```
// ------------------------------------------------------------\
//
// Iterative PBE Solver
// --------------------
// Solve discrete Smoluchowski Eqn with in/outflow and coagulation
// described by constant, additive and multiplicative kernels (analytic)
// in addition to a range of more physically realistic kernels.
//
// A. J. Smith (ajs224@cam.ac.uk)
//
// V4.5
//
//------------------------------------------------------------/

// To run with inflow rate=outflow rate=1, const kernel, 16 outer loops and a max clust
er size of 2^10 use:
// time ./pbSolverIterative -alpha 1 -k constant -loops 16 -p 10
// To run with the additive kernel and defaul in/outflow rates use:
// time ./pbSolverIterative -k additive -loops 256 -p 20
// In order to achieve convergence with the additive (and multiplicative?) kernels lowe
r the inflow rate:
// time ./pbSolverIterative -alpha 0.1 -k additive -loops 256 -p 10
//time ./pbSolverIterative -alpha 0.1 -k multiplicative -loops 64 -p 16

#include <iostream>
#include <sstream>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <cstdlib>
#include <cmath>
//#include "mfa_functions.h"

using namespace std;
//using namesapce mfa;


namespace mfaAnalytic
{
  enum kernelTypes { constant, additive, multiplicative, continuum, freemolecular, kine
tic, shearlinear, shearnonlinear, settling, inertiasettling, berry, condensation, spmte
st};

  kernelTypes kernelType; // Read from command line with -k flag.  Default is constant

  const int noMoments=4; // Number of moments to compute
  const string dataDir="data/"; // Output directory

} //namespace mfaAnalytic


// Coagulation kernel header
double k(unsigned long i, unsigned long j);


// Coagulation kernel definition
double k(unsigned long i, unsigned long j)
{

  using namespace mfaAnalytic;

  switch (kernelType)
    {
```

```
    case continuum: // Brownian motion (continuum regime)
      return (pow(i,1e0/3e0)+pow(j,1e0/3e0))*(pow(i,-1e0/3e0)+pow(j,-1e0/3e0));
      break;
    case freemolecular: // Brownian motion (free molecular regime)
      return pow(pow(i,1e0/3e0)+pow(j,1e0/3e0),2e0)*pow(pow(i,-1e0/3e0)+pow(j,-1e0/3e0)
,1e0/2e0);
      break;
    case kinetic: // Based on kinetic theory
      return (pow(i,1e0/3e0)+pow(j,1e0/3e0))*pow(i*j,1e0/2e0)*pow(i+j,-3e0/2e0);
      break;
    case shearlinear: // Shear (linear velocity profile)
      return pow(pow(i,1e0/3e0)+pow(j,1e0/3e0),3e0);
      break;
    case shearnonlinear: // Shear (nonlinear velocity profile):
      return pow(pow(i,1e0/3e0)+pow(j,1e0/3e0),7e0/3e0);
      break;
    case settling: // Gravitational settling
      return pow(pow(i,1e0/3e0)+pow(j,1e0/3e0),2e0)*abs(pow(i,1e0/3e0)-pow(j,1e0/3e0));
      break;
    case inertiasettling: // Inertia and gravitational settling
      return pow(pow(i,1e0/3e0)+pow(j,1e0/3e0),2e0)*abs(pow(i,2e0/3e0)-pow(j,2e0/3e0));
      break;
    case berry: // Analytic approximation of Berry's kernel
      return pow(i-j,2e0)*pow(i+j,-1e0);
      break;
    case condensation: // Condensation and/or branched-chain polymerisation
      return (i+2)*(j+2);  // with constant c=2
      break;
    case additive:
      return i+j;
      break;
    case multiplicative:
      return i*j;
      break;
    case spmtest:
      return pow(i*j,1e0/3e0);
      break;
    default: // constant kernel
      return 1e0;
    }
}



int main(int argc, char *argv[])
{

  using namespace mfaAnalytic;

  unsigned int p=16; // Maximum cluster size N=2^p, default is 16 (over-ridden with the
 -p flag)

  // Sometimes we can improve the convergence by doing more than log2(N) iterations (ca
reful not to do too many though--div 0!)
