```
#include "rk4_solver.hh"
#include <iostream>
#include <iomanip>
#include <math.h>
#include <ctime>
#include <cstdlib>
#include <cstdio>
#include <fstream>
#include <string>
#include <omp.h>
#include <cfloat>

void rk4::find_square(double x, double y, bool nlim){
  bool y_geq = y>ylim[1];
  bool x_leq = x<xlim[1];
  location = ( y_geq &&  x_leq)*7
            +( y_geq && !x_leq)*8
            +(!y_geq &&  x_leq)*9
            +(!y_geq && !x_leq)*10;
  xlim_next[1] = nlim*( (  x_leq *0.5*(xlim[0]+xlim[1]))
                       +((!x_leq)*0.5*(xlim[1]+xlim[2])))
                + !nlim * xlim_next[1];
  xlim_next[0] = nlim*((!x_leq)?xlim[1]:xlim_next[0]) + !nlim * xlim_next[0];
  xlim_next[2] = nlim*( x_leq  ?xlim[1]:xlim_next[2]) + !nlim * xlim_next[2];
  ylim_next[1] = nlim*( (  y_geq *0.5*(ylim[1]+ylim[2]))
                       +((!y_geq)*0.5*(ylim[0]+ylim[1])))
                + !nlim * ylim_next[1];
  ylim_next[0] = nlim*(  y_geq ?ylim[1]:ylim_next[0]) + !nlim * ylim_next[0];
  ylim_next[2] = nlim*((!y_geq)?ylim[1]:ylim_next[2]) + !nlim * ylim_next[2];
}

inline void rk4::push_node(int i, double x_, double y_, double sint_, double cost_){
  // Every new pushed node is a value node.
  barnes_list.push_back(1);

  // Decided that every value node has only one "children": itself.
  barnes_list.push_back(1);
  barnes_list.push_back(x_);
  barnes_list.push_back(y_);
  barnes_list.push_back(sint_);
  barnes_list.push_back(cost_);
  barnes_list.push_back(i);
  barnes_list.push_back(-1);
  barnes_list.push_back(-1);
  barnes_list.push_back(-1);
  barnes_list.push_back(-1);
}


inline void rk4::init_lims(){
  double low_  = fmin(minx-0.01*(maxx-minx), miny-0.01*(maxy-miny));
  double high_ = fmax(maxx+0.01*(maxx-minx), maxy+0.01*(maxy-miny));
  xlim[0] = low_;
  xlim[1] = 0.5*(low_+high_);
  xlim[2] = high_;
  ylim[0] = low_;
  ylim[1] = 0.5*(low_+high_);
  ylim[2] = high_;
  for (int i=0;i<3;i++){
    xlim_next[i] = xlim[i];
    ylim_next[i] = ylim[i];
  }
}

void rk4::barnes_compute(int cidx_, int &i, double xi, double yi, double thi,
                         double &sumx, double &sumy, double &sumtheta, int &N_comp,
                         double lgth){
  double mx=barnes_list[cidx_+2], my=barnes_list[cidx_+3],
         msin=barnes_list[cidx_+4], mcos=barnes_list[cidx_+5];
  int val = (int) barnes_list[cidx_], nchd=(int) barnes_list[cidx_+1],
      pid = (int) barnes_list[cidx_+6], n1=(int) barnes_list[cidx_+7],
      n2=(int) barnes_list[cidx_+8], n3=(int) barnes_list[cidx_+9],
      n4=(int) barnes_list[cidx_+10];
```

```
  double norm2 = (xi-mx)*(xi-mx) + (yi-my)*(yi-my);
  double norm = sqrt(norm2);

  if (val==1){
    if (i != pid){
      sumx += ((mx-xi)/norm)*(AA+J*(cos(thi)*mcos+sin(thi)*msin)) - BB *((mx-xi)/nor
m2);
      sumy += ((my-yi)/norm)*(AA+J*(cos(thi)*mcos+sin(thi)*msin)) - BB *((my-yi)/nor
