```
#include <complex>
#include <array>

#ifndef _CONSTHEADER
#define _CONSTHEADER

const std::complex<double> im(0., 1.);

const double PI = 3.14159265358979238463;
const double Pi = 3.14159265358979238463;
const double au2eV = 27.211;                    // E_h to eV
const double au2K = 3.15773e+5;                 // E_h/k_B
const double aucur2amp = 6.623617E-3;           // E_h*e/hbar to Amps
const double au2Vpm = 5.14220652E11;

const double etol = 0.001;

//Euler method (or Runge-Kutta 1st order)
const int sizeOfT0 = 2;
const std::array<std::array<double, sizeOfT0>, sizeOfT0> T0 =
{
    { 0.,   0.,
    0.,     1. }
};

//standard Runge-Kutta 4th order
const int sizeOfT1 = 5;
const std::array<std::array<double, sizeOfT1>, sizeOfT1> T1 =
{
    { 0.,       0.,         0.,         0.,         0.,
    1. / 2.,    1. / 2.,    0.,         0.,         0.,
    1. / 2.,    0.,         1. / 2.,    0.,         0.,
    1.,         0.,         0.,         1.,         0.,
    0.,         1. / 6.,    1. / 3.,    1. / 3.,    1. / 6. }
};

//Runge-Kutta-Fehlberg 5th order (RKF45)
const int sizeOfT2 = 8;
const std::array<std::array<double, sizeOfT2>, sizeOfT2> T2 =
{
    { 0.,       0.,             0.,             0.,             0.,
      0.,             0.,         0.,
    1. / 4.,    1. / 4.,        0.,             0.,             0.,
      0.,             0.,         0.,
    3. / 8.,    3. / 32.,       9. / 32.,       0.,             0.,
      0.,             0.,         0.,
    12. / 13.,  1932. / 2197.,  -7200. / 2197., 7296. / 2197,   0.,
      0.,             0.,         0.,
    1.,         439. / 216.,    -8.,            3680. / 513.,   -845. / 4104,
      0.,             0.,         0.,
    1. / 2.,    -8. / 27.,      2.,             -3544. / 2565., 1859. / 4104,
      -11. / 40.,     0.,         0.,
```

```
        0.,         16. / 135.,     0.,             6656. / 12825., 28561. /
          56430.,-9. / 50.,       2. / 55.,   0.,
        0.,         25. / 216.,     0.,             1408. / 2565.,  2197. / 4104.,
          -1. / 5.,       0.,         0. }
    };

    #endif
```

```
#include <iomanip>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <cstdlib>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <algorithm>
#include <random>
#include <conio.h>
#include <tchar.h>
#include <chrono>
#include <complex>
#include <omp.h>
#include <time.h>
```

```
#include "stdSetHeader.h"

#include "ConstHeader.h"
#include "memory.h"

#include <sys/stat.h>

using namespace std;

void invert(complex<double> **, complex<double> **, int );
complex<double> * ctof(complex<double> **, int , int );
void ftoc(complex<double> *, complex<double> **, int , int );

extern "C" void dsyev_(char*, char*, int *n, double *a, int *, double *, double *, 
   int *, int *);
```

```
#include "stdafx.h"
#include "solver.h"

double
tolerance = 4e-8, //1e-9 is bottom

Vini = 0.36e-3,                                 //unit: L
Vcyc = 6.9e-3,                                  //unit: L
Vsty = 0.8e-3,                                  //unit: L

I0 = 0.25*0.0512e-3,                            //unit: mol. Amount of initiator
  molecules, (s-BuLi)4 [CH3CHLiCH2CH3]4, tetramers, in 0.36 mL
                                                //in total volume of 0.960 mL of
                    initiator solution for long exp!, 0.0533 M

M0 = 6.98e-3,                                   //unit: mol. Initial amount of
  monomers, Styrene, in total (cyc + stock) volume 7.26 mL = 0.00726 L

ki = 1000.,                                     //unit: M¯¹min¯¹. Rate constant of
  styrene-butadiene initiation

kp = 1.,                                        //unit: M¯¹min¯¹. Rate constant of
  styrene propagation in cyclohexane
kta = 100.,
ktd = 0.2,

ka = 0.,                                        //ignoring this process
kd = 5., //0.05, //0.0001                       //

dt = 0.01,
t,                                              //time variable, min
t_wait = 2.*120.,                               //additional waiting time, min
time_of_initiator_additions,                    //min
&initiator_addition_time = time_of_initiator_additions,
t_max;                                          //min

string
experimental_data_profile_type,
&initiator_addition_profile_name = experimental_data_profile_type,
experimental_data_filename,
fileext = "",

parA = "P0066.txt",
parB = "P12272017.txt",
parC = "P01102018CUS.txt",
parD = "PMS50.txt",
parE = "P10042017.txt",
parF = "P03042019.txt",

par1 = "ConstCONT01052018.txt",
par2 = "BellCONT01052018.txt",
par3 = "ExpCONT01052018.txt",
```

```
    par4 = "LinCONT01052018.txt",
    par5 = "Exp2halfCONT01052018.txt",
    par6 = "LinDownCONT01052018.txt",


    parameter_file_name = parF;

    int
    number_of_initiator_additions = 20;

    bool
    run_single_case = true,
    dont_solve_equations = false,
    dont_record_model_distributions = false,
    dont_record_experimental_data = false,
    dont_record_line_dispersity = false,
    dont_record_time_dependant_concentrations = false,
    show_evolution = true,
    printDynamics = true,
    base_not_custom = true,

    run_forward = true;

    int
    N_print_total = 100,                        //number of points to print
    N_rescale_total = 200,                      //number of points to print
    h_max = I0,
    N_print_period,
    N_rescale_period,

    iN_print_total = 100, //*dtl
    iN_print_period,


    max_chain_length = 1.0e3, //3.0e3
    max_number_of_rows = 3e4;

    double
    z_rate_cuttoff_value = 800.0e3;

    ofstream
    outSSA,
    outMFI,
    Iout;

    std::streamsize precision_original = std::cout.precision();


    double** readDataIntoArray(string _fileName, int max_number_of_rows, int*
      number_of_rows, int number_of_columns)
    {
        const char* _fileName_char = _fileName.c_str();
```

```
        struct stat info;
        if (stat(_fileName_char, &info) != 0) {
            std::cerr << "ERROR: Cannot access file: " << _fileName << "\n";
            return NULL;
        }

        double **_DataArray = (double**)calloc(max_number_of_rows, sizeof(double*));

        for (int i = 0; i < max_number_of_rows; i++)
            _DataArray[i] = (double*)calloc(number_of_columns, sizeof(double));

        ifstream input;
        input.open(_fileName);

        string _line; istringstream _istringstream;

        *number_of_rows = 1;

        bool modeErase = false;

        while (!input.eof()) {
            getline(input, _line);

            _istringstream.str(_line);
            _istringstream >> _DataArray[*number_of_rows][0] >> _DataArray
              [*number_of_rows][1];
            _istringstream.clear();

            if (!modeErase && ((!_DataArray[*number_of_rows][0] > 0.)
                || (!_DataArray[*number_of_rows][0] > 0.)))