  unsigned int outerItLoops=4; // Try increasing this to around 256 for non constant ke
rnels (override with -loops flag)

  kernelType=constant; // default kernel type

  string kernelName;
  double * moments = new double[noMoments];

  double alpha, beta; // In and outflow factors
```

```
  // Default values
  alpha=1e-1;
  beta=alpha;

  bool numberDensityRep=true;

  // Output blurb
  cout << endl;
  cout << "Iterative PBE Solver - A. J. Smith (ajs224@cam.ac.uk)" << endl;
  cout << endl;
  cout << "This code solves the discrete Smoluchowski equation with in/outflow" << endl
;
  cout << "and coagulation described by constant, additive and multiplicative kernels "
 << endl;
  cout << "(admitting analytic solutions) in addition to a range of more physically " <
< endl;
  cout << "realistic kernels (run with --help for additional information)." << endl;
  cout << endl;

  // Process command line arguments
  for (int i=1; i<argc; ++i)
    {
      if (strcmp(argv[i], "--help") == 0)
        {
          //cout << "This is the help message" << endl;
          cout << "Usage: "<< argv[0] << " <flags>" << endl << endl;
          cout << "where <flags> is one or more of:" << endl << endl;
          cout << "\t" << "-alpha" << "\t\t" << "inflow factor (default is 1/10)" << en
dl;
          cout << "\t" << "-beta" << "\t\t" << "outflow factor (=alpha if omitted)" <<
endl;
          cout << "\t" << "-p" << "\t\t" << "power p, appearing in the maximum cluster
size, N=2^p (default is 16)" << endl;
          cout << "\t" << "-loops" << "\t\t" << "Can sometimes improve the convergence
by doing more than log2(N) " << endl;
          cout << "\t\t\titerations (careful not to do too many though! -- default is 4
)" << endl;
          cout << "\t" << "-mass" << "\t\t" << "Solves the PBE in mass flow form for mo
re direct comparison with" << endl;
          cout << "\t\t\tstochastic algorithms (default is in terms of number density)"
 << endl;
          cout << "\t\t\tN.B. In this case we expect m0=1, otherwise we expect m1=1 (us
eful convergence check)" << endl;
          cout << "\t" << "-k <type>" << "\t" << "kernel type, where <type> is one of:"
 << endl << endl;
          cout << "\t\t" << "constant" << "\t\t" << "constant kernel (default)" << endl
;
          cout << "\t\t" << "additive" << "\t\t" << "additive" << endl;
          cout << "\t\t" << "multiplicative" << "\t\t" << "multiplicative" << endl;
          cout << "\t\t" << "continuum" << "\t\t" << "Brownian motion (continuum regime
)" << endl;
          cout << "\t\t" << "freemolecular" << "\t\t" << "Brownian motion (free molecul
ar regime)" << endl;
          cout << "\t\t" << "kinetic" << "\t\t\t" << "Based on kinetic theory" << endl;
          cout << "\t\t" << "shearlinear" << "\t\t" << "Shear (linear velocity profile)
" << endl;
          cout << "\t\t" << "shearnonlinear" << "\t\t" << "Shear (nonlinear velocity pr
ofile)" << endl;
          cout << "\t\t" << "settling" << "\t\t" << "Gravitational settling" << endl;
          cout << "\t\t" << "inertiasettling" << "\t\t" << "Inertia and gravitational s
ettling" << endl;
          cout << "\t\t" << "berry" << "\t\t\t" << "Analytic approximation of Berry's k
```

```
ernel" << endl;
          cout << "\t\t" << "condensation" << "\t\t" << "Condensation and/or branched-c
hain polymerisation" << endl;
          cout << "\t\t" << "spmtest" << "\t\t\t" << "Kernel used to test the Single Pa
rticle Method (SPM)" << endl;
          cout << endl;
          cout << "Examples:" << endl << endl;
          cout << "* To run with inflow rate=outflow rate=1, const kernel, 16 outer loo
ps and a max cluster size of 2^10 use:" << endl;
          cout << "\ttime "<< argv[0] << " -alpha 1 -loops 16 -p 10" << endl;
          cout << "* To run with the additive kernel and default in/outflow rates use:"
 << endl;
          cout << "\ttime "<< argv[0] << " -k additive -loops 256 -p 20" << endl;
          cout << "* In order to achieve convergence with more complicated kernels lowe
r the inflow rate:" << endl;
          cout << "\ttime "<< argv[0] << " -alpha 0.1 -k multiplicative -loops 64 -p 16
" << endl;
          cout << "\ttime "<< argv[0] << " -alpha 0.05 -k freemolecular -loops 64 -p 16
