FMDV Documentation

Release 1.0

Sivaram Ambikasaran, Shyam Sundar Sankaran

Jun 26, 2019

Contents:

CHAPTER 1

About FMDV:

FMDV can be used to solve for multicomponent diffusion velocities in a fast manner. It does so by leveraging the low-rank nature of matrices that arise in the intermediate steps of solving the Stefan-Maxwell equations. The methods that are utilized in this library are detailed in this article The code is written in C++ and features an easy-to-use interface, where the user provides the through a `solver` object which abstracts data in the inverse binary diffusivities matrix through a member function `getInverseDiffusionCoefficient(int i, int j)` which returns the entry at the i^{th} row and j^{th} column of the matrix.

CHAPTER 2

Doc Contents

2.1 Tutorial

For this tutorial, we are going to be using the `basic_example_usage/main.cpp` file that is listed under `examples/` since it demonstrates all the features of this library. We go over the main features and functions that may be of interest to a user on this page.

This code can be used to solve for multicomponent diffusion velocities quickly from the Stefan-Maxwell's equation for a single grid point

$$\nabla X_p = \overbrace{\sum_{k=1}^{N} \frac{X_p X_k}{\mathcal{D}_{pk}} \left(v_k - v_p\right)}^{\text{Difference in velocities}} + \underbrace{\sum_{k=1}^{N} \frac{X_p X_k}{\mathcal{D}_{pk}} \left(\frac{D_k^{(T)}}{Y_k} - \frac{D_p^{(T)}}{Y_p}\right) \frac{\nabla T}{\rho T}}_{\text{Temperature gradient (Soret effect)}} + \overbrace{\left(Y_p - X_p\right) \frac{\nabla P}{P}}^{\text{Pressure gradient}} + \underbrace{\frac{\rho}{P} \sum_{k=1}^{N} Y_p Y_k \left(f_p - f_k\right)}_{\text{Difference in body force}}$$

This would mean that several input parameters such as mole-fraction, mass-fraction, pressure etc. are needed as an input to the solver.

2.1.1 Attributes Needed By Solver

The following attributes need to be a `double` since they describe data at the single grid point:

```
// Density at the grid point:
density
// Temperature at the grid point:
temperature
// Temperature gradient at the grid point:
temperature_gradient
// Pressure at the grid point:
pressure
// Pressure gradient at the grid point:
pressure_gradient
```

The following attributes need to be an `Eigen::VectorXd` of length `N_species` since it contains the values for each of the species used in the solver:

```
// Molecular weights of the species:
molecular_mass;
// Diameters of the species:
diameters;

// Thermal diffusivities of the species:
thermal_diffusivities;
// Collision energies of the species:
collision_energies;
// Mole fraction of the species:
// NOTE: A correct setup should ensure that this vector sums up to 1
mole_fraction;
// Mole fraction gradient of the species:
// NOTE: A correct setup should ensure that this vector sums up to 0
mole_fraction_gradient;
// Mass fraction of the species:
// NOTE: A correct setup should ensure that this vector sums up to 1
mass_fraction;
// Body forces of the species:
body_forces;
```

Below, we go over how one can set the various parameters that are needed. The main solver object is a derived object of the main `FMDV` class. The various attributes needed for the solve along with the inverse diffusivity matrix is abstracted through this class. For the sake of the tutorial, we are calling this derived class `fast_solver`. In this file, we have set all concerned parameters through the constructor. Alternatively, the parameters of interest can also be set / modified by changing these public attributes. The inverse diffusivity matrix is abstracted through the `getInverseDiffusionCoefficient` function which returns the entry at the i^{th} row and j^{th} column of the matrix.

2.1.2 Declaring Derived Class of `FMDV`

In `main.cpp` we have:

```
class fast_solver : public FMDV
{
public:
    fast_solver(int N_species) : FMDV(N_species)
    {
        // Density at the grid point:
        density = double(rand()) / RAND_MAX;
        // Temperature at the grid point:
        temperature = double(rand()) / RAND_MAX;
        // Temperature gradient at the grid point:
        temperature_gradient = double(rand()) / RAND_MAX;
        // Pressure at the grid point:
        pressure = double(rand()) / RAND_MAX;
        // Pressure gradient at the grid point:
        pressure_gradient = double(rand()) / RAND_MAX;

        // Molecular weights:
        molecular_mass = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
        molecular_mass = molecular_mass / molecular_mass.maxCoeff();
```

(continues on next page)