                modeErase = true;

            if (_line == "" || modeErase)
                continue;

            if (_DataArray[*number_of_rows][0] <= 0.) _DataArray[*number_of_rows][0]
              = 0.0;
            if (_DataArray[*number_of_rows][1] <= 0.) _DataArray[*number_of_rows][1]
              = 0.0;

            (*number_of_rows)++;
        }

        input.close();


        _DataArray[0][0] = *number_of_rows;

        double MW = 104.15;

        for (int i = 1; i < _DataArray[0][0]; i++) {
```

```
            _DataArray[i][0] = pow(10, _DataArray[i][0]);
            _DataArray[i][1] /= _DataArray[i][0];

            _DataArray[i][0] /= MW;
        }

        double
            integral = 0.;

        for (int i = 1; i < _DataArray[0][0]; i++)
            integral += _DataArray[i][0] * _DataArray[i][1];

        for (int i = 1; i < _DataArray[0][0]; i++)
            _DataArray[i][1] /= integral / (M0 / I0);


        //double
        //  total_m1_before = 0.,
        //  total_m2_before = 0.;
        //
        //ofstream before("before.dat");
        //
        //for (int i = 1; i < _DataArray[0][0]; i++){
        //  total_m1_before += _DataArray[i][1];
        //  total_m2_before += _DataArray[i][0] * _DataArray[i][1];
        //
        //  before << _DataArray[i][0] << "\t" << _DataArray[i][0] * _DataArray[i][1]
           << "\t" << total_m1_before << "\t" << _DataArray[i][1] << "\t" << endl;
        //}
        //
        //before.close();


        double **_RefinedDataArray = (double**)calloc(max_chain_length, sizeof
          (double*));

        for (int k = 0; k < max_chain_length; k++)
            _RefinedDataArray[k] = (double*)calloc(2, sizeof(double));


        //redescritize uniformly
        for (int k = 0, i = 1; k < max_chain_length; k++) {
            _RefinedDataArray[k][0] = k;

            while (_DataArray[i][0] < k && i < _DataArray[0][0]) {
                i++;
                _RefinedDataArray[k][1] += _DataArray[i][1];
            }
        }


        for (int i = 0; i < max_number_of_rows; i++)
```

```
            std::free(_DataArray[i]);

        std::free(_DataArray);


        //relaxation
        std::uniform_real_distribution<double> distribution(0., 1.);
        std::mt19937 rng;
        rng.seed(std::random_device{}());
        for (int i = 0; i < 20000; i++) { distribution(rng); }

        int
            number_relaxation_P = 4e4,
            number_relaxation_kP = 4e6,
            relaxation_kP_cutoff = 1e2,
            relaxation_kP_power = 5;

        for (int j = 0, k = 0; j < number_relaxation_P; j++) {
            k = max(int((max_chain_length - 1) * distribution(rng)), 1);

            double
                shift = _RefinedDataArray[k][1] - (_RefinedDataArray[k - 1][1] +
                  _RefinedDataArray[k][1] + _RefinedDataArray[k + 1][1]) / 3.;

            if (k > 1) _RefinedDataArray[k - 1][1] += shift / 2.;
            _RefinedDataArray[k][1] -= shift;
            _RefinedDataArray[k + 1][1] += shift / 2.;

        }


        for (int j = 0, k = 0; j < number_relaxation_kP; j++) {
            k = max(int((max_chain_length - 1) * distribution(rng)), 1);

            if (k < relaxation_kP_cutoff && (pow(k, relaxation_kP_power) / pow
              (relaxation_kP_cutoff, relaxation_kP_power) < distribution(rng)))
                continue;

            double
                shift = _RefinedDataArray[k][0] * _RefinedDataArray[k][1] -
                  (_RefinedDataArray[k - 1][0] * _RefinedDataArray[k - 1][1] +
                  _RefinedDataArray[k][0] * _RefinedDataArray[k][1] +
                  _RefinedDataArray[k + 1][0] * _RefinedDataArray[k + 1][1]) / 3.;

            if (k > 1) _RefinedDataArray[k - 1][1] += (1. / _RefinedDataArray[k - 1]
              [0]) * shift / 2.;
            _RefinedDataArray[k][1] -= (1. / _RefinedDataArray[k][0]) * shift;
            _RefinedDataArray[k + 1][1] += (1. / _RefinedDataArray[k + 1][0]) *
              shift / 2.;

        }
```

```
        _RefinedDataArray[0][0] = max_chain_length;


        //double
        //  total_m1_after = 0.,
        //  total_m2_after = 0.;
        //
        //ofstream after("after.dat");
        //
        //for (int k = 0, i = 1; k < max_chain_length; k++){
        //  total_m1_after += _RefinedDataArray[k][1];
        //  total_m2_after += _RefinedDataArray[k][0] * _RefinedDataArray[k][1];
        //
        //  after << _RefinedDataArray[k][0] << "\t" << _RefinedDataArray[k][0] *
          _RefinedDataArray[k][1] << "\t" << total_m1_after << "\t" <<
          _RefinedDataArray[k][1] << "\t" << _RefinedDataArray[k][1] << endl;
        //}
        //
        //after.close();
        //
        //
        ////check quality
        //std::cout << 100. * abs(total_m1_after - total_m1_before) / total_m1_before
           << "%\t" << 100. * abs(total_m2_after - total_m2_before) / total_m2_before
           << "%" << endl;


        return _RefinedDataArray;
    }

    double** allocateModelArray(int number_of_rows, int number_of_columns)
    {
        double **_DataArray = (double**)calloc(number_of_rows, sizeof(double*));

        for (int i = 0; i < number_of_rows; i++) {
            _DataArray[i] = (double*)calloc(number_of_columns, sizeof(double));
        }

        return _DataArray;
    }


    double compareDataVertical(double** ExpData, double** ModelData)
    {
        int
            countExpData = ExpData[0][0];

        double
            measure = 0.;

        for (int i = 1; i < countExpData; i++) {
            double
```

```
            currentY_ExpDataValue = ExpData[i][0] * ExpData[i][1],
            currentY_ModelDataValue = ExpData[i][0] * ModelData[int(ExpData[i]
              [0])][1];

        measure += pow(currentY_ExpDataValue - currentY_ModelDataValue, 2);
    }

    return sqrt(measure / (countExpData * (countExpData - 1)));
}

double compareData(double** ExpData, double** ModelData)
{
    int
        countData = ExpData[0][0];

    double
        measure = 0.;

    for (int i = 1; i < countData; i++) {
        double
            currentY_ModelDataValue = ExpData[i][0] * ModelData[int(ExpData[i]
              [0])][1],
            min_distance = 1e6,
            scaling_factor = 1e-3;

        int
            shift = 100,
            min_position = i;

        for (int j = max(0, i - shift); j < min(countData, i + shift); j++) {
            double
                min_distance_temp = sqrt(pow(scaling_factor * (i - j), 2) + pow
                  (currentY_ModelDataValue - ExpData[j][0] * ExpData[j][1], 2));

            if (min_distance_temp < min_distance)
                min_position = j;

            min_distance = min(min_distance, min_distance_temp);
        }

        measure += pow(min_distance, 2);
    }

    return sqrt(measure / (countData * (countData - 1)));
}


double get_initiator_addition_amount(string _name, int _total, int _current) {

    double
        _amount,
        t = double(_current) / double(_total),
```

```
            tmax = double(_total) / double(_total + 1);

        if (_name == "Constant") //any
            _amount = I0 / double(_total);

        else if (_name == "Linear") //20
            _amount = 0.0952385 * I0 * double(_current + 1) / double(_total);

        else if (_name == "Exponential") //20
            _amount = 0.000350827 * I0 * exp(6.7 * double(_current + 1) / double
              (_total));

        else if (_name == "Bell") //20
            _amount = 0.1167525 * I0 * exp(-17. * pow(double(_current + 1) / double
              (_total) - 0.5, 2));

        else if (_name == "Lineardown") //20
            _amount = 1.001107 * ((0.09706 - 0.09919) * I0 + 0.09919 * I0 * double
              (_total - _current) / double(_total));

        else if (_name == "Exp2half") //20
            if (_current <= 18)
                _amount = 0.986008 * I0 / (2. * 18.);
            else
                _amount = 0.986008 * 0.021 * (I0 / 2.) * exp(8.71 * double(_current -
                   18 + 1) / double(_total));

        else if (_name == "Sin") //any
            _amount = (2. / double(_total)) * I0 * pow(sin(Pi * t + Pi / 2), 2);

        else if (_name == "Circle") //50
            _amount = (2.55412 / double(_total)) * I0 * pow(t * (1. - t), 0.5);

        else if (_name == "UpCircle") //50
            _amount = (1.02 * 8.10626 / double(_total + 1)) * I0 * (0.5 - pow(t *
              (tmax - t), 0.5));

        else if (_name == "Triangle") //50
            if (_current <= _total / 2)
                _amount = 0.075534 * I0 * double(_current + 1) / double(_total);
            else
                _amount = 0.078755 * I0 * (-0.02147393 + double(_total - _current +
                  1) / double(_total));

        else if (_name == "UpTriangle") //50
            if (_current <= _total / 2)
                _amount = 0.07839 * I0 * (0.5 - double(_current) / double(_total));
            else
                _amount = 0.08173 * I0 * (0.5 - double(_total - _current) / double
                  (_total));

        else if (_name == "Trapezoid") //50
```

```
            _amount = 0.019417 * I0 * (0.5 + double(_total - _current + 1) / double
              (_total));

        else if (_name == "2xConstant") //50
            if (_current <= int(double(_total) / 3.))
