI_Comm comm
)
```

Inside the definition, typename T is to define the size of matrix, and typename S is to define the scalar types of entries of matrix. We give the meaning of the input parameter as below:

- **S ProbSize**: the size of matrix to generate;
- **Nilpotency<S> nilp**: the nilpotent matirx object for generation;
- **S lbandwidth**: the bandwidth of lower-diagonal band of initial matrix;
- **std::string spectrum**: the file path of spectra file;
- **MPI_Comm comm**: the working MPI communicator.

4.2 Generation Workflow

1. Include the head file

```
#include <smg2s/smg2s.h>
```

2. Generate the Nilpotent Matrix Object:

```
Nilpotency<int> nilp;
nilp.NilpType1(length,probSize);
```

3. Create the parallel Sparse Matrix Object Mt:

```
parMatrixSparse<std::complex<double>,int> *Mt;
```

4. Generate a new matrix by SMG2S:

```
MPI_Comm comm; //working MPI Communicator
Mt = smg2s<std::complex<double>,int>(probSize, nilp,
lbandwidth, spectrum, comm);
```

Here, the **probsize** parameter represent the matrix size, **nilp** is the nilpotency matrix object that we have declared previously, **lbandwidth** is the bandwidth of lower-diagonal band. **spectrum** is the file path of spectra file, if **spectrum** is set as **" "**, SMG2S will use the mechanism inside to generate the spectral distribution. **comm** is the basic object used by MPI to determine which processes are involved in a communication.

The given spectra file is in **pseudo-Matrix Market Vector format**. For the complex eigenvalues, the given spectrum is stored in three columns, the first column is the coordinates, the second column is the real part of complex values, and the third column is the imaginary part of complex values.

```
%%MatrixMarket matrix coordinate complex general
3 3 3
1 10 6.5154
2 10.6288 3.4790
3 10.7621 5.0540
```

For the eigenvalues values, the given spectrum is stored in two columns, the first column is the coordinates, the second column is related values.

```
%%MatrixMarket matrix coordinate complex general
3 3
1 10
2 10.6288
3 10.7621
```

4.3 Creation of Given Spectrum

In the directory ./verification/tests, we give an example to generate the file of given spectrum, which can be reused by the users to create their own values. This is a small C++ file named as vecgen.cpp.

If the users want to generate the eigenvalues in time without loading from local file, they can customize their eigenvalues generation by the function specGen in the file ./verification/tests/specGen.h, and set the parameter spectrum of smg2s to be " ".

```
template<typename T, typename S>
void parVector<T,S>::specGen(std::string spectrum)
```

In this function, the eigenvalues are stored by the distributed vector text-colorblueparVector. And the filling of values on this parVector can be done by the method SetValueGlobal implemented in parVector, which takes the global indices to set values.

4.4 Customize the Low Band of Initial Matrix

We know that the low band bandwidth of initial matrix can be set by the parameter lbandwidth of smg2s. Additionaly, the distribution of entries of initial matrix can also be customized by the function matInit provided by the file ./verification/tests/specGen.h. En default, these entries are filled in random. The different mechanism to fill them will influence the sparsity of final generated sparse matrix.

```
template<typename T, typename S>
void matInit(
parMatrixSparse<T,S> *Am,
parMatrixSparse<T,S> *matAop,
S probSize,
S lbandwidth
)
```

In this function, distributed matrix *Am* and *matAop* should be filled with the same way. And these entries of matrix can be filled by the method Loc_SetValue implemented in parMatrixSparse. Loc_SetValue uses the global indices of matrix to set values.

Chapter 5

Interface to Other Languages/Libraries

Until now, SMG2S provides interfaces to C, Python, PETSc and Trilinos.

5.1 Interface to C

SMG2S install command will generate a shared library libsmg2s.so (libsmg2s2c.dylib on OS X platform) into ${INSTALL_DIRECTORY}/lib. It can be used to profit the C wrapper of SMG2S.

The way to use:

1. Add this shared library to LD_LIBRARY_PATH:

```
export LD_LIBRARY_PATH=${INSTALL_DIRECTORY}/lib
```

2. Include the header file:

```
#include <interface/C/c_wrapper.h>
```

3. create Nilpotency object :

```
struct NilpotencyInt *n;
n = newNilpotencyInt();
NilpType1(n, 2, 10);
```

4. After that, you need to create the parallel Sparse Matrix Object Mt like this :

```
struct parMatrixSparseRealDoubleInt *m;
m = newParMatrixSparseRealDoubleInt();
```

5. Generate by SMG2S :

```
smg2sRealDoubleInt(m, 10, n, 3 ," ",MPI_COMM_WORLD);
```

6. Release Nilpotency Object and parMatrixSparse Object :

```
ReleaseNilpotencyInt(&n);
ReleaseParMatrixSparseRealDoubleInt(&m);
```

SMG2S provides the C interface to different data types. For the data type of matrix size, it can be either *int* or *longint*; for the data type of matrix entries, it can be either *complex* or *real* with *single* or *double* precision.

The Nilpotent Matrix object is implemented for both *int* and *longint* as below:

```
struct NilpotencyInt;
struct NilpotencyLongInt;
```

The interface of C for parMatrixSparse Object and smg2s function can be defined as below, **SUFFIX** can be replaced by one of the selected data types:

- *ComplexDoubleLongInt*;
- *ComplexDoubleInt*;
- *ComplexSingleLongInt*;
- *ComplexSingleInt*;
- *RealDoubleLongInt*;
- *RealDoubleInt*;
- *RealSingleLongInt*;
- *RealSignleInt*.

```
//long int case

struct parMatrixSparseSUFFIX;
/*parVectorMap C wrapper*/

struct parVectorMapLongInt *newparVectorMapLongInt(void);

/*complex double long int*/

struct parMatrixSparseComplexSUFFIX *newPar\
MatrixSparseSUFFIX(void);

void ReleaseParMatrixSparseSUFFIX(struct \
parMatrixSparseSUFFIX **ppInstance);

void LOC_MatViewSUFFIX(struct parMatrix\
SparseSUFFIX *m);

void GetLocalSizeSUFFIX(struct parMatrix\
SparseSUFFIX *m, __int64_t *rs, __int64_t *cs);

void Loc_ConvertToCSRSUFFIX(struct parMatrix\
SparseComplexDoubleLongInt *m);

void Loc_CSRGetRowsArraySizesSUFFIX(struct parMatri\
xSparseSUFFIX *m, __int64_t *size,__int64_t *size2);

void Loc_CSRGetRowsArraysSUFFIX(struct par\
MatrixSparseSUFFIX *m, __int64_t size, int **rows,\
__int64_t size2, int **cols, double **real, double **imag);

void smg2sSUFFIX(struct parMatrixSparseSUFFIX \
 *m, __int64_t probSize, struct NilpotencyLongInt \
*nilp, __int64_t lbandwidth, char *spectrum, MPI_Comm comm);
```

5.2 Interface to Python

SMG2S uses SWIG to generate the wrapper of SMG2S to Python. Generate the shared library and install the python module of smg2s.

```
#install online from pypi
CC=mpicxx pip install smg2s

#bulid in local
cd ./interface/python;
CC=mpicxx python setup.py build_ext --inplace
#or
CC=mpicxx python setup.py build
#or
CC=mpicxx python setup.py install

#run
mpirun -np 2 python generate.py
```

Before the utilization, make sure that **mpi4py** is installed.
This is a little example of usage :

```
from mpi4py import MPI
import smg2s
import sys

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

sys.stdout.write(
"Hello, World! I am process %d of %d on %s.\n"
% (rank, size, name))

if rank == 0:
    print ('INFO ]> Starting ...')
    print("INFO ]> The MPI World Size is %d" %size)

#bandwidth for the lower band of initial matrix
lbandwidth = 3

#create the nilpotent matrix
nilp = smg2s.NilpotencyInt()

#setup the nilpotent matrix:
nilp.NilpType1(2,10)

