Free Implementation of GMRES in C++

Version 0.2

Fri Jul 7 2023

Kelly Black <kjblack@gmail.com>

Contents

Chapter 1

Overview

1.1 Introduction

The codes given here are a free implementation of the Generalized Minimal Residual Method (GMRES) in c++. The method is fully described by Saad[5] and Kelley[4]. The code given here is based on the pseudo code given by Barrett *et al*[1]. The codes were adapted to c++ after examining the matlab codes by Burkardt[2]. This specific implementation makes use of restarts, and it was most influenced by the code available from the United States' National Institute of Standards and Technology[3].

The GMRES algorithm is defined in the file GMRES.h. The algorithm is given in a template function, named `GMRES`. The subroutine assumes that three classes are defined that include several specific operators and functions. There is an additional subroutine called by `GMRES` named `Update`. This subroutine is used to generate an updated approximation based on the Krylov subspace generated within the `GMRES` subroutine.

There is an additional set of subroutines that are used within the `GMRES` subroutine. These routines are defined in the files `util.h` and `util.cpp`. The file `util.h` includes the file `util.cpp`. Both files are assumed to be in the same directory as the `GMRES.h` file.

In this overview the call for the GMRES routine is first given, and then the three required classes are stated in turn. First the `Operation` class is discussed, then the `Approximation` class, and finally the `Preconditioner` class is discussed. Each class must implement specific operations and define a given set of methods. The expectations are given within each section.

1.2 Calling the GMRES Subroutine

The GMRES routine is named `GMRES`, and the definition for the method is given in Listing 1.1. There are seven parameters for the function. The first four parameters are pointers to the respective classes. The next three parameters dictate the behaviour and limits of the GMRES implementation and include the number of vectors in the Krylov subspace, the number of restarts, and the tolerance respectively.

Listing 1.1: The definition for the GMRES routine.

```
template<class Operation,class Approximation,class Preconditioner,class Double>
int GMRES
(Operation* linearization, //!< Performs the linearization of the PDE on the approximation.
 Approximation* solution,  //!< The approximation to the linear system. (and initial estimate!)
 Approximation* rhs,       //!< the right hand side of the equation to solve.
 Preconditioner* precond,  //!< The preconditioner used for the linear system.
 int krylovDimension,      //!< The number of vectors to generate in the Krylov subspace.
 int numberRestarts,       //!< Number of times to repeat the GMRES iterations.
 Double tolerance          //!< How small the residual should be to terminate the GMRES iterations.
 )
```

The basic idea is that an approximation to a linear system is to be generated. The system can be represented by

$$L\vec{x} = \vec{b}. \tag{1.1}$$

The parameters in the subroutine correspond to the following symbols in equation (1.1):

$$\begin{aligned} L &= \texttt{linearization}, \\ \vec{x} &= \texttt{solution}, \\ \vec{b} &= \texttt{rhs}. \end{aligned}$$

The parameter in the `Operation` class, `linearization`, is used to implement the action of the matrix multiplied by a vector. The vector is given by an object in the `Approximation` class which includes the initial estimate, `solution`, and the right hand side, `rhs`. The variable `solution` is updated, and it is changed by calling the routine.

The final class, `Preconditioner`, is used to implement a preconditioner for the system. The assumption is that the preconditioned system is in the form

$$P^{-1}L\vec{x} = P^{-1}\vec{b}.$$

The GMRES algorithm requires that the operator, `precond` in the GMRES routine, be used to generate a sequence of vectors given by

$$\vec{v}_{n+1} = L\hat{v}_n,$$

where $\hat{v}_n$ is a normalized vector that is orthogonal to the previous vectors generated. The `GMRES` subroutine calculates this under the assumption that the system to solve is given by the system

$$P\vec{v}_{n+1} = L\hat{v}_n, \tag{1.2}$$

where the routine solves for the vector $\vec{v}_{n+1}$.

Listing 1.2: The definition for the Update routine.

```
template <class Approximation, class Double >
void
Update
(Double **H,        //<! The upper diagonal matrix constructed in the GMRES routine.
 Approximation *x,  //<! The current approximation to the linear system.
 Double *s,         //<! The vector e_1 that has been multiplied by the Givens rotations.
 std::vector<Approximation> *v,   //<! The orthogonal basis vectors for the Krylov subspace.
 int dimension)     //<! The number of vectors in the basis for the Krylov subspace.
)
```

Finally, the file includes an additional subroutine, `Update`. This subroutine is used to perform the back-solve and update to determine the next approximation to the linear system. This is used within the `GMRES` subroutine and is not expected to be called by another routine.