                _amount = 1.51515 * I0 / double(_total);
            else if (_current <= int(2. * double(_total) / 3.))
                _amount = 0.;
            else
                _amount = 1.51515 * I0 / double(_total);

        else if (_name == "2xLinear") //50
            if (_current <= int(double(_total) / 2.))
                _amount = 0.076805 * I0 * double(_current + 1) / double(_total);
            else
                _amount = 0.076805 * I0 * double(_current - int(double(_total) /
                  2.)) / double(_total);

        else if (_name == "2xLineardown") //50
            if (_current <= int(double(_total) / 2.))
                _amount = 0.078755 * I0 * (-0.02147393 + double(_total - _current + 2
                   - int(double(_total) / 2.)) / double(_total));
            else
                _amount = 0.078755 * I0 * (-0.02147393 + double(_total - _current +
                  1) / double(_total));

        else if (_name == "ExpDown") //50
            _amount = 0.3853 * 0.000350827 * I0 * exp(6.7 * double(_total - _current
              + 1) / double(_total));

        else {
            //std::cerr << "No profile <" << _name << "> found..." << endl
            //  << "Press Enter to continue";

            //std::cin.get();

            return 0.;
        }

        return _amount;
    }


    double
    eps = 1e-8;

    const int
    number_of_parameters = 6;


    void run_direct_solution_of_equatuions_batches()
    {
```

```
        auto F = [](solverclass *solver, bool forward_not_reverse, bool
          printDynamics)->double {

            int exp_data_rows_count = 0;

            double** ExpData = readDataIntoArray(experimental_data_filename,
              max_number_of_rows, &exp_data_rows_count, 2);

            ofstream outTemp;

            outTemp.open("outMFI" + fileext + ".out");
            outTemp << "time" << "\t" << "M" << "\t" << "It" << "\t" << "I";
            outTemp << 0 << "\t" << M0 << "\t" << 0 << "\t" << 0 << endl;
            outTemp.close();

            struct {
                void operator()(solverclass *solver, double _time, int
                  *N_print_current, bool _output_distribution = false, double
                  **ExpData = NULL, double **ModelData = NULL) {
                    *N_print_current = 0;

                    std::cout << (*solver).i_end << "-" << trunc(_time) << "min";
                    std::cout << "," << (*solver).indNt[0] << "," << trunc(100.*
                      ((*solver).M) / M0) << "%* ";

                    int NumberOfMoments = 4;

                    double *moments = (double *)calloc(NumberOfMoments, sizeof
                      (double));

                    for (int i_index = 0; i_index < (*solver).z_max_states; i_index+
                      +)
                        for (int i_moment = 0; i_moment < NumberOfMoments; i_moment+
                         +)
                            moments[i_moment] += ((*solver).N[i_index] / I0) * double
                         (pow(i_index, i_moment));


                    ofstream _ofstream;

                    if (_output_distribution) {
                        if (!dont_record_model_distributions) {
                            _ofstream.open("ModelDistributions/outDistributionMFI" +
                         fileext + to_string(int(trunc((*solver).time))) + ".out");

                            if (ModelData != NULL) {
                                ModelData[0][0] = max_chain_length;
                                ModelData[0][1] = (*solver).I;
                            }

                            for (int k = 1; k < max_chain_length; k++) {
                                if (ModelData != NULL) {
```

```
                            ModelData[k][0] = k;
                            ModelData[k][1] = ((*solver).N[k]) / I0;
                        }

                        _ofstream << k << "\t" << (k * (((*solver).N[k]) /
                    I0)) << "\t" << (((*solver).N[k]) / I0) << "\n";
                    }

                    _ofstream << endl;

                    _ofstream.close();
                }

                if (!dont_record_experimental_data) {
                    _ofstream.open("DataDistributions/outData" + fileext +
                    ".out");

                    for (int k = 1; k < ExpData[0][0]; k++)
                        _ofstream << ExpData[k][0] << "\t" << ExpData[k][0] *
                     ExpData[k][1] << "\t" << ExpData[k][1] << endl;

                    _ofstream.close();
                }

                if (!dont_record_line_dispersity) {
                    _ofstream.open("outLine.out", std::ofstream::out |
                    std::ofstream::app);

                    _ofstream << initiator_addition_profile_name << "\t"
                        << int(time_of_initiator_additions) << "\t"
                        << (moments[0]) << "\t"
                        << (moments[1]) << "\t"
                        << (moments[2]) << "\t"
                        << ((moments[1] / moments[0]) * 104.15) << "\t"
                        << (moments[2] / (moments[1] * moments[1]))
                        << endl;

                    _ofstream.close();
                }
            }

            if (!dont_record_time_dependant_concentrations) {
                _ofstream.open("outMFI" + fileext + ".out",
                std::ofstream::out | std::ofstream::app);

                _ofstream << _time / 60. << "\t" << (*solver).M / I0 << "\t"
                << (*solver).It / I0 << "\t" << (*solver).I / I0 << endl;

                _ofstream.close();
            }

            std::free(moments);
```

```
        }
    }output;

    std::cout << "F";

    (*solver).kd = ((forward_not_reverse) ? 1. : -1.) * kd;
    (*solver).ki = ((forward_not_reverse) ? 1. : -1.) * ki;
    (*solver).kp = ((forward_not_reverse) ? 1. : -1.) * kp;
    (*solver).I = 0.;
    (*solver).It = 0.;
    (*solver).M = ((forward_not_reverse) ? 1. : 0.01) * M0;
    (*solver).M0 = M0;
    (*solver).z_rate_cuttoff_value = z_rate_cuttoff_value;
    (*solver).z_max_states = max_chain_length;
    (*solver).allocateWorkingArrays();
    (*solver).tolerance = tolerance;

    (*solver).time = 0.;

    (*solver).initiator_addition_time = time_of_initiator_additions;
    (*solver).initiator_addition_profile_name =
      experimental_data_profile_type;
    (*solver).base_not_custom = base_not_custom;

    for (int k = 1; k < (*solver).z_max_states; k++)
        (*solver).N[k] = 0.;

    if (!forward_not_reverse)
        if (bool take_from_experimental_file = true) {
            for (int k = 1; k < ExpData[0][0]; k++)
                (*solver).N[int(ExpData[k][0])] = I0 * max(0., ExpData[k]
                 [1]);
        }
        else {
            string _tline;
            istringstream _tistringstream;
            ifstream tempModelDistribution;

            double
                _ti,
                _tkP,
                _tP;

            tempModelDistribution.open("ModelDistributions/
              outDistributionMFI" + fileext + "95_" + ".out");

            if (!forward_not_reverse) {
                while (!tempModelDistribution.eof()) {
                    getline(tempModelDistribution, _tline);

                    _tistringstream.str(_tline);
                    _tistringstream >> _ti >> _tkP >> _tP;
```

```
                    _tistringstream.clear();

                    for (int k = 1; k < ExpData[0][0]; k++)
                        (*solver).N[int(_ti)] = I0 * max(0., _tP);
                }
            }

            tempModelDistribution.close();
        }

        N_print_period = int(100. / dt) / N_print_total;
        N_rescale_period = int(t_max / dt) / N_rescale_total;
        iN_print_period = int(t_max / dt) / iN_print_total;

        int
            N_print_current = N_print_period - 1,
            N_rescale_current = N_rescale_period - 1,
            iN_print_current = iN_print_period - 1,
            current_addition = 0;

        t = -dt;

        double
            current_I = (*solver).I,
            current_t = 0.,
            current_M = (*solver).M;

        double
            current_factor = 0.035 + 1. / (M0 - (*solver).M),
            previous_factor = 0.;

        while (((t += dt) <= t_max) && (((*solver).M > (0.01 * M0)) || (!