" << endl;
          cout << endl;
          return 0;
        }
      else if (strcmp(argv[i], "-alpha") == 0)
        {
          // Read inflow factor
          alpha = atof(argv[++i]); // default 1/10
          beta=alpha;
        }
      else if (strcmp(argv[i], "-beta") == 0)
        {
          // Read outflow factor
          // If omitted inflow=outflow rate
          beta = atof(argv[++i]); // default 2
        }
      else if (strcmp(argv[i], "-mass") == 0)
        {
          // Solves the equation in mass flow form
          numberDensityRep=false;
        }
      else if (strcmp(argv[i], "-k") == 0)
        {
          // read constants appearing in multiplicative kernel k(x,y)=c*x^a*y^b
          // c=argv[++i];
          // a=argv[++i];
          // b=argv[++i];

          // Just read one of the 3 basic kernel types with analytic solution for now
          char *kArg=argv[++i];
          if (strcmp(kArg, "additive") == 0)
            kernelType=additive;
          else if (strcmp(kArg, "multiplicative") == 0)
            kernelType=multiplicative;
          else if (strcmp(kArg, "continuum") == 0)
            kernelType=continuum;
          else if (strcmp(kArg, "freemolecular") == 0)
            kernelType=freemolecular;
          else if (strcmp(kArg, "kinetic") == 0)
            kernelType=kinetic;
          else if (strcmp(kArg, "shearlinear") == 0)
            kernelType=shearlinear;
          else if (strcmp(kArg, "shearnonlinear") == 0)
            kernelType=shearnonlinear;
          else if (strcmp(kArg, "settling") == 0)
```

```
            kernelType=settling;
          else if (strcmp(kArg, "inertiasettling") == 0)
            kernelType=inertiasettling;
          else if (strcmp(kArg, "berry") == 0)
            kernelType=berry;
          else if (strcmp(kArg, "condensation") == 0)
            kernelType=condensation;
          else if (strcmp(kArg, "spmtest") == 0)
            kernelType=spmtest;
          else
            kernelType=constant; // actually this is default anyway
        }
      else if (strcmp(argv[i], "-p") == 0)
        {
          // Read p, where the maximum cluster size, N=2^p
          p = atoi(argv[++i]); // default 16
        }
      else if (strcmp(argv[i], "-loops") == 0)
        {
          // Read number of outer convergence loops
          outerItLoops = atoi(argv[++i]); // default 4
        }
    }

  const unsigned long N=pow(2,p); // Max particle size, i.e., i<N in n_i. # of particle
s is sum_i n_i
  // I choose a power of 2, because for pure coagulations the cluster sizes double with
 each iteration
  // so we can't do more than log2(N)=p iterations before gelation occurs.
  const int L=outerItLoops*floor(log2(N)); // Number of iterations to perform

  // Find out which kernel type is selected
  switch (kernelType)
    {
    case continuum:
      kernelName="continuum";
      break;
    case freemolecular:
      kernelName="freemolecular";
      break;
    case kinetic:
      kernelName="kinetic";
      break;
    case shearlinear:
      kernelName="shearlinear";
      break;
    case shearnonlinear:
      kernelName="shearnonlinear";
      break;
    case settling:
      kernelName="settling";
      break;
    case inertiasettling:
      kernelName="inertiasettling";
      break;
    case berry:
      kernelName="berry";
      break;
    case condensation:
      kernelName="condensation";
      break;
      // Analytic kernels
    case additive:
```

```
      kernelName="additive";
      break;
    case multiplicative:
      kernelName="multiplicative";
      break;
    case spmtest:
      kernelName="spmtest";
      break;
    default: // constant kernel
      kernelName="constant";
    }

  // Set up output file streams
  ofstream outputFile;
  ofstream momentsFile;
  //ofstream diamsFile;

  string outputFileName=dataDir+kernelName+"_data_";
  string momentsFileName=dataDir+kernelName+"_moments_";
  //string diamsFileName=dataDir+"diameters_";
  string ext=".txt";

  string desc;
  stringstream out;

  // For simplicity let's index stuff from 1 - N instead of 0 - (N-1)
  double *n = new double[N+1];  // Allocate N ints and save ptr in n. Requires 2^{p+3-1
0} Mb, i.e., ~8Gb when p=20.