Mt = smg2s.parMatrixSparseDoubleInt()

#Generate Mt by SMG2S
#vector.txt is the file that stores the given    \
spectral distribution in local filesystem.
Mt=smg2s.smg2sDoubleInt(10,nilp,lbandwidth,        \
"vector.txt", MPI.COMM_WORLD)
```

5.3 Interface to PETSc

SMG2S provides the interface to scientific computational softwares PETSc/SLEPc.
The way of Usage:
Include the header file:

```
#include <interface/PETSc/petsc_interface.h>
```

Create parMatrixSparse type matrix :

```
parMatrixSparse<std::complex<double>,int> *Mt;
```

Restore this matrix into CSR format :

```
Mt->Loc_ConvertToCSR();
```

Create PETSc MAT type :

```
MatCreate(PETSC_COMM_WORLD,&A);
```

Convert to PETSc MAT format :
Create PETSc MAT type :

```
A = ConvertToPETSCMat(Mt);
```

Here are the example of Arnoldi, GMRES, and another Krylov method.

5.4 Interface to Trilinos/Teptra

SMG2S is able to convert its distributed to the CSR one-dimensional distributed matrix defined by Teptra in Trilinos.
The way of usage:
Include header file

```
#include <interface/Trilinos/trilinos_interface.hpp>
```

Create parMatrixSparse type matrix :

```
parMatrixSparse<std::complex<double>,int> *Mt;
```

Create Trilinos/Teptra MAT type :

```
parMatrixSparse<std::complex<double>,int> *Mt;
```

Convert to Trilinos MAT format :

```
K = ConvertToTrilinosMat(Mt);
```

Here is a full example of Trilinos.

5.5 Create Your Inferface

On each process, the submatrix is stored by the std::map<T,S> provided by C++, which can be gotten through the function *GetDynMatLoc* implemented in the sparse matrix implementation. The column index and related entry value can be gotten by the C++ iterator.

```
parMatrixSparse<S, T > *M

T col;
S val;

/*On each process*/
parVectorMap<T> *pmap = M->GetYMap();

/*Get row number on each proc*/
T lsize = pmap->GetLocalSize();

std::map<T, S> *dynloc;
std::map<T, S>::iterator it;

dynloc = M->GetDynMatLoc();
/*Get col indices and values*/
for(T i = 0; i < lsize; i++){
 for(it = dynloc[i].begin(); it != dynloc[i].end(); ++it){
         col = it->first;
         val = it->second;
 }
}
```

Chapter 6

Verification of Eigenvalues

SMG2S provides the functionality to verify the abilibity to keep given spectrum. In the directory of **./verification/**. The implementation of the functionality is powerInverse.cpp.

6.1 Prerequisites

The verification method is implemented based on the shifted inverse method proposed by PETSc/SLEPc. Before the starting of verification, it is necessary to have the two on the platforms.

If not, the download and installation of PETSc can be found: [PETSc Download] and [SLPEc Installation]. The download and installation of PETSc can be found: [SLEPc Download] and [SLEPc Installation].

6.2 Verifcation by Shifted Inverse Power Method

1. compile the file powerInverse.cpp by the command

```
make
```

This will generate an executable powerInverse.exe.

2. Suppose the given eigenvalues are stored in the file vector.txt by the pseudo-Matrix Matrix Vector format, run the verification script as below:

```
#!/bin/bash
EXEC=./powerInverse.exe
N=100
L=10
TEST_TOL=0.00001
DEGREE=4

LENGTH=$(awk 'NR==2{print $1}' vector.txt)
echo "Test Eigenvalues number = "${LENGTH}

for ((i=3;i<=${LENGTH}+2;i++))
do
```

```
real=$(awk 'NR=='$i'{print $2}' vector.txt)
imag=$(awk 'NR=='$i'{print $3}' vector.txt)
srun -n 1 ${EXEC} -n ${N} -l ${L} -eps_monitor_conv   \
-eps_power_shift_type constant -st_type sinvert      \
-exact_value ${real}+${imag}i -test_tol ${TEST_TOL}   \
-degree ${DEGREE}
done
```

Here we list the meaning of the critical parameters in the script above:

- N: the size of matrix to generate, which should be equal to the number of given eigenvalues;
- L: the bandwidth of low part diagonal of matrix to generate;
- TEST_TOL: the tolerance to check if the accuracy of one eigenvalue can be accepted or not;
- DEGREE: the continuous one for the nilpotency matrix.

6.3 Script for result cleaning

The result file generated during the verification can be cleaned into the pseudo-Matrix Market Vector by the script below:

```
#!/bin/bash

grep "@> The eigenvalue" $1 > tmp.txt
awk '{print $5 " " $7 }' tmp.txt > tmp2.txt
awk '{print substr($0, 1, length($0)-1)}' tmp2.txt \
> tmp3.txt

awk '{print NR  " " $0}' tmp3.txt > tmp4.txt
NB=`wc -l tmp4.txt | awk '{print $1}'`
awk 'BEGIN{print '$NB' " " '$NB' " " '$NB'}{print}' \
tmp4.txt  > tmp5.txt

awk 'BEGIN{print "%%MatrixMarket matrix coordinate \
 real general"}{print}' tmp5.txt   > $2

rm tmp.txt tmp2.txt tmp3.txt tmp4.txt tmp5.txt
```

Execution of this script:

```
  ./traitement.sh results.txt results_clean.txt
```

In this command, the 1st and 2nd arguments for the execution are separately the initial results file and the final cleaned and formatted file.

6.4 Plot by Graphic User Interface

6.4.1 Prerequisites for GUI

You need to have Python2.X or Python3.X to run it. Moreover, UI uses some libraries to support a dynamic and intuitive graphical user interface, you can see the list of libraries. Normally, some of them are included in Python distribution. You can find below the list of necessary libraries of the UI.

- Modules which are bundled in the Python installation: Tkinter, re, sys, decimal ;
- Modules which need to be installed in addition to Python: NumPy & SciPy, Matplotlib, Pillow(PIL)

Install modules to Python 2.X:

```
apt-get install python-tk python-imaging-tk
pip -m install Pillow
python -mpip install -U pip
python -mpip install -U matplotlib
pip install -U numpy scipy
```

Install modules to Python 3.X:

```
sudo apt-get install python3-tk python-imaging-tk
pip -m install Pillow
python -mpip install -U pip
python -mpip install -U matplotlib
pip install -U numpy scipy
```

6.4.2 How to use the GUI

To use the GUI:

```
python main.py
```

When you launch the program, a new windows opens like Fig. 6.4.2 :

The first step is to select the files which be display. When you have selected a file, the button change in green color as Fig. 6.4.2 (Attention, the files imported should be in the pseudo-Matrix Market vector format that we have talked):

After that, you can click on "Display" to build and open the graphic on the right side of the window. Click on "New window" to open your graphic on a new window. It's possible to open as many windows as you want like Fig. 6.4.2:

It will be generate with automatic lens scaling, but your can generate it with your own scales as Fig. 6.4.2:

Figure 6.1: Home Screen Capture

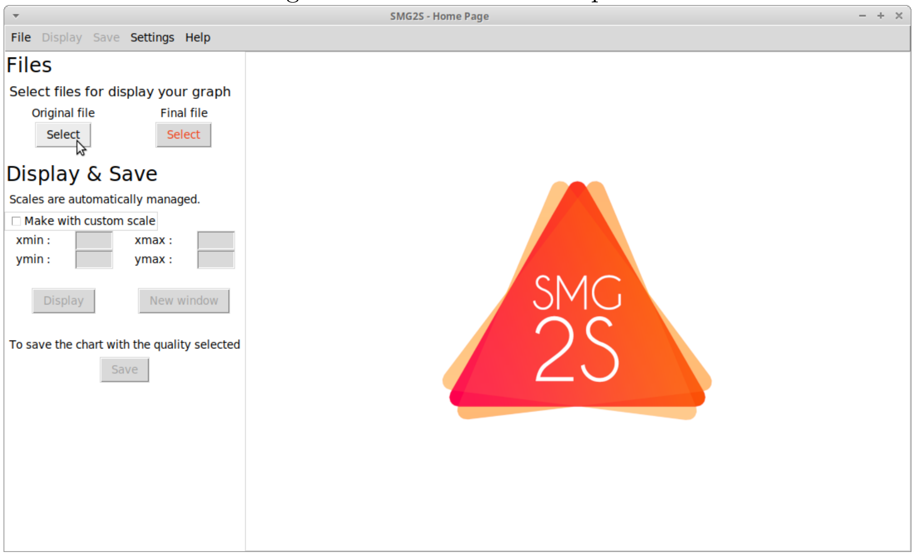


Figure 6.2: Home Screen Capture After Selection

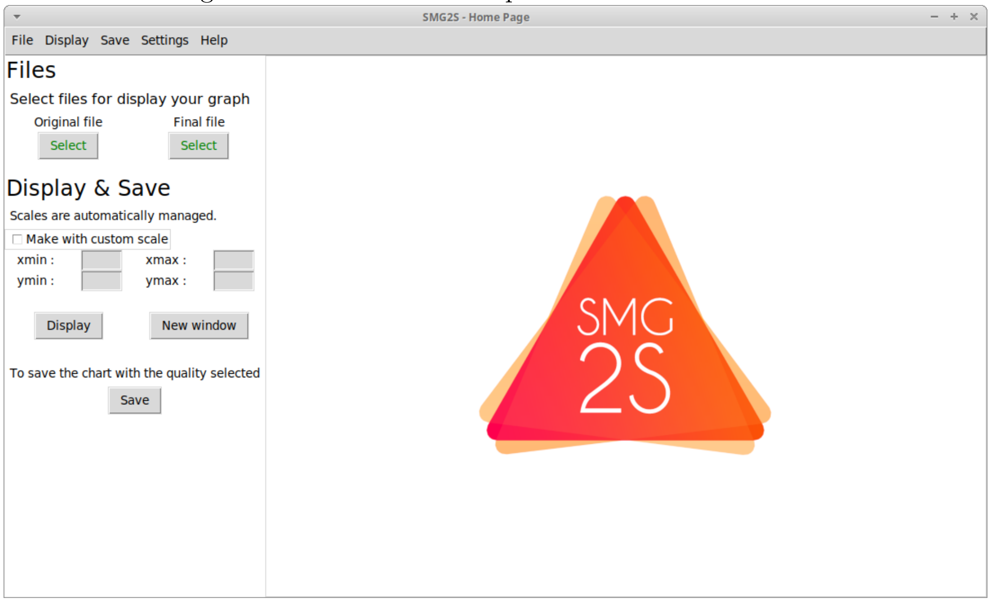


Figure 6.3: Home Screen Plot Capture

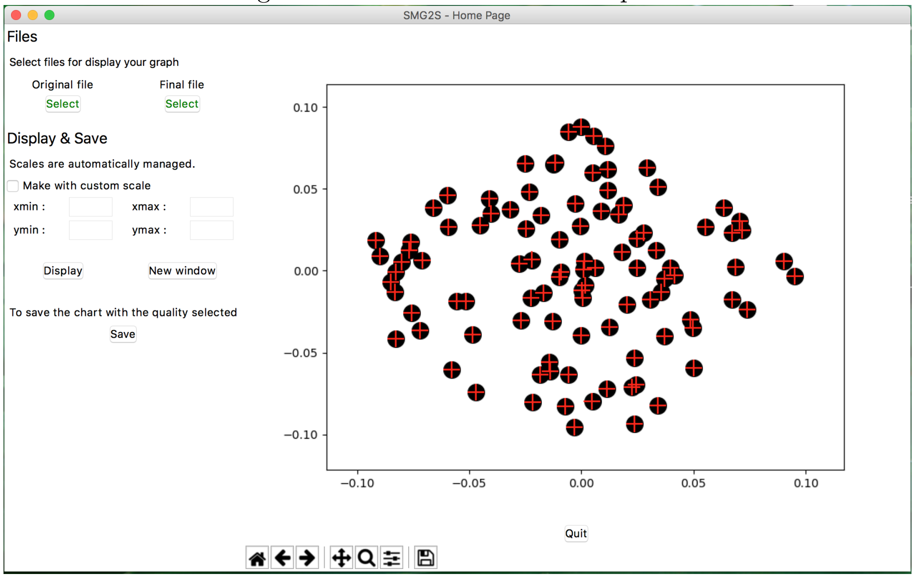
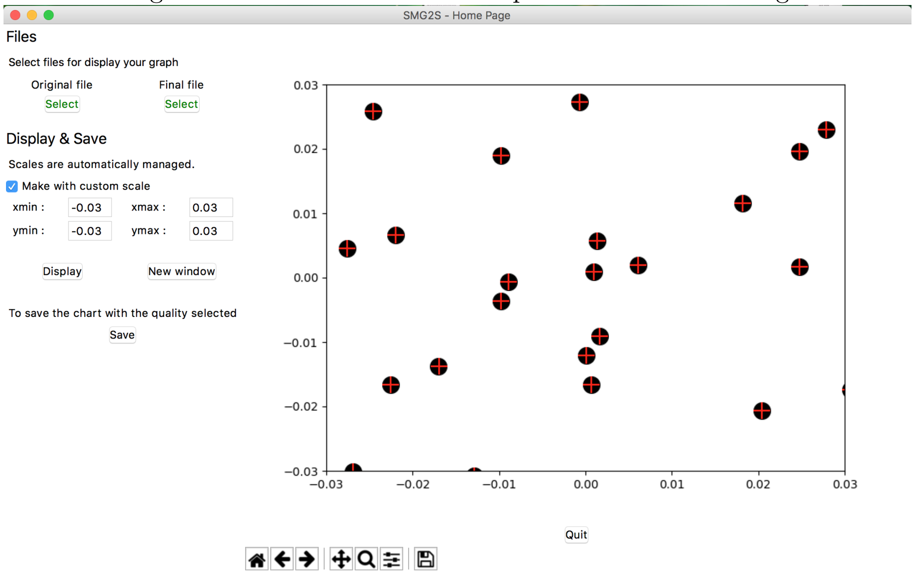


Figure 6.4: Home Screen Plot Capture with Lens Scaling



Chapter 7

Generation Methodologies

In this chapter, we summarize the methodologies for the generation of both non-Hermitian and non-Symmetric matrices.

7.1 Theorem

Theorem 1 *Let's consider a collection of matrices $M(t) \in \mathbb{C}^{n\times n}$, $n \in \mathbb{N}^*$. If $M(t)$ verifies:*