          forward_not_reverse))) {
            double
                dtl = 0.01*dt,
                dtc = 0.;

            (*solver).time = t + dtc;

            while (dtc < dt) {

                auto add_initiator = [solver, dtl, &iN_print_current,
                &current_addition, forward_not_reverse, &current_I,
                &current_t](double time_interval)->void {
                    if ((((*solver).time) <= time_of_initiator_additions) ||
                (!forward_not_reverse)) {
                        double
                            current_addition_amount =
                (time_of_initiator_additions == 0.) ? I0 :
                            ((time_interval * number_of_initiator_additions /
                 (time_of_initiator_additions)) *
                get_initiator_addition_amount(experimental_data_profile_type,
```

```
 number_of_initiator_additions, current_addition));

       (*solver).It += ((forward_not_reverse) ? 1. : 0.) *
current_addition_amount;

       if ((++iN_print_current > iN_print_period - 1) &&
((time_of_initiator_additions != 0.) || (!
forward_not_reverse))) {
           Iout
               << ((!forward_not_reverse) ? t_max -
(*solver).time : (*solver).time)
               << "\t" << ((!forward_not_reverse) ? 0. :
current_addition_amount / time_interval)
               << "\t" << (*solver).I / I0
               << "\t" << -(I0 * (current_I - ((*solver).I /
 I0)) / (current_t - (t_max - (*solver).time)))
               << "\n";

           current_I = (*solver).I / I0;
           current_t = t_max - (*solver).time;

           iN_print_current = 0;
       }

       if ((((*solver).time) / ((time_of_initiator_additions
 == 0.) ? 1. : time_of_initiator_additions)) >= double
(current_addition + 1) / double
(number_of_initiator_additions))
           current_addition++;
   }
};

double
   _ddt = dt - dtc;

if (_ddt > dtl) {
   if (!dont_solve_equations)
(*solver).solve_RungeKuttaFehlberg5thOrder(&dtl, true);

   add_initiator(dtl);

   dtc += dtl;
}
else {
   if (!dont_solve_equations)
(*solver).solve_RungeKuttaFehlberg5thOrder(&_ddt, false);

   add_initiator(_ddt);

   dtc += _ddt;
}
```

```
                    (*solver).time = t + dtc;
                }

                Iout.flush();

                if (printDynamics && ++N_print_current > N_print_period - 1) {
                    output(solver, t, &N_print_current, false); //show_evolution
                }

            }

            cout << endl << endl << (*solver).I / I0 << "\t" << I0 / (*solver).I;

            double** ModelData = allocateModelArray((*solver).z_max_states, 2);

            output(solver, t_max, &N_print_current, true, ExpData, ModelData);

            double result = compareData(ExpData, ModelData);


            for (int i = 0; i < max_chain_length; i++)
                std::free(ExpData[i]);

            std::free(ExpData);

            for (int i = 0; i < max_chain_length; i++)
                std::free(ModelData[i]);

            std::free(ModelData);

            (*solver).deallocateWorkingArrays();

            std::cout << "\n\n"
                << setprecision(1)
                << "M:   " << 100.*(*solver).M / M0 << "%"
                << endl << setprecision(precision_original);

            return result;
        };

        if (run_single_case) {

            double G_measure = 0.;


            ifstream tempfile(parameter_file_name);
            string _line;
            istringstream _istringstream;

            while (!tempfile.eof()) {
                getline(tempfile, _line);
```

```
                string suffix;

                _istringstream.str(_line);
                _istringstream >> experimental_data_profile_type >>
                  time_of_initiator_additions >> suffix;
                _istringstream.clear();

                t_max = time_of_initiator_additions + t_wait;

                experimental_data_filename = "Data/" + experimental_data_profile_type
                   + "/" + to_string(int(time_of_initiator_additions)) + suffix +
                  "min.dat";

                fileext = "_" + experimental_data_profile_type + "_" + to_string(int
                  (time_of_initiator_additions)) + suffix + "min";

                std::cout << fileext << endl;
                std::cout << experimental_data_filename << "\n" << endl;

                Iout.open("Iout" + fileext + ".dat");

                solverclass solver;

                G_measure += F(&solver, run_forward, printDynamics);

                std::cout << (solver).I / I0 << endl;

                Iout.close();

            }

            tempfile.close();

            std::cin.get();

            std::cout << G_measure << endl;
        }
    }


    void main(int argc, char* argv[])
    {
        std::cout << "Rate: " << ((base_not_custom) ? "Base" : "Custom") << "\n" <<
          endl;

        std::cout << "Using file: " << parameter_file_name << "\n" << endl;

        run_direct_solution_of_equatuions_batches();
    }
```

```
#include "stdSetHeader.h"

#include "ConstHeader.h"
#include "memory.h"


#ifndef _SOLVERCLASS
#define _SOLVERCLASS

class solverclass
{
private:

public:
    static clock_t
        _time, _time_ref, _time_prev;

    static int
        i_begin, i_end,
        currentBt;

    static bool
        base_not_custom;

    static string
        initiator_addition_profile_name;

    static const int
        maxBt = 10,
        aux_dimension = maxBt,
        number_of_basepoints = 6;

    static double
        cutoffB,
        _beta,
        _local_etol,
        &tolerance,
        RK4C[maxBt][maxBt];

    static memory solver_memory;

    static memory::Sarray
        vIt, kIt,
        vI, kI,
        vM, kM, tt;

    static memory::Darray
        vNt, kNt,
        vN, kN;

    static memory::Sarray
        Nt,
```

```
        N;

    static memory::intSarray
        indNt;

    static double
        It,
        I,
        M,
        M0,

        time,

        initiator_addition_time,

        Ntotal,

        ki, kp, ka, kd, kta, ktd,

        z_rate_cuttoff_value,

        q[number_of_basepoints];

    static int
        z_max_states;


    solverclass(){

        cutoffB = pow(10, -12);//-9
        _beta = 0.840896;
        _local_etol = 0.;

        Ntotal = 0.;
        time = 0.;

        int _mode = 2;

        //set RK4C from Butcher tableau
        if (_mode == 0)
        {
            currentBt = 1;
            auto setRK4C = [](const array<array<double, sizeOfT0>, sizeOfT0> _T,
              const int _size){
                for (int _i = 1; _i < _size; _i++)
                {
                    for (int _j = 1; _j < _size; _j++)
                    {
                        double _value = 0.; try { _value = _T[_i][_j]; }
                        catch (...){};
                        RK4C[_i][_j] = _T[_i][_j];
                    }
```

```
                }
                return true;
            }; setRK4C(T0, sizeOfT0);
        }

        else if (_mode == 1)
        {
            currentBt = 4;