m2);
      sumtheta += (msin*cos(thi)-sin(thi)*mcos)/norm;
      N_comp += nchd;
    }
  }
  else{
    if ((lgth/norm)>barnes_theta){
      if (n1!=-1) barnes_compute(n1,i,xi,yi,thi,sumx,sumy,sumtheta,N_comp, lgth/2.);
      if (n2!=-1) barnes_compute(n2,i,xi,yi,thi,sumx,sumy,sumtheta,N_comp, lgth/2.);
      if (n3!=-1) barnes_compute(n3,i,xi,yi,thi,sumx,sumy,sumtheta,N_comp, lgth/2.);
      if (n4!=-1) barnes_compute(n4,i,xi,yi,thi,sumx,sumy,sumtheta,N_comp, lgth/2.);
    }
    else{
      sumx += nchd*(((mx-xi)/norm)*(AA+J*(cos(thi)*mcos+sin(thi)*msin)) - BB *((mx-x
i)/norm2));
      sumy += nchd*(((my-yi)/norm)*(AA+J*(cos(thi)*mcos+sin(thi)*msin)) - BB *((my-y
i)/norm2));
      sumtheta += nchd*(msin*cos(thi)-sin(thi)*mcos)/norm;
      N_comp += nchd;
    }
  }
}

void rk4::smart_compute_xx(double t_, double* x_, double* y_, double* theta_, double
* outputX, double* outputY, double* output_theta){

  // Create a fresh Barnes-Hut list
  barnes_list.clear();
  push_node(0, x_[0],y_[0],sin(theta_[0]),cos(theta_[0]));
  barnes_list[0] = 0;
  barnes_list[(y_[0]>0)?((x_[0]<0)?7:8):((x_[0]<0)?9:10)] = barnes_list.size();
  cidx = barnes_list.size();
  push_node(0, x_[0],y_[0],sin(theta_[0]),cos(theta_[0]));
  minx=DBL_MAX;
  maxx=-DBL_MAX;
  miny=DBL_MAX;
  maxy=-DBL_MAX;
  for (int i=0; i<N; i++){
    minx = (x_[i]<minx)?x_[i]:minx;
    maxx = (x_[i]>maxx)?x_[i]:maxx;
    miny = (y_[i]<miny)?y_[i]:miny;
    maxy = (y_[i]>maxy)?y_[i]:maxy;
  }
  // Initialize xlim, ylim, xlim_next, ylim_next
  init_lims();

  for(int i=1; i<N; i++){
    not_found = 1;
    cidx = 0;
    location = 0;
    init_lims();
    do{
      // If the current node is a root :
      if (barnes_list[cidx] == 0){

        // Find the child node corresponding to our particle i
        find_square(x_[i],y_[i],1);
        fidx = cidx+location;

        // If child node is empty node, create the new node.
        if (barnes_list[fidx]==-1){
          barnes_list[fidx]=barnes_list.size();
          push_node(i, x_[i],y_[i],sin(theta_[i]),cos(theta_[i]));

          // We increment the number of children of this node
```

```
          barnes_list[cidx+1] += 1;
          N_child = barnes_list[cidx+1];
          barnes_list[cidx+2] = ((N_child-1)*barnes_list[cidx+2] + x_[i])/N_child;
          barnes_list[cidx+3] = ((N_child-1)*barnes_list[cidx+3] + y_[i])/N_child;
          barnes_list[cidx+4] = ((N_child-1)*barnes_list[cidx+4] + sin(theta_[i]))/N
_child;
          barnes_list[cidx+5] = ((N_child-1)*barnes_list[cidx+5] + cos(theta_[i]))/N
_child;

          // Declare having found final node of particle i
          // - can leave the do-while loop and go to next
          // particle i+1.
          not_found = 0;
        }
        // If child node not empty, navigate to this next node.