$$\begin{cases} \dfrac{dM(t)}{dt} = AM(t) - M(t)A, \\ M(t=0) = M_0. \end{cases}$$

Then $M(t)$ and M_0 are similar. $M(t)$ has the same eigenvalues as M_0.

Based on this theorem proposed by H. Gachlier [1], a matrix M_0 with given spectra can be transferred to another one $M(t)$ that satisfies *Theorem* 1 and keeps the spectra of M_0. SMG2S is proposed by selecting many parameters such as the matrices A and M_0.

In details, we select A as a nilpotent matrix shown as Fig. 3.1, and M_0 can be an initial simple matrix with customized eigenvalues, e.g., M_0 can be diagonal, block diagonal, upper triangular and lower triangular.

7.2 Algorithm

For the selected A and M_0, the procedure of SMG2S is simple, which is shown as Algorithm 1. Firstly, it reads an array $Spec_{in} \in \mathbb{C}^n$, as the given eigenvalues. Secondly, it generates an initial M_0 with $Spec_{in}$, and scales it with $(2d)!$. Meanwhile, it generates a nilpotent matrix A with the parameters d and p. More details about the generation of A can be found in Section 3.1. The final matrix M_t can be generated as $M_t = \frac{1}{(2d)!}M_{2d}$, where M_{2d} is the result after $2d$ times of loop $M_{i+1} = M_i + (\prod_{k=i+1}^{2d} k)(\widetilde{A_A})^i(M_0)$.

Algorithm 1 Matrix Generation Method

