Linear algebra system of equation solver

The linear algebra system of equation solver library provides a lightweight implementation of the Conjugate Gradient (CG) algorithm to iteratively solve the linear system of the form $Ax = b$. It is particularly well suited to solving partial differential equations for CFD applications using an implicit discretisation.

Introduction

Solving the linear system of equations in the form of $Ax = b$ can be achieved with a simple Gauss elimination. However, for large coefficients matrices A, such as occurring in CFD applications, inverting the original matrix A is not practical. Instead, an approximate procedure can be used which produces results much faster compared to a direct inversion of the matrix.

Furthermore, the coefficient matrix A is typically sparse, where most elements are zero. Storing the entire matrix would result in wasted storage and, in fact, in most cases, is not possible. Solving the system of the form $Ax = b$ requires an efficient matrix storage solution, which is commonly achieved by the compressed spare row (CSR)[1] matrix storage format.

The CSR matrix is efficient in the storage of data but more complicated in terms of inserting elements into the matrix. Since inserting and retrieving data can be somewhat more challenging, a dedicated vector class is also required to achieve an efficient matrix-vector multiplication.

Both of these classes are then used by a solver class which provides a base class to implement iterative solvers, for which one such implementation is provided in the form of the Conjugate Gradient (CG) algorithm. While this is the only provided algorithm, the matrix and vector class are general enough to allow for easy extension of the library by providing a dedicated solver class implementation.

Installation

The library comes with dedicated build scripts that compile and install the library into either a shared or static library. Different installation scripts are provided for different platforms.

All source code is provided within the `linearAlgebraLib` folder. In order to test that the library is working correctly, the compilation can be tested with a few tests that are provided in the `tests` folder. Within that folder, the file `mainTest.cpp` collects all tests, which in turn are provided in the `tests/unit`, `tests/integration`, and `tests/system` folders, respectively. Running the executable that is generated by the `tests/mainTest.cpp` file will verify that the compilation has worked correctly.

In order for the library to be compiled correctly, the gtest library needs to be installed to test the code. It can be built for either Windows or UNIX, and you can find detailed instructions for how to compile gtest here.

Once gtest is installed and available, you can go ahead and use one of the build scripts discussed below to compile the mesh reading library on either Windows or UNIX.

Windows

On Windows, we can either build the mesh reading library all into a single executable (library + tests), or into separate test code and either static or dynamic library. The build scripts to achieve that are given below. All instructions assume that you are using a developer PowerShell, which is automatically installed with Visual Studio.

> [!NOTE]
> In order to execute PowerShell scripts, you will have to allow to run remotely signed scrips first (see the first example). You may also need to unblock the PowerShell files first before using them. You will also need to have Visual Studio with the C++ development kit installed and run this scrupt inside a Developer PowerShell console. You may also need to activate 64-bit compilation, which may cause issues if the compilation tries to mix 32-bit and 64-bit binaries. If this sounds too complicated, simply install WSL and follow the (simplified) instructions for UNIX below.

Single executable (tests + library)

`.\buildAndRun.ps1`

Tests with separate static mesh reader library

`.\buildStaticAndRun.ps1`

Tests with separate dynamic mesh reader library

`.\buildDynamicAndRun.ps1`

UNIX (macOS, Ubuntu, etc.)

Similar to the instruction given on Windows, we can run the build bash script to check that the compilation and execution have worked correctly. These scripts assume that you have the CGNS and HDF5 library compiled and installed within your home directory, i.e. within `~/libs`.

Unlike Windows, we can run scripts directly but sometimes permissions may not be granted for the current user, in which case we can use the `chmod` command to grant permission to run a specific file as `chmod +x buildAndRun.sh`, for example. We can also grant permissions for all bash scripts using regular expressions with `chmod +x *.sh`.

To build the tests and library, open a terminal and type in one of the following commands depending on what you want to build

Single executable (tests + library)

`./buildAndRun.sh`

Tests with separate static mesh reader library

`./buildStaticAndRun.sh`

Tests with separate dynamic mesh reader library

`./buildDynamicAndRun.sh`

Usage

The following shows how to work with the different classes. This is a lightweight library and if you know how to create a vector, a matrix, and a linear algebra solver instance, you know all there is to work with this library.

Creating a matrix

The following example shows how to create a matrix and use it:

```
#include <iostream>
#include "linearAlgebraLib/linearAlgebraLib.hpp"

int main() {
  // create a matrix with 3 rows and 3 columns
  linearAlgebraLib::SparseMatrixCSR matrix(3, 3);

  // set value for the first row, and second column to 1.8
  matrix.set(1, 2, 1.8);

  // get the value for the first row and second column
  auto value = matrix.get(1, 2);

  // print the entire matrix to the console
  std::cout << matrix << std::endl;

  return 0;
}
```

Creating a vector

The following example shows how to create a vector and use it:

```
#include <iostream>
#include "linearAlgebraLib/linearAlgebraLib.hpp"

int main() {
  // create a vector with 3 entries
  linearAlgebraLib::Vector vector(3);

  // set data
  vector[0] = 1.2;
  vector[2] = -7.1;

  // get individual components
  auto v0 = vector[0];

  // calculate L2 norm of vector
  auto l2norm = vector.getL2Norm();

  // play around with arithmetic operations
  linearAlgebraLib::Vector v1(3);
  linearAlgebraLib::Vector v2(3);

  // set vectors if you like, skipped here ...

  auto addition = v1 + v2;
  auto subtraction = v1 - v2;
  auto dotProduct = v1.transpose() * v2;
  auto scalarMultiplication = 2.0 * v1;

  // print content of vector to the console
  std::cout << vector << std::endl;

  // create vector and matrix for matrix-vector product
  linearAlgebraLib::Vector vector(3);
  linearAlgebraLib::SparseMatrixCSR matrix(3, 3);

  // set vector and matrix if you like, skipped here ...

  auto resultVector = matrix * vector;

  return 0;
}
```

Creating a linear system of equations

The following example shows how to create a linear solver instance and use it:

```
#include <iostream>
#include "linearAlgebraLib/linearAlgebraLib.hpp"

int main() {
  // define how many cells are being used, i.e. mesh size
  unsigned numberOfCells = 1000;

  // create vector and matrix for linear algebra solver, we'll use the notation Ax=b here
  linearAlgebraLib::Vector b(numberOfCells);
  linearAlgebraLib::SparseMatrixCSR A(numberOfCells, numberOfCells);

  // set vector and matrix if you like, skipped here ...

  // create conjugate gradient linear solver
  linearAlgebraLib::ConjugateGradient solver(numberOfCells);

  // set coefficient matrix and right-hand side vector in linear system
  solver.setRightHandSide(b);
  solver.setCoefficientMatrix(A);

  // solve the linear system, using 100 iterations at most and a convergence threshold of 1e-12
  linearAlgebraLib::Vector x = solver.solve(100, 1e-12);

  // if we computed A * x now, this should result in b.
  auto b_calculated = A * x;

  // we can check now that b and b_calculated are the same ...

  return 0;
}
```

References

---

1. Matt Eding - Sparse Matrices ↩