            auto setRK4C = [](const array<array<double, sizeOfT1>, sizeOfT1> _T,
              const int _size){
                for (int _i = 1; _i < _size; _i++)
                {
                    for (int _j = 1; _j < _size; _j++)
                    {
                        double _value = 0.; try { _value = _T[_i][_j]; }
                        catch (...){};
                        RK4C[_i][_j] = _T[_i][_j];
                    }
                }
                return true;
            }; setRK4C(T1, sizeOfT1);
        }

        else if (_mode == 2)
        {
            currentBt = 6;
            auto setRK4C = [](const array<array<double, sizeOfT2>, sizeOfT2> _T,
              const int _size){
                for (int _i = 1; _i < _size; _i++)
                {
                    for (int _j = 1; _j < _size; _j++)
                    {
                        double _value = 0.; try { _value = _T[_i][_j]; }
                        catch (...){};
                        RK4C[_i][_j] = _T[_i][_j];
                    }
                }
                return true;
            }; setRK4C(T2, sizeOfT2);
        }

    };


    void allocateWorkingArrays()
    {
        int _n = z_max_states + 1;

        vIt = solver_memory.allocate_Sarray(aux_dimension);
        vI = solver_memory.allocate_Sarray(aux_dimension);
        vM = solver_memory.allocate_Sarray(aux_dimension);
        vN = solver_memory.allocate_Darray(_n, aux_dimension);
```

```
        kIt = solver_memory.allocate_Sarray(aux_dimension);
        kI = solver_memory.allocate_Sarray(aux_dimension);
        kM = solver_memory.allocate_Sarray(aux_dimension);
        kN = solver_memory.allocate_Darray(_n, aux_dimension);

        N = solver_memory.allocate_Sarray(_n);

        indNt = solver_memory.allocate_int_Sarray(_n * _n);
    }

    void deallocateWorkingArrays()
    {
        int _n = z_max_states + 1;

        free(vIt);
        free(vI);
        free(vM);
        solver_memory.release_Darray(vN, _n);

        free(kIt);
        free(kI);
        free(kM);
        solver_memory.release_Darray(kN, _n);

        free(N);

        free(indNt);
    }


    static void move(int _rank, double *_dt, int _i_begin = i_begin, int _i_end =
       i_end)
    {
        double
            &_I = vI[_rank - 1],
            &_It = vIt[_rank - 1],
            &_M = vM[_rank - 1];

        auto _kp = [](int i)
            -> double{double _i = double(i);

        double
            _base = 160. + 10700. * (1. - sqrt(600. / (_i + 600.))) - 100. * exp
              (-0.0002 * pow(_i - 100, 2)),
            _fh = pow(500, 3) / (pow(500, 3) + pow(_i, 3)),
            _f = _fh * _base + (1. - _fh) * 0.9 * _base,
            _effective_rate = _f;

        double
            f_0min = 2.5* (0.5 * 1650. / 1992.156) * (2500. * (exp(-0.0002 * pow
              (_i - 160, 2))) + 7500. * (exp(-0.0005 * pow(_i - 83, 2))) + 700. *
```

```
                (exp(-0.0004 * pow(_i - 250, 2))) - 1365. + 20. * _i + 7000. * exp
              (-0.000065 * (_i * _i - 150. * _i + 150 * 150)));

        double
            _baseLin = 160. + 10700. * (1. - sqrt(600. / (_i + 600.))) - 100. *
              exp(-0.0002 * pow(_i - 100, 2)),
            f_Linear_20_min = 0.95 * 2.0 * (281. * exp(-0.003 * pow(_i - 20, 2))
              + _baseLin - 540. + 2200. * (exp(-0.0005 * pow(_i - 60, 2))) + 50.
              * (exp(-0.0007 * pow(_i - 40, 2)))),
            f_Linear_30_min = 0.91 * 1.67 * (60. * exp(-0.003 * pow(_i - 20, 2))
              + _baseLin - 355. + 250. * (exp(-0.0005 * pow(_i - 60, 2))) + 950.
              * (exp(-0.0009 * pow(_i - 40, 2))) + 100. * (exp(-0.0005 * pow(_i -
               105, 2)))),
            f_Linear_40_min = 0.95 * 1.42 * (150. * exp(-0.003 * pow(_i - 20, 2))
               + _baseLin - 245. + 650. * (exp(-0.0009 * pow(_i - 40, 2))) + 50.
              * (exp(-0.0005 * pow(_i - 105, 2)))),
            f_Linear_50_min = 0.98 * 1.15 * (80. * exp(-0.003 * pow(_i - 30, 2))
              + _baseLin - 120. + 225. * (exp(-0.0009 * pow(_i - 30, 2)))),
            f_Linear_60_min = _baseLin,
            f_Linear_80_min = 0.95 * _baseLin;

        if ((!base_not_custom) && (initiator_addition_profile_name == "Linear")){
            if (initiator_addition_time == 0.) _effective_rate = f_0min;
            if (initiator_addition_time == 20.) _effective_rate =
              f_Linear_20_min;
            if (initiator_addition_time == 30.) _effective_rate =
              f_Linear_30_min;
            if (initiator_addition_time == 40.) _effective_rate =
              f_Linear_40_min;
            if (initiator_addition_time == 50.) _effective_rate =
              f_Linear_50_min;
            if (initiator_addition_time == 60.) _effective_rate =
              f_Linear_60_min;
            if (initiator_addition_time == 80.) _effective_rate =
              f_Linear_80_min;
        }


        double
            _f_supp_Const = 10700. * (1. - sqrt(600. / (_i + 600.))),
            _f_Const = _fh * _f_supp_Const + (1. - _fh) * 0.9 * _f_supp_Const,

            f_Const_80_min = 265.
            + _f_Const
            - 90. * exp(-0.0005 * pow(_i - 50, 2))
            - 130. * exp(-0.00025 * pow(_i - 100, 2))
            - 375. * exp(-0.00005 * pow(_i - 200, 2)),

            f_Const_100_min = 245.
            + 1.1 * (_f_Const
            - 40. * exp(-0.0005 * pow(_i - 50, 2))
            - 130. * exp(-0.00025 * pow(_i - 100, 2))
```

```
        - 355. * exp(-0.00005 * pow(_i - 200, 2))
        - 100. * exp(-0.0001 * pow(_i - 300, 2))),

        f_Const_120_min = 272.6
        + 1.1 * (_f_Const
        - 70. * exp(-0.0005 * pow(_i - 50, 2))
        - 150. * exp(-0.00025 * pow(_i - 100, 2))
        - 410. * exp(-0.00005 * pow(_i - 200, 2))
        - 140. * exp(-0.0001 * pow(_i - 300, 2))
        - 180. * exp(-0.0001 * pow(_i - 400, 2)));


    if ((!base_not_custom) && (initiator_addition_profile_name ==
      "Constant")){
        if (initiator_addition_time == 0.) _effective_rate = f_0min;
        if (initiator_addition_time == 80.) _effective_rate = f_Const_80_min;
        if (initiator_addition_time == 100.) _effective_rate =
          f_Const_100_min;
        if (initiator_addition_time == 120.) _effective_rate =
          f_Const_120_min;
    }


    double
        bell_er = 160.
        + 10700. * (1. - sqrt(600. / (_i + 600.)))
        - 100. * exp(-0.0002 * pow(_i - 100, 2)),

        _f_Bell = _fh * bell_er + (1. - _fh) * 0.9 * bell_er,


        f_Bell_80_min = 0.91 * _f_Bell,
        f_Bell_100_min = 0.89 * _f_Bell,
        f_Bell_120_min = 0.84 * _f_Bell;


    if ((!base_not_custom) && (initiator_addition_profile_name == "Bell")){
        if (initiator_addition_time == 0.) _effective_rate = f_0min;
        if (initiator_addition_time == 80.) _effective_rate = f_Bell_80_min;
        if (initiator_addition_time == 100.) _effective_rate =
          f_Bell_100_min;
        if (initiator_addition_time == 120.) _effective_rate =
          f_Bell_120_min;
    }


    double
        expo_er = 160.
        + 10700. * (1. - sqrt(600. / (_i + 600.)))