(continued from previous page)

```
        // Diameters:
        diameters = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
        diameters = diameters / diameters.maxCoeff();

        // Thermal diffusivities:
        thermal_diffusivities = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
        // Collision energies:
        collision_energies = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
        // Mole fraction:
        mole_fraction  = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
        mole_fraction /= mole_fraction.sum();
        // Mole fraction gradient:
        mole_fraction_gradient = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
        // Ensuring that molefraction gradient sums up to zero:
        mole_fraction_gradient /= mole_fraction_gradient.sum();
        mole_fraction_gradient  = mole_fraction_gradient.array() - 1 / double(N_
↪species);

        // Mass fraction:
        mass_fraction  = molecular_mass.cwiseProduct(mole_fraction);
        mass_fraction /= mass_fraction.sum();
        // Body forces:
        // The idea here is that body forces are proportional to mass of the species:
        body_forces = double(rand()) / RAND_MAX * mass_fraction;
    }

    double getInverseDiffusionCoefficient(int i, int j)
    {
        // 1/D = K * P / T**1.5 * ((d_i+d_j))^2 * sqrt(m_i*m_j/(m_i+m_j))
        double dia_sum              = diameters(i) + diameters(j);
        double elastic_matrix_entry =   (pressure / (temperature * sqrt(temperature)))
                                    * dia_sum * dia_sum * sqrt(molecular_mass(i) *␣
↪molecular_mass(j)
                                    / (molecular_mass(i) + molecular_mass(j)));

        // This segment computes collision integral based on collision energies:
        if(collision_energies(i) != 0 && collision_energies(j) != 0)
        {
            static const double m[] = {6.8728271691,  9.4122316321,  7.7442359037,
                                    0.23424661229, 1.45337701568, 5.2269794238,
                                    9.7108519575,  0.46539437353, 0.00041908394781};

            double T  = temperature * sqrt(collision_energies(i) * collision_
↪energies(j));
            double Nr = m[0] + T * (m[1] + T * (m[2] + T *  m[3]));
            double Dr = m[4] + T * (m[5] + T * (m[6] + T * (m[7] + T * m[8])));

            return (Nr/Dr) * elastic_matrix_entry;
        }

        else
        {
            return elastic_matrix_entry;
        }
    }
};
```

Here, we have set the attributes needed using the constructor itself. However, we can alternatively also just declare the object with the associated `getInverseDiffusionCoefficient` and set the attributes later before calling the solve function:

```
class fast_solver : public FMDV
{
public:
    fast_solver(int N_species) : FMDV(N_species)
    {}

    double getInverseDiffusionCoefficient(int i, int j)
    {
        // 1/D = K * P / T**1.5 * ((d_i+d_j))^2 * sqrt(m_i*m_j/(m_i+m_j))
        double dia_sum              = diameters(i) + diameters(j);
        double elastic_matrix_entry =   (pressure / (temperature * sqrt(temperature)))
                                    * dia_sum * dia_sum * sqrt(molecular_mass(i) *␣
↪molecular_mass(j)
                                    / (molecular_mass(i) + molecular_mass(j)));

        // This segment computes collision integral based on collision energies:
        if(collision_energies(i) != 0 && collision_energies(j) != 0)
        {
            static const double m[] = {6.8728271691,  9.4122316321,  7.7442359037,
                                    0.23424661229, 1.45337701568, 5.2269794238,
                                    9.7108519575,  0.46539437353, 0.00041908394781};

            double T  = temperature * sqrt(collision_energies(i) * collision_
↪energies(j));
            double Nr = m[0] + T * (m[1] + T * (m[2] + T *  m[3]));
            double Dr = m[4] + T * (m[5] + T * (m[6] + T * (m[7] + T * m[8])));

            return (Nr/Dr) * elastic_matrix_entry;
        }

        else
        {
            return elastic_matrix_entry;
        }
    }
};

fast_solver FS(N);

// Density at the grid point:
FS.density = double(rand()) / RAND_MAX;
// Temperature at the grid point:
FS.temperature = double(rand()) / RAND_MAX;
// Temperature gradient at the grid point:
FS.temperature_gradient = double(rand()) / RAND_MAX;
// Pressure at the grid point:
FS.pressure = double(rand()) / RAND_MAX;
// Pressure gradient at the grid point:
FS.pressure_gradient = double(rand()) / RAND_MAX;

// Molecular weights:
FS.molecular_mass = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
FS.molecular_mass = FS.molecular_mass / FS.molecular_mass.maxCoeff();
```

(continues on next page)