1.3 The Operation Class

The first class examined is the `Operation` class. This class is used to perform the actions equivalent to a matrix multiply. It is assumed that this is in the form of a right multiply, i.e.

$$L \cdot \vec{x}.$$

Listing 1.3: An example of the multiply operation defined for the Operation class.

```
Approximation Operation::operator*(class Approximation vector)
{
        Solution result;
    ...
    return(result);
}
```

The `Operation` class must have a number of operations defined. In particular the class should have the multiply operator defined. An example of the required definition is given in Listing 1.3.

Methods	Operations
`norm`	$+$
`getN`	$-$
2 constructors	$+=$
axpy	$=$
	$*$

Table 1.1: The methods and operations that must be defined for the `Approximation` class.

1.4 The Approximation Class

The `Approximation` class is used to keep track of the approximation to the solution to the linear system. It is the class used to store $\vec{x}$ as well as $\vec{b}$ in equation (1.1). It is assumed that the `Approximation` class has a number of methods and operations defined, and the complete list is given in Table 1.1.

Examples of the basic form for the definitions are given in Listing 1.4. Note the types of the arguments. For example, the addition operator is for the sum of two objects from the `Approximation` class while the multiplication operator is for an object from the `Approximation` class multiplied on the right by a real valued variable.

Listing 1.4: An example of the operations that must be defined for the `Approximation` class.

```
Approximation Approximation::operator+(const Approximation& vector)
{
        Approximation result(this->getN());
    ...
        return(result);
}


Approximation Approximation::operator-(const Approximation& vector)
{
        Approximation result(this->getN());
    ...
        return(result);
}

Approximation Approximation::operator+=(const Approximation& vector)
{
    ...
        return(*this);
}

Approximation Approximation::operator=(const Approximation& vector)
{
    ...
        return(*this);
}

Approximation Approximation::operator*(const double& value)
{
        Approximation result(this->getN());
    ...
        return(result);
}
```

There are four methods that must be defined for the `Approximation` class. Examples of the definitions are shown in Listing 1.5. Note that there are two constructors that must be called. The first is a constructor that requires the number of entries to allocate in the approximation. The second is a constructor that makes a copy of the `Approximation` object passed to it.

Also note that the `axpy` method is a method that adds a scalar multiplied by another `Approximation` object to the current object. This method is used in the modified Gramm-Schmidt orthogonalization of the vectors that make up the Krylov subspace.

Listing 1.5: An example of the methods that must be defined for the `Approximation` class.

```
Approximation::Approximation(int size)
{
  ...
}

Approximation::Approximation(const Approximation& oldCopy)
{
```

```
  ...
}

double Approximation::norm(const Approximation& v1)
{
        double norm = v1.getEntry(0)*v1.getEntry(0);
        int lupe;
        for(lupe=v1.getN();lupe>0;--lupe)
                {
                        norm += v1.getEntry(lupe)*v1.getEntry(lupe);
                }
        return(sqrt(norm));
}


int Approximation::getN() const
{
  return(N);
}

void Approximation::axpy(Approximation* vector,
                                        double multiplier)
{
        int lupe;
        for(lupe=getN();lupe>=0;--lupe)
                {
                        setEntry(getEntry(lupe)+multiplier*vector->getEntry(lupe),lupe);
                }
}
```

1.5 The Preconditioner Class

The final class is the `Preconditioner` class. This class only requires one method, the `solve` method. This method is used to solve the system given in equation (1.2). An example of the definition is given in Listing 1.6. Notice that it returns an object from the `Approximation` class.

Listing 1.6: An example of the solve method that must be defined for the `Preconditioner` class.

```
Approximation Preconditioner::solve(const Approximation &current)
{
        Approximation multiplied(current);

        // Perform the forward solve to invert the first part of the
        // Cholesky decomposition.
        int lupe;
        intermediate[0] = current.getEntry(0)/vector[0][0];
        for(lupe=1;lupe<=getN();++lupe)
                intermediate[lupe] =
                        (current.getEntry(lupe)-vector[lupe][1]*intermediate[lupe-1])
                        /vector[lupe][0];

        // Perform the backwards solve for the Cholesky decomposition.
        multiplied(getN()) = intermediate[getN()]/vector[getN()][0];
        for(lupe=getN()-1;lupe>=0;--lupe)
                multiplied(lupe) = (intermediate[lupe]-multiplied(lupe+1)*vector[lupe+1][1])
                        /vector[lupe][0];

        // The previous solves wiped out the boundary conditions. Restore
        // the left and right boundaru condition before sending the result
        // back.
        multiplied(0) = current.getEntry(0);
        multiplied(getN()) = current.getEntry(getN());
        return(multiplied);
}
```