        - 100. * exp(-0.0002 * pow(_i - 100, 2)),

        _f_Expo = _fh * expo_er + (1. - _fh) * 0.9 * expo_er,
```

```
        _f_Expo120 = _fh * expo_er + (1. - _fh) * 0.8 * expo_er,


        f_Exponential_80_min = 0.91 * _f_Expo,
        f_Exponential_100_min = 0.89 * _f_Expo,
        f_Exponential_120_min = 0.84 * _f_Expo;


    if ((!base_not_custom) && (initiator_addition_profile_name ==
      "Exponential")){
        if (initiator_addition_time == 0.) _effective_rate = f_0min;
        if (initiator_addition_time == 80.) _effective_rate =
          f_Exponential_80_min;
        if (initiator_addition_time == 100.) _effective_rate =
          f_Exponential_100_min;
        if (initiator_addition_time == 120.) _effective_rate =
          f_Exponential_120_min;
    }



    double
        _f_supp_E2h = 10700. * (1. - sqrt(600. / (_i + 600.))),
        _f_E2h = _fh * _f_supp_E2h + (1. - _fh) * 0.9 * _f_supp_E2h,

        f_e2h_58_min = 0.99 * (265.
        + _f_E2h
        - 90. * exp(-0.0005 * pow(_i - 50, 2))
        - 130. * exp(-0.00025 * pow(_i - 100, 2))
        - 375. * exp(-0.00005 * pow(_i - 200, 2)))
        - 20.
        + 40. * exp(-0.0005 * pow(_i - 40, 2)),

        f_e2h_86_min = 0.80 * (245.
        + 1.1 * (_f_E2h
        - 40. * exp(-0.0005 * pow(_i - 50, 2))
        - 130. * exp(-0.00025 * pow(_i - 100, 2))
        - 355. * exp(-0.00005 * pow(_i - 200, 2))
        - 100. * exp(-0.0001 * pow(_i - 300, 2))))
        + 20.
        + 60. * exp(-0.0005 * pow(_i - 60, 2)),

        f_e2h_115_min = 0.80 * (272.6
        + 1.1 * (_f_E2h
        - 70. * exp(-0.0005 * pow(_i - 50, 2))
        - 150. * exp(-0.00025 * pow(_i - 100, 2))
        - 410. * exp(-0.00005 * pow(_i - 200, 2))
        - 140. * exp(-0.0001 * pow(_i - 300, 2))
        - 180. * exp(-0.0001 * pow(_i - 400, 2))))
        + 20.
        + 80. * exp(-0.0005 * pow(_i - 80, 2));
```

```
            if ((!base_not_custom) && (initiator_addition_profile_name ==
              "Exp2half")){
                if (initiator_addition_time == 58.) _effective_rate = f_e2h_58_min;
                if (initiator_addition_time == 86.) _effective_rate = f_e2h_86_min;
                if (initiator_addition_time == 115.) _effective_rate = f_e2h_115_min;
            }

            return _effective_rate*1.00;
            };

            auto _N = [_rank](int i)
                -> double{return (vN[i][_rank - 1]); };


            if (ki > 0.){
                kIt[_rank] = -kd * _It;

                kI[_rank] = 4.* kd * _It -ki * _I * _M;

                kM[_rank] = -ki * _I * _M;

                for (int _i = 1; _i <= _i_end; _i++)
                    kM[_rank] += -kp * _kp(_i) * _N(_i) * _M;

                for (int _i = 1; _i <= _i_end; _i++)
                    kN[_i][_rank] = -kp * _kp(_i) * _N(_i) * _M + ((_i == 1) ? ki *
                      _I : kp * _kp(_i - 1) *  _N(_i - 1)) * _M;
            }
            else{
                kI[_rank] = -ki * _N(1) * _M;

                kM[_rank] = -ki * _N(1) * _M;

                for (int _i = 1; _i <= _i_end; _i++)
                    kM[_rank] += -kp * _kp(_i + 1) * _N(_i + 1) * _M;

                for (int _i = 1; _i <= _i_end; _i++)
                    kN[_i][_rank] = -kp * _kp(_i + 1) * _N(_i + 1) * _M + ((_i ==
                      1) ? ki * _N(1) : kp * _kp(_i) *  _N(_i)) * _M;
            }

            vIt[_rank] = max(0., evaluate_function_from_coefficients(vIt[0], kIt,
              _rank, _dt));
            vI[_rank] = max(0., evaluate_function_from_coefficients(vI[0], kI, _rank,
               _dt));
            vM[_rank] = max(0., evaluate_function_from_coefficients(vM[0], kM, _rank,
               _dt));

            for (int _i = 1; _i <= _i_end; _i++)
                vN[_i][_rank] = evaluate_function_from_coefficients(vN[_i][0], kN
                  [_i], _rank, _dt);
```

```
        }


        static bool optimize_dt_embedded(int _rank, double *_dt, int _i_begin =
          i_begin, int _i_end = i_end)
        {
            double _max_error = 0.;

            auto evaluate_error_from_function = [&_max_error, _rank, _dt]
              (memory::Sarray _Sp1, memory::Sarray _Sp2){
                _max_error = max(_max_error, abs(_Sp1[_rank] -
                  evaluate_function_from_coefficients(_Sp1[0], _Sp2, _rank + 1,
                  _dt)));
            };

            evaluate_error_from_function(vIt, kIt);
            evaluate_error_from_function(vI, kI);
            evaluate_error_from_function(vM, kM);

            for (int _i = 1; (_i <= _i_end); _i++)
                evaluate_error_from_function(vN[_i], kN[_i]);

            double _dt_local_copy = *_dt;

            if (_max_error != 0.)