  double *nold = new double[N+1];
  //double K[N][N];
  double d, summa;

  double n_in;

  //vector<double> xDiam (N);

  // Initialise PSD to a delta delta_{i1}, i.e., mono-dispersed
  for (unsigned long i=1; i<=N; i++)
    {
      n[i] = 0e0;
    }
  n[1]=1e0;

  //out << N;
  out << "p" << p << "_alpha" << alpha << "_beta" << beta << "_loops" << outerItLoops;
  desc=out.str();

  outputFileName+=desc+ext;
  momentsFileName+=desc+ext;
  //diamsFileName+=desc+ext;

  outputFile.open(outputFileName.c_str(), ios::out);
  momentsFile.open(momentsFileName.c_str(), ios::out);
  //diamsFile.open(diamsFileName.c_str(), ios::out);

  cout << "Running iterative solver (";
  if(numberDensityRep)
    cout << "in terms of number density";
  else
    cout << "in massflow form";
  cout << ") with maximum particle size of " << N << " and " << L << " iterations." <<
endl << endl;
  kernelName[0]=toupper(kernelName[0]); // capitalise
```

```
  cout << kernelName << " kernel selected." << endl;
  cout << "Inflow rate (1/alpha):" << 1/alpha << endl;
  cout << "Outflow rate (1/beta):" << 1/beta << endl;
  cout << endl;

  cout.precision(10);
  cout.width(20);
  //cout.fill('0');
  //cout.setf(ios::showpos);
  cout.setf(ios::scientific);
  outputFile.precision(8);
  momentsFile.precision(8);

  // Output header
  cout << "Iter\tm0\t\t\tm1\t\t\tm2\t\t\tm3" << endl;

  int l=1;

  // Iterate L times
  while (l<=L)
    {

      // Let's compute the moments of the distribution
      for(int moment=0;moment<noMoments;moment++)
        {
          moments[moment]=0e0;
          for(unsigned long i=1;i<=N;i++)
            {
              moments[moment]+=pow(i,moment)*n[i];
            }
        }

      /*
      for(int i=0;i<N;i++)
        {
          cout << n[i] << " ";
        }
      cout << endl;

      system("sleep 0");
      */

      cout << l << "\t" << moments[0] << "\t" <<moments[1] << "\t" << moments[2] << "\t
" << moments[3] << endl;
      momentsFile << l << "\t" << moments[0] << "\t" <<moments[1] << "\t" << moments[2]
 << "\t" << moments[3] << endl;

      // Update old distribution to new distribution
      for(unsigned long i=1;i<=N;i++)
        {
          nold[i]=n[i];
        }

      for(unsigned long i=1;i<=N;i++) // Loop over N particle sizes
        {

          // Compute sums in numerator and denominator
          d=0e0;
          for(unsigned long j=1;j<=N;j++)
            {
              if(numberDensityRep)
                d+=k(i,j)*nold[j];
              else
```

```
                d+=k(i,j)*nold[j]/j;
              }

            summa=0e0;
            for(unsigned long j=1;j<=i-1;j++)
              {
                //summa+=K[i-j][j]*nold[i-j]*nold[j];
                if(numberDensityRep)
                  summa+=k(i-j,j)*nold[i-j]*nold[j];
                else
                  summa+=k(i-j,j)*nold[i-j]*nold[j]/j;
              }

            //cout <<"i="<<i<<", d="<<d<<", summa="<<summa<<endl;

            // Evaluate n_in (here delta_{i1}), i.e., clusters of size 1 flow into the do
main
            if(i==1)
              n_in=1e0;
            else
              n_in=0e0;

            if(numberDensityRep)
              summa*=0.5;

            // Iterate baby!
            //n[i]=(n_in/alpha+0.5*summa)/(1e0/beta+d);
            n[i]=(n_in/alpha+summa)/(1e0/beta+d);

            //n[i]=0.5*summa/d; // Pure coagulation

          }

        l++; // Update iteration counter

      }

    // Dump steady-state PSD
    for(unsigned long i=1;i<=N;i++) // Loop over N particle sizes
      {

        outputFile << i << "\t" << n[i] << endl;
      }

    // Close files
    outputFile.close();
    momentsFile.close();


    // Clean up memory like a good boy
    delete [] n;
    n = NULL;

    delete [] nold;
    nold = NULL;

    delete [] moments;
    moments = NULL;

    //delete [][] K;
    //K=NULL;
```

```
  return 0;
}
```