Chapter 2

GMRES

A c++ implementation of the GMRES method for approximating the solution to a linear system.
This set of software is composed of Several include files that define template functions to implement the GMRES method. The code is based on the GMRES method with restarts. The code is influenced by the IML++ implementation as well as John Burkardt's MATLAB implementation.
Please see the file latex/refman.pdf for more details on how to use the code. An example of how to use the code can be found in the example directory.

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 ArrayUtils< number > Class Template Reference

Header file for the basic utilities associated with managing arrays.
`#include <util.h>`

Public Member Functions

- ArrayUtils ()

Static Public Member Functions

- static number ***** fivetensor (int n1, int n2, int n3, int n4, int n5)
- static number **** fourtensor (int n1, int n2, int n3, int n4)
- static number *** threetensor (int n1, int n2, int n3)
- static number ** twotensor (int n1, int n2)
- static number * onetensor (int n1)
- static void delfivetensor (number *****u)
- static void delfourtensor (number ****u)
- static void delthreetensor (number ***u)
- static void deltwotensor (number **u)
- static void delonetensor (number *u)

5.1.1 Detailed Description

template<class number>
class ArrayUtils< number >

Header file for the basic utilities associated with managing arrays.

Author

Kelly Black `kjblack@gmail.com`

Version

0.1

Copyright

BSD 2-Clause License

5.1.2 LICENSE

5.1.3 DESCRIPTION

Class to provide a set of basic utilities that are used by numerous other classes.
This is the definition (header) file for the ArrayUtils class. It includes the definitions for the methods that are used to construct and delete arrays used in a variety of other classes.

5.1.4 Constructor & Destructor Documentation

5.1.4.1 ArrayUtils()

```
template<class number >
ArrayUtils< number >::ArrayUtils ( ) [inline]
```

Base constructor for the ArrayUtils class.
There is not anything to do so this is an empty method.

5.1.5 Member Function Documentation

5.1.5.1 delfivetensor()

```
template<class number >
void ArrayUtils< number >::delfivetensor (
            number ***** u ) [static]
```

Template for deleting a five dimensional array.

Parameters

u	pointer to the array to be deleted.

5.1.5.2 delfourtensor()

```
template<class number >
void ArrayUtils< number >::delfourtensor (
            number **** u ) [static]
```

Template for deleting a four dimensional array.

Parameters

u	pointer to the array to be deleted.

5.1.5.3 delonetensor()

```
template<class number >
void ArrayUtils< number >::delonetensor (
            number * u ) [static]
```

Template for deleting a one dimensional array.

Parameters

u	pointer to the array to be deleted.

5.1.5.4 delthreetensor()

```
template<class number >
void ArrayUtils< number >::delthreetensor (
            number *** u ) [static]
```

Template for deleting a three dimensional array.

Parameters

u	pointer to the array to be deleted.

5.1.5.5 deltwotensor()

```
template<class number >
void ArrayUtils< number >::deltwotensor (
            number ** u ) [static]
```

Template for deleting a two dimensional array.

Parameters

u	pointer to the array to be deleted.

5.1.5.6 fivetensor()

```
template<class number >
number ***** ArrayUtils< number >::fivetensor (
            int n1,
            int n2,
            int n3,
            int n4,
            int n5 ) [static]
```

Template for allocating a five dimensional array.

Parameters

n1	Number of entries for the first dimension.
n2	Number of entries for the second dimension.
n3	Number of entries for the third dimension.
n4	Number of entries for the fourth dimension.
n5	Number of entries for the fifth dimension.

Returns

A pointer to the array created.

5.1.5.7 fourtensor()

```
template<class number >
number **** ArrayUtils< number >::fourtensor (
            int n1,
            int n2,
            int n3,
            int n4 ) [static]
```

Template for allocating a four dimensional array.

Parameters

n1	Number of entries for the first dimension.
n2	Number of entries for the second dimension.
n3	Number of entries for the third dimension.
n4	Number of entries for the fourth dimension.