                *_dt = max(pow(10, -6), min(pow(10, -3), _beta * (*_dt) * pow
                  (((_local_etol == 0.) ? etol : _local_etol) / _max_error,
                  (((_local_etol == 0.) ? etol : _local_etol) >= _max_error) ? 0.2 :
                  0.25)));

            return (_dt_local_copy != *_dt) ? true : false;
        }

        static double evaluate_function_from_coefficients(double _s, memory::Sarray
          _Sp, int _rp, double *_dtp)
        {
            for (int _k = 1; _k <= currentBt; _k++)
                _s += RK4C[_rp][_k] * _Sp[_k] * *_dtp;

            return _s;
        };


        static void copyIn(int _index, int _i_begin = i_begin, int _i_end = i_end)
        {
            vIt[_index] = It;
            vI[_index] = I;
            vM[_index] = M;

            for (int _i = 1; _i <= _i_end - 1; _i++)
                vN[_i][_index] = N[_i];
```

```
        }

        static void copyOut(int _index, int _i_begin = i_begin, int _i_end = i_end)
        {
            It = vIt[_index];
            I = vI[_index];
            M = vM[_index];

            for (int _i = 1; _i <= _i_end - 1; _i++)
                N[_i] = vN[_i][_index];

        }


        static void solve_RungeKuttaFehlberg5thOrder(double *_dt, bool optimize)
        {
            i_begin = 0;
            i_end = z_max_states - 1;

            estimate_boundaries();
            copyIn(0);

            for (int _rank = 1; _rank <= currentBt; _rank++)
                move(_rank, _dt);

            if (optimize) optimize_dt_embedded(currentBt - 1, _dt);
            copyOut(currentBt);
        }

        static void estimate_boundaries()
        {
            int
                _i,
                _step = 10,
                _shift_value = 25;

            auto shift = [_shift_value](){
                i_begin = max(0, i_begin - _shift_value);
                i_end = min(z_max_states - 1, i_end + _shift_value);
            };

            shift();

            for (_i = i_end - 1; ((_i >= i_begin) && (N[_i] < cutoffB)); _i -= _step)
              {} i_end = _i;

            shift();
        }

        ~solverclass(){};
    };
```

```
#endif
```

```
#include "stdafx.h"

double
solverclass::RK4C[maxBt][maxBt],
solverclass::cutoffB;

memory
solverclass::solver_memory;

clock_t
solverclass::_time,
solverclass::_time_ref,
solverclass::_time_prev;

int
solverclass::currentBt;

bool
solverclass::base_not_custom;

string
solverclass::initiator_addition_profile_name;

double
solverclass::_beta,
solverclass::_local_etol;

double&
solverclass::tolerance = _local_etol;

memory::intSarray
solverclass::indNt;

memory::Sarray
solverclass::vIt,
solverclass::vI,
solverclass::vM,
solverclass::kIt,
solverclass::kI,
solverclass::tt,
solverclass::kM;

memory::Darray
solverclass::kNt,
solverclass::vNt,
solverclass::kN,
solverclass::vN;

memory::Sarray
solverclass::Nt,
solverclass::N;
```

```
    double
    solverclass::time,
    solverclass::initiator_addition_time;

    double
    solverclass::It,
    solverclass::I,
    solverclass::M,
    solverclass::M0,
    solverclass::Ntotal;

    int
    solverclass::z_max_states;

    double
    solverclass::z_rate_cuttoff_value,
    solverclass::q[number_of_basepoints];

    int
    solverclass::i_begin,
    solverclass::i_end;

    double
    solverclass::ki,
    solverclass::kp,
    solverclass::ka,
    solverclass::kd,
    solverclass::kta,
    solverclass::ktd;
```

```
#ifndef _memory_D
#define _memory_D

using namespace std;

class memory
{
public:

    typedef complex<double> **** cQarray;
    typedef complex<double> *** cTarray;
    typedef complex<double> ** cDarray;
    typedef complex<double> *  cSarray;

    typedef double * Sarray;
    typedef double ** Darray;
    typedef double *** Tarray;
    typedef double **** Qarray;

    typedef int * intSarray;
    typedef int ** intDarray;
    typedef int *** intTarray;
    typedef int **** intQarray;


    //                                                                      ↵
      ************************************************************************ ↵
      ************************************************************//
    //                  DYNAMIC ALLOCATING SPACE FOR ARRAYS OF HIGH DIMENSIONS    ↵
                          //
    //                                                                      ↵
      ************************************************************************ ↵
      ************************************************************//

    //allocate space for the 4 Dimensional array and shift all pointers so we go  ↵
      1 to N instead of 0 to N-1
    memory::Qarray allocate_Qarray_shift(int a, int b, int c, int d)
    {
        int i, j, k;
        memory::Qarray s;

        s = (memory::Qarray)calloc(a, sizeof(memory::Tarray));
        --s;
        for (i = 1; i <= a; ++i){
            s[i] = (memory::Tarray)calloc(b, sizeof(memory::Darray));
            --s[i];
            for (j = 1; j <= b; ++j){
                s[i][j] = (memory::Darray)calloc(c, sizeof(memory::Sarray));
                --s[i][j];
                for (k = 1; k <= c; ++k){
                    s[i][j][k] = (memory::Sarray)calloc(d, sizeof(double));
```

```
                    --s[i][j][k];
                }
            }
        }
        return s;
    }
    //allocate space for the 4 Dimensional array of complex doubles and shift all
       pointers so we go 1 to N instead of 0 to N-1
    memory::cQarray allocate_complex_Qarray_shift(int a, int b, int c, int d)
    {
        int i, j, k;
        memory::cQarray s;

        s = (memory::cQarray)calloc(a, sizeof(memory::cTarray));
        --s;
        for (i = 1; i <= a; ++i){
            s[i] = (memory::cTarray)calloc(b, sizeof(memory::cDarray));
            --s[i];
            for (j = 1; j <= b; ++j){
                s[i][j] = (memory::cDarray)calloc(c, sizeof(memory::cSarray));
                --s[i][j];
                for (k = 1; k <= c; ++k){
                    s[i][j][k] = (memory::cSarray)calloc(d, sizeof
                     (complex<double>));
                    --s[i][j][k];
                }
            }
        }
        return s;
    }
    //allocate space for the 4 Dimensional array defined from 0 to N-1
    memory::Qarray allocate_Qarray(int a, int b, int c, int d)
    {
        int i, j, k;
        memory::Qarray s;

        s = (memory::Qarray)calloc(a, sizeof(memory::Tarray));
        for (i = 0; i < a; ++i){
            s[i] = (memory::Tarray)calloc(b, sizeof(memory::Darray));
            for (j = 0; j< b; ++j){
                s[i][j] = (memory::Darray)calloc(c, sizeof(memory::Sarray));
                for (k = 0; k< c; ++k){
                    s[i][j][k] = (memory::Sarray)calloc(d, sizeof(double));
                }
            }
        }
        return s;
    }
    //allocate space for the 4 Dimensional array of complex doubles defined from
      0 to N-1
    memory::cQarray allocate_complex_Qarray(int a, int b, int c, int d)
    {
```

```
        int i, j, k;
        memory::cQarray s;

        s = (memory::cQarray)calloc(a, sizeof(memory::cTarray));
        for (i = 0; i < a; ++i){
            s[i] = (memory::cTarray)calloc(b, sizeof(memory::cDarray));
            for (j = 0; j< b; ++j){
                s[i][j] = (memory::cDarray)calloc(c, sizeof(memory::cSarray));
                for (k = 0; k< c; ++k){
                    s[i][j][k] = (memory::cSarray)calloc(d, sizeof
                     (complex<double>));
                }
            }
        }
        return s;
    }

    memory::Tarray allocate_Tarray_shift(int a, int b, int c)
    {
        int i, j;
        memory::Tarray s;

        s = (memory::Tarray)calloc(a, sizeof(memory::Darray));
        --s;
        for (i = 1; i <= a; ++i){
            s[i] = (memory::Darray)calloc(b, sizeof(memory::Sarray));
            --s[i];
            for (j = 1; j <= b; ++j){
                s[i][j] = (memory::Sarray)calloc(c, sizeof(double));
                --s[i][j];
            }
        }
        return s;
    }

    memory::cTarray allocate_complex_Tarray_shift(int a, int b, int c)
    {
        int i, j;
        memory::cTarray s;

        s = (memory::cTarray)calloc(a, sizeof(memory::cDarray));
        --s;
        for (i = 1; i <= a; ++i){
            s[i] = (memory::cDarray)calloc(b, sizeof(memory::cSarray));
            --s[i];
            for (j = 1; j <= b; ++j){
                s[i][j] = (memory::cSarray)calloc(c, sizeof(complex<double>));
                --s[i][j];
            }
        }
        return s;
    }
```

```
memory::Tarray allocate_Tarray(int a, int b, int c)
{
    int i, j;
    memory::Tarray s;

    s = (memory::Tarray)calloc(a, sizeof(memory::Darray));
