1. Introduction

This document is to discuss how to replace function **solve_**(int *matrixid, double *rhs_sol, int *ierr) in M3D-C1 with a set of individual functions, therefore, PPPL is able to access and control the M3D-C1 matrices freely.

Currently the computed matrices from M3D-C1 have been stored in a data structure which consists of row, column, and computed values (either real or complex). The function **solve_(..)** is designed as an interface function which transfers the computed values from the data structure to the matrix format needed by the selected solver (SuperLU and Petsc have been considered) and solve the matrix afterwards. The current implementation is convenient and effective for the users who do not want to get into detail about how a selected solver to solve a matrix. However, considering that assembling and solving the matrices are encapsulated inside **solver_(...)**, it's not convenient for users who want to investigate the matrices. Therefore, breaking this function into a set of individual interface functions can provide the flexibility for users to access the data structure to retrieve the computed values for matrices, apply operations (additions, multiplications, and subtractions etc.) on the matrices, and check the matrices validities etc.

Therefore, solving a M3D-C1 matrix can be executed either by using the **solve_** (..) or using an combination of several functions (proposed as below). In the later situation, the SCOREC interface provides the functionality for PPPL to query any needed information from the data structure. Therefore, PPPL can assemble and solve the matrices and have fully ownership of the matrices.

2. Proposed functions

The following functions are proposed to provide the necessary information for PPPL to assemble and solve one matrix.

1. void checkMatrixStatus(int *matrixId, int *true/false);
   Description: check if a matrix with matrixId needs to be solved.
   Input: matrixId
   Output: true – to be solved
   false – not to be solved

2. void getMatrixLocalDofNum(int *matrixId, int *ldb);

Description: get the total number of degree of freedom on the processor
Input: matrixId
Output: ldb

3. void getMatrixFirstDof(int *matrixId, int *firstdof);
   Description: get the first dof in the proc.
   Input: matrixId
   Output: firstdof

4. void getMatrixGlobalDofs(int *matrixId, int *numglobaldofs);
   Description: get the global number of dofs.
   Input: matrixId
   Output: numglobaldofs

5. void getMatrixPetscDnnzOnnz(int *matrixId, int *valType, int *d_nnz, int *o_nnz)
   Description: set the d_nnz (array containing the number of nonzeros in the various rows of the DIAGONAL portion of the local submatrix) and o_nnz (array containing the number of nonzeros in the various rows of the OFF-DIAGONAL portion of the local submatrix)
   Input: matrixId
   valType: the matrix can be real (valType = 0) or complex (valType = 1)
   Output: d_nnz and o_nnz where d_nnz and o_nnz have the size of ldb from function 2

6. void getMatrixNNZRowSize(int *matrixId, int *valType, *int rowSize)
   Description: get the row size of nonzero values
   Input: matrixId, valType
   Output: rowSize

7. void getMatrixNNZRowId(int *matrixId, int *valType, int *ith, int *rowId)
   Description: get the ith rowID of the nonzero valuves
   Input: matrixId, valType
   ith: 0<= ith <rowSize
   Output: rowId

8. void getMatrixNNZColSize(int *matrixId, int *valType, int *rowId, int *colSize)
   Description: get the column size of the nonzero values for one particular row with input rowID
   Input: matrixId, valType, rowId (the same as function 4)

Output: colSize

9. void getMatrixNNZValues(int *matrixId, int *valType, int *rowId, int *colId, double *values)

Description: get the nonzero values and the corresponding column Ids for one particular rwo with input rowID

Input: matrixId, valType, rowId

Output: colId – all of the nonzero column ids for the input rowId
values – all of the nonzero values for the input rowId

10. void setMatrixSoln(int *matrixId, double *sol)

Description: associate the solutions of the matrixId with the mesh

Input: matrixId, sol

11. void cleanMatrixValue(int *matrixId)

Description: clean all of the data associated with the matrixId

3. Applications

A pesudo code (in C) is given below to demonstrate how to solve one matrix with a given matrixId using SuperLU solver using the above functions.

```
// declare variables
  int flag, ldb, m_loc;
  int *rowptr, *colind;
  int nnz_loc = 0;  // number of nonzeros at the local processors – real
  double *nzval = 0;
  int counter;
  int rowSize, rowId;
  int colSize, *colId;
  double *values;
  int i;
  int numglobaldofs, firstdof;

// variables for superlu
  SuperMatrix A;
  superlu_options_t options;
  ScalePermstruct_t ScalePermstruct;
  SuperLUStat_t stat;
```

```
  LUstruct_t LUstruct;
  SOLVEstruct_t SOLVEstruct;

// Step 1: check the matrix status
     checkMatrixStatus(*matrixId, &flag);
     if(flag==0)   // the matrix does not need to besolved
          return;

// Step 2: get the local dofs
      getMatrixLocalDofNum(*matrixId, &ldb);
      m_loc = ldb;

// Step 3: allocate memory for rowptr
      if( !(rowptr = intMalloc_dist(m_loc+1))) ABORT("Malloc failed for rowptr.");

// Step 4: set up rowptr and nnz_loc
     rowptr[0] = 0;
     counter = 1;
     getMatrixNNZRowSize(*matrixId,  *valType,  &rowSize)
     for(i=0; i<rowSize; i++) {
        getMatrixNNZRowId( *matrixId, *valType, i, &rowId);
        getMatrixNNZColSize(*matrixId, *valType, &rowId,  &colSize);
        rowptr[counter] = colSize + rowpter[counter-1];
        counter++;
        nnz_loc += colSize;
     }

// Step 5: allocate memory for nnz_loc
      if( !(nzval = doubleMalloc_dist(nnz_loc)) ) ABORT("Malloc failed for nzval.");
      if( !(colind = intMalloc_dist(nnz_loc)) ) ABORT("Malloc failed for colind");

// Step 6: put the computed values to the nzval array
     counter = 0;
    for(i=0; i<rowSize; i++) {
        getMatrixNNZRowId( *matrixId, *valType, i, &rowId);
        getMatrixNNZColSize(*matrixId, *valType, &rowId,  &colSize);
        colId = malloc(colSize*sizeof(int));
        values = malloc(colSize*sizeof(double));
```

```
      getMatrixNNZValues(*matrixId, *valType,  &rowId,  colId, values);
      memcpy(nzval+counter, values, colSize); // copy the nonzeros to the nzval
      counter += colSize;
      free(colId);
      free(values);
    }
   if(counter != nnz_loc)
     cout << P_pid() << " has difference in count of " << counter - nnz_loc << endl;


// Step 7: set up the matrix for superLU solver
    getMatrixGlobalDofs(*matrixId, &numglobaldofs);
    getMatrixFirstDof(*matrixId, &firstdof);

    dCreate_CompRowLoc_Matrix_dist(&A, numglobaldofs, numglobaldofs, nnz_loc,
                                          m_loc, firstdof, nzval, colind, rowptr,
                                          SLU_NR_loc, SLU_D, SLU_GE);
// Step 8: clean the unused data
   cleanMatrixValue(*matrixId);

// Step 9: solver the superLU matrix A
    set_default_options_dist(&options);
    ScalePermstructInit(A.nrow, A.ncol, &ScalePermstruct);
    LUstructInit(A.nrow, A.ncol, &LUstruct);
    PStatInit(&stat);
    pdgssvx(&options, &A, &ScalePermstruct, rhs_sol, ldb, 1, &grid, &LUstruct,
            &SOLVEstruct, &berr, &stat, ier);

// Step 10: set back the matrix solution to mesh
     setMatrixSoln(*matrixId, rhs_sol);
```