Returns

A pointer to the array created.

5.1.5.8 onetensor()

```
template<class number >
number * ArrayUtils< number >::onetensor (
            int n1 ) [static]
```

Template for allocating a one dimensional array.

Parameters

n1	Number of entries for the dimension.

Returns

A pointer to the array.

5.1.5.9 threetensor()

```
template<class number >
number *** ArrayUtils< number >::threetensor (
            int n1,
            int n2,
            int n3 ) [static]
```

Template for allocating a three dimensional array.

Parameters

n1	Number of entries for the first dimension.
n2	Number of entries for the second dimension.
n3	Number of entries for the third dimensions.

Returns

A pointer to the array created.

5.1.5.10 twotensor()

```
template<class number >
number ** ArrayUtils< number >::twotensor (
            int n1,
            int n2 ) [static]
```

Template for allocating a two dimensional array.

Parameters

n1	Number of entries for the first dimension.
n2	Number of entries for the second dimension.

Returns

A pointer to the array created.

The documentation for this class was generated from the following files:

- util.h
- util.cpp

Chapter 6

File Documentation

6.1 GMRES.h File Reference

Template files for implementing a GMRES algorithm to solve a linear sytem.

```
#include "util.h"
#include <cmath>
#include <vector>
```

Functions

- template<class Approximation , class Double >
  void Update (Double **H, Approximation *x, Double *s, std::vector< Approximation > *v, int dimension)
- template<class Operation , class Approximation , class Preconditioner , class Double >
  int GMRES (Operation *linearization, Approximation *solution, Approximation *rhs, Preconditioner *precond, int krylovDimension, int numberRestarts, Double tolerance)

6.1.1 Detailed Description

Template files for implementing a GMRES algorithm to solve a linear sytem.

Author

Kelly Black kjblack@gmail.com

Version

0.2

Copyright

6.1.2 LICENSE

6.1.3 DESCRIPTION

This file includes the template functions necessary to implement a restarted GMRES algorithm. This is based on the pseudo code given in the book Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition [1] .
Also, some changes were implemented based on the matlab code by John Burkardt [2] `http://people.sc.fsu.edu/~jbur`
The idea for using a template came from the IML++ code [3] `http://math.nist.gov/iml++/` Also, the method for calculating the entries for the Givens rotation matrices came from the IML++ code as well.
Additional sources that informed this work are Tim Kelley's book Iterative Methods for Linear and Nonlinear Equations [4] Another book is is Yousef Saad's book Iterative Methods for Sparse Linear Systems [5]

6.1.4 Function Documentation

6.1.4.1 GMRES()

```
template<class Operation , class Approximation , class Preconditioner , class Double >
int GMRES (
            Operation * linearization,
            Approximation * solution,
            Approximation * rhs,
            Preconditioner * precond,
            int krylovDimension,
            int numberRestarts,
            Double tolerance )
```

Implementation of the restarted GMRES algorithm. Follows the algorithm given in the book Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.

Returns

The number of iterations required. Returns zero if it did not converge.

Parameters

linearization	Performs the linearization of the PDE on the approximation.
solution	The approximation to the linear system. (and initial estimate!)
rhs	the right hand side of the equation to solve.
precond	The preconditioner used for the linear system.
krylovDimension	The number of vectors to generate in the Krylov subspace.
numberRestarts	Number of times to repeat the GMRES iterations.
tolerance	How small the residual should be to terminate the GMRES iterations.

6.1.4.2 Update()

```
template<class Approximation , class Double >
```

```
void Update (
            Double ** H,
            Approximation * x,
            Double * s,
            std::vector< Approximation > * v,
            int dimension )
```

Update the current approximation to the solution to the linear system. This assumes that the update is created using a GMRES routine, and the upper Hessenberg matrix has been transformed to an upper diagonal matrix already. Note that it changes the values of the values in the coefficients vector, s, which means that the s vector cannot be reused after this without being re-initialized.