        else {
          barnes_list[cidx+1] += 1;
          N_child = barnes_list[cidx+1];
          barnes_list[cidx+2] = ((N_child-1)*barnes_list[cidx+2] + x_[i])/N_child;
          barnes_list[cidx+3] = ((N_child-1)*barnes_list[cidx+3] + y_[i])/N_child;
          barnes_list[cidx+4] = ((N_child-1)*barnes_list[cidx+4] + sin(theta_[i]))/N
_child;
          barnes_list[cidx+5] = ((N_child-1)*barnes_list[cidx+5] + cos(theta_[i]))/N
_child;
          cidx = barnes_list[fidx];
          xlim[0]=xlim_next[0];
          xlim[1]=xlim_next[1];
          xlim[2]=xlim_next[2];
          ylim[0]=ylim_next[0];
          ylim[1]=ylim_next[1];
          ylim[2]=ylim_next[2];
        }
      }

      // If current node just a value : need to turn the node into a root.
      else{

        // Copy the values of value-node, and push them as new node
        ox = barnes_list[cidx+2];
        oy = barnes_list[cidx+3];
        osint = barnes_list[cidx+4];
        ocost = barnes_list[cidx+5];
        oid = barnes_list[cidx+6];
        barnes_list[cidx] = 0;
        find_square(ox,oy,0);
        barnes_list[cidx+location]=barnes_list.size();
        push_node(oid, ox, oy, osint, ocost);
        // Note: current index "cidx" remains identical.
      }
    } while(not_found);
  }

#pragma omp parallel for
  for(int i=0; i<N; i++){
    cidx = 0;
    outputX[i] = 0.;
    outputY[i] = 0.;
    output_theta[i] = 0.;
    double sumx = 0.;
    double sumy = 0.;
    double sumtheta = 0.;
    int N_comp = 0;
    barnes_compute(0, i, x_[i], y_[i], theta_[i], sumx, sumy, sumtheta, N_comp, maxx
-minx);
    outputX[i] = (1./float(N_comp))*sumx;
    outputY[i] = (1./float(N_comp))*sumy;
    output_theta[i] = (float(K)/float(N_comp))*sumtheta;
  }

  barnes_list.clear();
}

void rk4::compute_xx(double t_, double* x_, double* y_, double* theta_, double* outp
```

```
utX, double* outputY, double* output_theta){

#pragma omp parallel for
  for (int i=0; i<N; i++){
    outputX[i] = 0.;
    outputY[i] = 0.;
    output_theta[i] = 0.;
    double sumx = 0.;
    double sumy = 0.;
    double sumtheta = 0.;
    for (int j=0; j<N; j++){
      if (j!=i){
        double norm2 = (x_[j]-x_[i])*(x_[j]-x_[i]) + (y_[j]-y_[i])*(y_[j]-y_[i]);
        double norm = sqrt(norm2);
        sumx += ((x_[j]-x_[i])/norm)*(AA+J*cos(theta_[j]-theta_[i])) - BB*((x_[j]-x_
[i])/norm2);
        sumy += ((y_[j]-y_[i])/norm)*(AA+J*cos(theta_[j]-theta_[i])) - BB*((y_[j]-y_
[i])/norm2);
        sumtheta += sin(theta_[j]-theta_[i])/norm;
      }
    }
    outputX[i] += (1./float(N))*sumx;
    outputY[i] += (1./float(N))*sumy;
    output_theta[i] += (float(K)/float(N))*sumtheta;
  }
}


void rk4::compute_Gs(double t, double* Gs_x, double* ff_x, double* Gs_y, double* ff_