```
1: function MATGEN($input$:$Spec_{in} \in \mathbb{C}^n$, $p$, $d$, $h$, $output$: $M_t \in \mathbb{C}^{n \times n}$)
2:     Insert the entries in $h$ lower diagonals of $M_o \in \mathbb{C}^{n \times n}$
3:     Generate $M_0$ with $Spec_{in}$, then $M_0 = (2d)!M_0$
4:     Generate nilpotent matrix $A \in \mathbb{C}^{n \times n}$ with selected parameters $d$ and $p$
5:     for $i = 0, \cdots, 2d-1$ do
6:         $M_{i+1} = M_i + (\prod_{k=i+1}^{2d} k)(\widetilde{A_A})^i(M_0)$
7:     $M_t = \frac{1}{(2d)!}M_{2d}$
```

7.3 Initialization of M_0

There are different ways to form the initial matrix M_0 with given spectrum $Spec_{in}$. The good selection of this matrix can introduce different properties of generated matrices. In this section, we introduce the initialization of M_0 for generating non-Hermitian matrices and non-Symmetric matrices with complex eigenvalues.

7.3.1 Non-Hermitian Case

In order to generate non-Hermitian matrices, M_0 can be constructed as a lower triangular matrix having h non-zero diagonals. The main diagonal is filled with $Spec_{in}$, and the rest $h-1$ lower diagonals are filled with randomly generated values. Fig. 7.1 gives an example of both initial and generated non-Hermitian matrices.

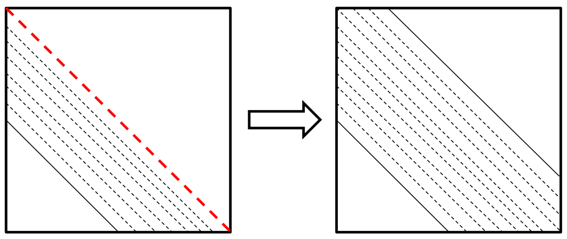


Figure 7.1: Matrix generation example.

7.3.2 Non-Symmetric Case

For the non Symmetric matrix whose entries are all real, if exist, they have pairs of conjugate complex eigenvalues which are symmetric to the real axis in the real-imaginary plain. For instance, denote one pair as $a + bi$ and $a - bi$. They are eigenvalues of a 2×2 matrix B as

$$B = \begin{bmatrix} a & b \\ -b & a \end{bmatrix}$$

In this case, M_0 can also be constructed as a lower triangular matrix having h non-zero diagonals. In the first step, M_0 can be initialized as a block diagonal matrix, which is constructed with a series 2×2 matrices like B for all the conjugate eigenvalues, shown as the left figure of Fig. 7.2. The rest $h-2$ lower diagonals are filled with randomly generated real values. Fig. 7.2 gives an example of both initial and generated non-Hermitian matrices.

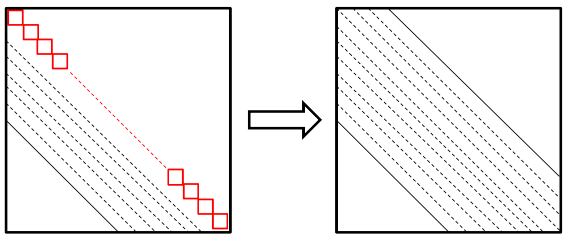


Figure 7.2: Matrix generation example.

Bibliography

[1] H. Galicher, F. Boillod-Cerneux, S. Petiton, and C. Calvin. Generate very large sparse matrices starting from a given spectrum.

[2] X. Wu, S. Petiton, and Y. Lu. A parallel generator of non-hermitian matrices computed from given spectra. In *VECPAR 2018: 13th International Meeting on High Performance Computing for Computational Science*, 2018.