6.2 GMRES.h

Go to the documentation of this file.

```

#include "util.h"
#include <cmath>
#include <vector>

template <class Approximation, class Double >
void Update
(Double **H,         //<! The upper diagonal matrix constructed in the GMRES routine.
 Approximation *x,   //<! The current approximation to the linear system.
 Double *s,          //<! The vector e_1 that has been multiplied by the Givens rotations.
 std::vector<Approximation> *v,  //<! The orthogonal basis vectors for the Krylov subspace.
 int dimension)      //<! The number of vectors in the basis for the Krylov subspace.
{

  // Solve for the coefficients, i.e. solve for c in
  // H*c=s, but we do it in place.
  int lupe;
  for (lupe = dimension; lupe >= 0; --lupe)
      {
          s[lupe] = s[lupe]/H[lupe][lupe];
          for (int innerLupe = lupe - 1; innerLupe >= 0; --innerLupe)
              {
                  // Subtract off the parts from the upper diagonal of the
                  // matrix.
                  s[innerLupe] -=  s[lupe]*H[innerLupe][lupe];
              }
      }

  // Finally update the approximation.
    typename std::vector<Approximation>::iterator ptr = v->begin();
  for (lupe = 0; lupe <= dimension; lupe++)
      *x += (*ptr++) * s[lupe];
}



template<class Operation,class Approximation,class Preconditioner,class Double>
int GMRES
(Operation* linearization,
 Approximation* solution,
 Approximation* rhs,
 Preconditioner* precond,
 int krylovDimension,
 int numberRestarts,
 Double tolerance
 )
{

    // Allocate the space for the givens rotations, and the upper
    // Hessenburg matrix.
    Double **H      = ArrayUtils<Double>::twotensor(krylovDimension+1,krylovDimension);

    // The Givens rotations include the sine and cosine term. The
    // cosine term is in column zero, and the sine term is in column
    // one.
    Double **givens = ArrayUtils<Double>::twotensor(krylovDimension+1,2);

    // The vector s the right hand side for the system that the matrix
    // H satisfies in order to minimize the residual over the Krylov
    // subspace.
    Double *s = ArrayUtils<Double>::onetensor(krylovDimension+1);

    // Determine the residual and allocate the space for the Krylov
    // subspace.
    std::vector<Approximation> V(krylovDimension+1,
                                 Approximation(solution->getN()));
```

```
    Approximation residual = precond->solve((*rhs)-(*linearization)*(*solution));
    Double rho             = residual.norm();
    Double normRHS         = rhs->norm();

    // variable for keeping track of how many restarts had to be used.
    int totalRestarts = 0;

    if(normRHS < 1.0E-5)
        normRHS = 1.0;

    // Go through the requisite number of restarts.
    int iteration = 1;
    while( (--numberRestarts >= 0) && (rho > tolerance*normRHS))
        {

            // The first vector in the Krylov subspace is the normalized
            // residual.
            V[0] = residual * (1.0/rho);

            // Need to zero out the s vector in case of restarts
            // initialize the s vector used to estimate the residual.
            for(int lupe=0;lupe<=krylovDimension;++lupe)
                s[lupe] = 0.0;
            s[0] = rho;

            // Go through and generate the pre-determined number of vectors
            // for the Krylov subspace.
            for( iteration=0;iteration<krylovDimension;++iteration)
                {
                    // Get the next entry in the vectors that form the basis for
                    // the Krylov subspace.
                    V[iteration+1] = precond->solve((*linearization)*V[iteration]);

                    // Perform the modified Gram-Schmidt method to orthogonalize
                    // the new vector.
                    int row;
                    typename std::vector<Approximation>::iterator ptr = V.begin();
                    for(row=0;row<=iteration;++row)
                        {
                            H[row][iteration] = Approximation::dot(V[iteration+1], *ptr);
                            //subtract H[row][iteration]*V[row] from the current vector
                            V[iteration+1].axpy(&(*ptr++),-H[row][iteration]);
                        }

                    H[iteration+1][iteration] = V[iteration+1].norm();
                    V[iteration+1] *= (1.0/H[iteration+1][iteration]);

                    // Apply the Givens Rotations to insure that H is
                    // an upper diagonal matrix. First apply previous
                    // rotations to the current matrix.
                    double tmp;
                    for (row = 0; row < iteration; row++)
                        {
                            tmp = givens[row][0]*H[row][iteration] +
                                givens[row][1]*H[row+1][iteration];
                            H[row+1][iteration] = -givens[row][1]*H[row][iteration]
                                + givens[row][0]*H[row+1][iteration];
                            H[row][iteration]  = tmp;
                        }

                    // Figure out the next Givens rotation.
                    if(H[iteration+1][iteration] == 0.0)
                        {
                            // It is already lower diagonal. Just leave it be....
                            givens[iteration][0] = 1.0;
                            givens[iteration][1] = 0.0;
                        }
                    else if (fabs(H[iteration+1][iteration]) > fabs(H[iteration][iteration]))
                        {
                            // The off diagonal entry has a larger
                            // magnitude. Use the ratio of the
                            // diagonal entry over the off diagonal.
                            tmp = H[iteration][iteration]/H[iteration+1][iteration];
                            givens[iteration][1] = 1.0/sqrt(1.0+tmp*tmp);
                            givens[iteration][0] = tmp*givens[iteration][1];
                        }
                    else
                        {
                            // The off diagonal entry has a smaller
                            // magnitude. Use the ratio of the off
                            // diagonal entry to the diagonal entry.
                            tmp = H[iteration+1][iteration]/H[iteration][iteration];
                            givens[iteration][0] = 1.0/sqrt(1.0+tmp*tmp);
                            givens[iteration][1] = tmp*givens[iteration][0];
                        }

                    // Apply the new Givens rotation on the
```

```
                    // new entry in the uppper Hessenberg matrix.
                    tmp = givens[iteration][0]*H[iteration][iteration] +
                        givens[iteration][1]*H[iteration+1][iteration];
                    H[iteration+1][iteration] = -givens[iteration][1]*H[iteration][iteration] +
                        givens[iteration][0]*H[iteration+1][iteration];
                    H[iteration][iteration] = tmp;

                    // Finally apply the new Givens rotation on the s
                    // vector
                    tmp = givens[iteration][0]*s[iteration] + givens[iteration][1]*s[iteration+1];
                    s[iteration+1] = -givens[iteration][1]*s[iteration] +
givens[iteration][1]*s[iteration+1];
                    s[iteration] = tmp;

                    rho = fabs(s[iteration+1]);
                    if(rho < tolerance*normRHS)
                        {
                            // We are close enough! Update the approximation.
                            Update(H,solution,s,&V,iteration);
                            ArrayUtils<double>::deltwotensor(givens);
                            ArrayUtils<double>::deltwotensor(H);
                            ArrayUtils<double>::delonetensor(s);
                            //delete [] V;
                            //tolerance = rho/normRHS;
                            return(iteration+totalRestarts*krylovDimension);
                        }

                } // for(iteration)

            // We have exceeded the number of iterations. Update the
            // approximation and start over.
            totalRestarts += 1;
            Update(H,solution,s,&V,iteration-1);
            residual = precond->solve((*linearization)*(*solution) - (*rhs));
            rho = residual.norm();

        } // while(numberRestarts,rho)


    ArrayUtils<double>::deltwotensor(givens);
    ArrayUtils<double>::deltwotensor(H);
    ArrayUtils<double>::delonetensor(s);
    //delete [] V;
    //tolerance = rho/normRHS;

    if(rho < tolerance*normRHS)
        return(iteration+totalRestarts*krylovDimension);

    return(0);
}
```