    for (i = 0; i < a; ++i){
        s[i] = (memory::Darray)calloc(b, sizeof(memory::Sarray));
        for (j = 0; j< b; ++j){
            s[i][j] = (memory::Sarray)calloc(c, sizeof(double));
        }
    }
    return s;
}

memory::cTarray allocate_complex_Tarray(int a, int b, int c)
{
    int i, j;
    memory::cTarray s;

    s = (memory::cTarray)calloc(a, sizeof(memory::cDarray));
    for (i = 0; i < a; ++i){
        s[i] = (memory::cDarray)calloc(b, sizeof(memory::cSarray));
        for (j = 0; j< b; ++j){
            s[i][j] = (memory::cSarray)calloc(c, sizeof(complex<double>));
        }
    }
    return s;
}

memory::Darray allocate_Darray_shift(int a, int b)
{
    int i;
    memory::Darray s;

    s = (memory::Darray)calloc(a, sizeof(memory::Sarray));
    --s;
    for (i = 1; i <= a; ++i){
        s[i] = (memory::Sarray)calloc(b, sizeof(double));
        --s[i];
    }
    return s;
}

memory::cDarray allocate_complex_Darray_shift(int a, int b)
{
    int i;
    memory::cDarray s;

    s = (memory::cDarray)calloc(a, sizeof(memory::cSarray));
    --s;
```

```
            for (i = 1; i <= a; ++i){
                s[i] = (memory::cSarray)calloc(b, sizeof(complex<double>));
                --s[i];
            }
            return s;
        }

        memory::Darray allocate_Darray(int a, int b)
        {
            int i;
            memory::Darray s;

            s = (memory::Darray)calloc(a, sizeof(memory::Sarray));
            for (i = 0; i < a; ++i){
                s[i] = (memory::Sarray)calloc(b, sizeof(double));
            }
            return s;
        }

        memory::cDarray allocate_complex_Darray(int a, int b)
        {
            int i;
            memory::cDarray s;

            s = (memory::cDarray)calloc(a, sizeof(memory::cSarray));
            for (i = 0; i < a; ++i){
                s[i] = (memory::cSarray)calloc(b, sizeof(complex<double>));
            }
            return s;
        }

        memory::Sarray allocate_Sarray_shift(int a)
        {

            memory::Sarray s;

            s = (memory::Sarray)calloc(a, sizeof(double));
            --s;
            return s;
        }

        memory::cSarray allocate_complex_Sarray_shift(int a)
        {

            memory::cSarray s;

            s = (memory::cSarray)calloc(a, sizeof(complex<double>));
            --s;
            return s;
        }

        memory::Sarray allocate_Sarray(int a)
```

```
        {

            memory::Sarray s;

            s = (memory::Sarray)calloc(a, sizeof(double));
            return s;
        }

        memory::cSarray allocate_complex_Sarray(int a)
        {

            memory::cSarray s;

            s = (memory::cSarray)calloc(a, sizeof(complex<double>));
            return s;
        }

        memory::intSarray allocate_int_Sarray(int a)
        {

            memory::intSarray s;

            s = (memory::intSarray)calloc(a, sizeof(int));
            return s;
        }

        memory::intDarray allocate_int_Darray(int a, int b)
        {
            int i;
            memory::intDarray s;

            s = (memory::intDarray)calloc(a, sizeof(memory::intSarray));
            for (i = 0; i < a; ++i){
                s[i] = (memory::intSarray)calloc(b, sizeof(int));
            }
            return s;
        }

        memory::intTarray allocate_int_Tarray(int a, int b, int c)
        {
            int i, j;
            memory::intTarray s;

            s = (memory::intTarray)calloc(a, sizeof(memory::intDarray));
            for (i = 0; i < a; ++i){
                s[i] = (memory::intDarray)calloc(b, sizeof(memory::intSarray));
                for (j = 0; j< b; ++j){
                    s[i][j] = (memory::intSarray)calloc(c, sizeof(int));
                }
            }
            return s;
        }
```

```
memory::intQarray allocate_int_Qarray(int a, int b, int c, int d)
{
    int i, j, k;
    memory::intQarray s;

    s = (memory::intQarray)calloc(a, sizeof(memory::intTarray));
    for (i = 0; i < a; ++i){
        s[i] = (memory::intTarray)calloc(b, sizeof(memory::intDarray));
        for (j = 0; j< b; ++j){
            s[i][j] = (memory::intDarray)calloc(c, sizeof
              (memory::intSarray));
            for (k = 0; k< c; ++k){
                s[i][j][k] = (memory::intSarray)calloc(d, sizeof(int));
            }
        }
    }
    return s;
}



//
  ************************************************************************
  ************************************************************//
//                  RELEASING MEMORY FOR MATRICIES  OF HIGH
  DIMENSIONS
   //
//
  ************************************************************************
  ************************************************************//



void memory::release_Qarray_shift(Qarray a, int n, int m, int o)
{
    int i, j, k;

    for (i = 1; i <= n; ++i){
        for (j = 1; j <= m; ++j){
            for (k = 1; k <= o; ++k){
                free(a[i][j][k] + 1);
            }
            free(a[i][j] + 1);
        }
        free(a[i] + 1);
    }
    free(a + 1);
}

void memory::release_Qarray(Qarray a, int n, int m, int o)
{
```

```
        int i, j, k;

        for (i = 0; i<n; ++i){
            for (j = 0; j<m; ++j){
                for (k = 0; k<o; ++k){
                    free(a[i][j][k]);
                }
                free(a[i][j]);
            }
            free(a[i]);
        }
        free(a);
    }

    void memory::release_cQarray_shift(cQarray a, int n, int m, int o)
    {
        int i, j, k;

        for (i = 1; i <= n; ++i){
            for (j = 1; j <= m; ++j){
                for (k = 1; k <= o; ++k){
                    free(a[i][j][k] + 1);
                }
                free(a[i][j] + 1);
            }
            free(a[i] + 1);
        }
        free(a + 1);
    }

    void memory::release_cQarray(cQarray a, int n, int m, int o)
    {
        int i, j, k;

        for (i = 0; i<n; ++i){
            for (j = 0; j<m; ++j){
                for (k = 0; k<o; ++k){
                    free(a[i][j][k]);
                }
                free(a[i][j]);
            }
            free(a[i]);
        }
        free(a);
    }


    void memory::release_Tarray_shift(Tarray a, int n, int m)
    {
        int i, j;

        for (i = 1; i <= n; ++i){
```

```
            for (j = 1; j <= m; ++j){
                free(a[i][j] + 1);
            }
            free(a[i] + 1);
        }
        free(a + 1);
    }

    void memory::release_cTarray_shift(cTarray a, int n, int m)
    {
        int i, j;

        for (i = 1; i <= n; ++i){
            for (j = 1; j <= m; ++j){
                free(a[i][j] + 1);
            }
            free(a[i] + 1);
        }
        free(a + 1);
    }

    void memory::release_Tarray(Tarray a, int n, int m)
    {
        int i, j;

        for (i = 0; i<n; ++i){
            for (j = 0; j<m; ++j){
                free(a[i][j]);
            }
            free(a[i]);
        }
        free(a);
    }

    void memory::release_cTarray(cTarray a, int n, int m)
    {
        int i, j;

        for (i = 0; i<n; ++i){
            for (j = 0; j<m; ++j){
                free(a[i][j]);
            }
            free(a[i]);
        }
        free(a);
    }

    //clear allocated space 2 dim

    void memory::release_Darray_shift(Darray a, int n)
    {
        int i;
```

```
    for (i = 1; i <= n; ++i){
        free(a[i] + 1);
    }
    free(a + 1);
}

void memory::release_cDarray_shift(cDarray a, int n)
{
    int i;

    for (i = 1; i <= n; ++i){
        free(a[i] + 1);
    }
    free(a + 1);
}

void memory::release_Darray(Darray a, int n)
{
    int i;

    for (i = 0; i<n; ++i){
        free(a[i]);
    }
    free(a);
}

void memory::release_cDarray(cDarray a, int n)
{
    int i;

    for (i = 0; i<n; ++i){
        free(a[i]);
    }
    free(a);
}

void memory::release_intDarray(intDarray a, int n)
{
    int i;

    for (i = 0; i<n; ++i){
        free(a[i]);
    }
    free(a);
}

void memory::release_intTarray(intTarray a, int n, int m)
{
    int i, j;

    for (i = 0; i<n; ++i){
```

```
                for (j = 0; j<m; ++j){
                    free(a[i][j]);
                }
                free(a[i]);
            }
            free(a);
        }

        void memory::release_intQarray(intQarray a, int n, int m, int o)
        {
            int i, j, k;

            for (i = 0; i<n; ++i){
                for (j = 0; j<m; ++j){
                    for (k = 0; k<o; ++k){
                        free(a[i][j][k]);
                    }
                    free(a[i][j]);
                }
                free(a[i]);
            }
            free(a);
        }


    private:

    };

    #endif
```