(continued from previous page)

```
// Diameters:
FS.diameters = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
FS.diameters = FS.diameters / FS.diameters.maxCoeff();

// Thermal diffusivities:
FS.thermal_diffusivities = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
// Collision energies:
FS.collision_energies = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
// Mole fraction:
FS.mole_fraction  = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
FS.mole_fraction /= FS.mole_fraction.sum();
// Mole fraction gradient:
mole_fraction_gradient = (Eigen::VectorXd::Random(N_species)).cwiseAbs();
// Ensuring that molefraction gradient sums up to zero:
FS.mole_fraction_gradient /= FS.mole_fraction_gradient.sum();
FS.mole_fraction_gradient  = FS.mole_fraction_gradient.array() - 1 / double(N_
↪species);

// Mass fraction:
FS.mass_fraction  = FS.molecular_mass.cwiseProduct(FS.mole_fraction);
FS.mass_fraction /= FS.mass_fraction.sum();
// Body forces:
// The idea here is that body forces are proportional to mass of the species:
FS.body_forces = double(rand()) / RAND_MAX * FS.mass_fraction;
```

2.1.3 Getting the Species Velocities

Once the solver object has been set with the required attributes, a single function call is all that is needed to get the diffusion velocities:

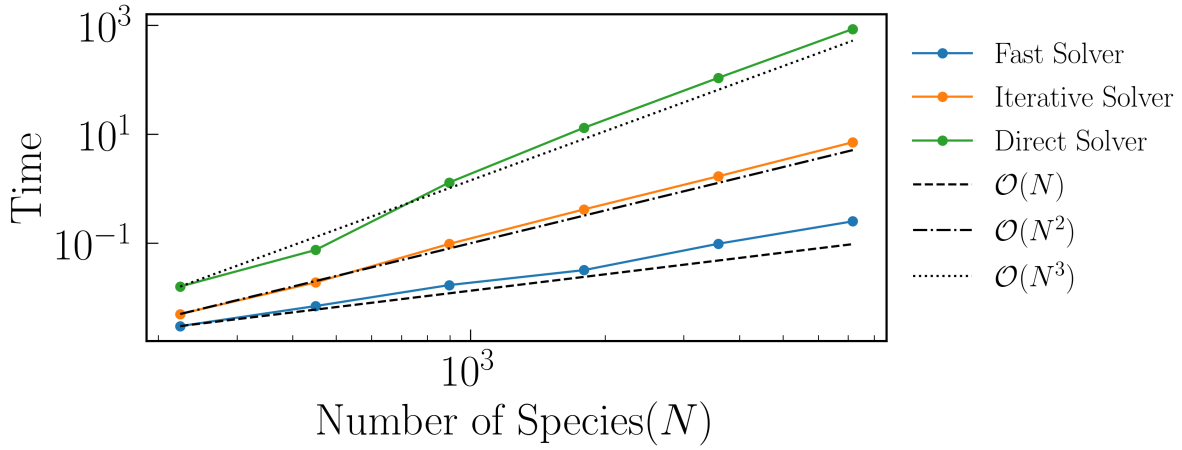
```
Eigen::VectorXd v_sp = FS.computeSpeciesVelocities(1e-14);
```

NOTE: Currently we are performing the solve for a single grid point. However, when solving over several grid points, it is quite common for the considered physical problem to have a common matrix V for all grid points. In such a case, further speedups can be acheived since the matrix doesn't have to be factorized at all grid points. Currently this interface doesn't support it in favour of simplicity, and would perform the factorization of the matrix for each grid point. If further speedups are required, this is worth looking into. We would be glad to aid in the development if needed :)

2.2 Benchmarks

All the following benchmarks have been carried out on an i7-8750H(with OpenMP enabled, this is 12 threads), with gcc-8.3 and Eigen version 3.3.7 Presented below are the results as obtained when running the 1D example provided under `examples`. 10^{-12} was given as the input tolerance for the fast method

N	Fast Method(s)	Iterative Method(s)	Direct Method(s)
224	0.00388074	0.00501709	0.0168638
450	0.00721523	0.019369	0.0758743
896	0.0171265	0.0971904	1.3037
1794	0.0326325	0.417033	13.1705
3586	0.097653	1.69336	108.792
7171	0.252024	7.14304	858.239


10^3
10^1
10^{-1}
Time
10^3
Number of Species(N)
Fast Solver
Iterative Solver
Direct Solver
$\mathcal{O}(N)$
$\mathcal{O}(N^2)$
$\mathcal{O}(N^3)$

CHAPTER 3

Other Links

Learn more about FMDV by visiting the

- Code Repository: http://github.com/sivaramambikasaran/Fast_Multi_Component_Diffusion
- Documentation: http://fmdv.rtfd.io