6.3 util.h File Reference

`#include "util.cpp"`

Classes

- class ArrayUtils< number >

    Header file for the basic utilities associated with managing arrays.

6.3.1 Detailed Description

6.4 util.h

Go to the documentation of this file.

```
#ifndef UTILROUTINE
#define UTILROUTINE


template <class number>
class ArrayUtils
{

public:

    ArrayUtils(){};
```

```

    // Define the methods used to allocate the memory for and define
    // multi-dimensional arrays.
    static number *****fivetensor(int n1,int n2,int n3,int n4,int n5);
    static number ****fourtensor(int n1,int n2,int n3,int n4);
    static number ***threetensor(int n1,int n2,int n3);
    static number **twotensor(int n1,int n2);
    static number *onetensor(int n1);

    // Define the methods used to delete the memory that was allocated
    // for multi-dimensional arrays.
    static void delfivetensor(number *****u);
    static void delfourtensor(number ****u);
    static void delthreetensor(number ***u);
    static void deltwotensor(number **u);
    static void delonetensor(number *u);

};


#include "util.cpp"


#endif

```

Bibliography

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994. 1, 18

[2] John Burkardt. gmres_r.m, November 2014. based on a code by Tim Kelley found at http://people.sc.fsu.edu/~jburkardt/m_src/toms866/solvers/gmres_r.m. 1, 18

[3] IML++. Gmres.h, November 2014. found at http://math.nist.gov/iml++/. 1, 18

[4] Tim Kelley. *Iterative Methods for Linear and Nonlinear Equations*. SIAM, Philadelphia, PA, 2004. 1, 18

[5] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, second edition edition, 2003. 1, 18

Index