y, double* Gs_theta, double* ff_theta){
  // First, we calculate G1:
#pragma omp parallel for
  for(int i=0; i<(N); i++){
    Gs_x[i]=x0[i];
    Gs_x[i+1*N]=x0[i];
    Gs_x[i+2*N]=x0[i];
    Gs_x[i+3*N]=x0[i];
    Gs_x[i+4*N]=x0[i];
    Gs_y[i]=y0[i];
    Gs_y[i+1*N]=y0[i];
    Gs_y[i+2*N]=y0[i];
    Gs_y[i+3*N]=y0[i];
    Gs_y[i+4*N]=y0[i];
    Gs_theta[i]=theta0[i];
    Gs_theta[i+1*N]=theta0[i];
    Gs_theta[i+2*N]=theta0[i];
    Gs_theta[i+3*N]=theta0[i];
    Gs_theta[i+4*N]=theta0[i];
  }
  // Then, we compute f(G1):
  if (enable_BH) smart_compute_xx(t+C[0]*h_step, x0, y0, theta0, ff_x, ff_y, ff_thet
a);
  else compute_xx(t+C[0]*h_step, x0, y0, theta0, ff_x, ff_y, ff_theta);

  // Calculating G2:
#pragma omp parallel for
  for(int i=0; i<N; i++){
    Gs_x[i+1*N]     += A[0]*h_step*ff_x[i];
    Gs_y[i+1*N]     += A[0]*h_step*ff_y[i];
    Gs_theta[i+1*N] += A[0]*h_step*ff_theta[i];
  }

  // Computing f(G2):
  if (enable_BH) smart_compute_xx(t+C[1]*h_step, (Gs_x+1*N), (Gs_y+1*N), (Gs_theta+1
*N), (ff_x+1*N), (ff_y+1*N), (ff_theta+1*N));
  else compute_xx(t+C[1]*h_step, (Gs_x+1*N), (Gs_y+1*N), (Gs_theta+1*N), (ff_x+1*N),
 (ff_y+1*N), (ff_theta+1*N));

  // Calculating G3:
  for (int i=0; i<N; i++){
    Gs_x[i+2*N]     += A[1]*h_step*ff_x[i];
    Gs_x[i+2*N]     += A[2]*h_step*ff_x[i+1*N];
    Gs_y[i+2*N]     += A[1]*h_step*ff_y[i];
```

```
    Gs_y[i+2*N]     += A[2]*h_step*ff_y[i+1*N];
    Gs_theta[i+2*N] += A[1]*h_step*ff_theta[i];
    Gs_theta[i+2*N] += A[2]*h_step*ff_theta[i+1*N];
  }

  // Computing f(G3):
  if (enable_BH) smart_compute_xx(t+C[2]*h_step, (Gs_x+2*N), (Gs_y+2*N), (Gs_theta+2
*N), (ff_x+2*N), (ff_y+2*N), (ff_theta+2*N));
  else compute_xx(t+C[2]*h_step, (Gs_x+2*N), (Gs_y+2*N), (Gs_theta+2*N), (ff_x+2*N),
 (ff_y+2*N), (ff_theta+2*N));


  // Calculating G4:
#pragma omp parallel for
  for (int i=0; i<N; i++){
    Gs_x[i+3*N]     += A[3]*h_step*ff_x[i];
    Gs_x[i+3*N]     += A[4]*h_step*ff_x[i+1*N];
    Gs_x[i+3*N]     += A[5]*h_step*ff_x[i+2*N];
    Gs_y[i+3*N]     += A[3]*h_step*ff_y[i];
    Gs_y[i+3*N]     += A[4]*h_step*ff_y[i+1*N];
    Gs_y[i+3*N]     += A[5]*h_step*ff_y[i+2*N];
    Gs_theta[i+3*N] += A[3]*h_step*ff_theta[i];
    Gs_theta[i+3*N] += A[4]*h_step*ff_theta[i+1*N];
    Gs_theta[i+3*N] += A[5]*h_step*ff_theta[i+2*N];
  }

  // Computing f(G4):
  if (enable_BH) smart_compute_xx(t+C[3]*h_step, (Gs_x+3*N), (Gs_y+3*N), (Gs_theta+3
*N), (ff_x+3*N), (ff_y+3*N), (ff_theta+3*N));
  else compute_xx(t+C[3]*h_step, (Gs_x+3*N), (Gs_y+3*N), (Gs_theta+3*N), (ff_x+3*N),
 (ff_y+3*N), (ff_theta+3*N));


  // Calculating G5:
#pragma omp parallel for
  for (int i=0; i<N; i++){
    Gs_x[i+4*N]     += A[6]*h_step*ff_x[i];
    Gs_x[i+4*N]     += A[7]*h_step*ff_x[i+1*N];
    Gs_x[i+4*N]     += A[8]*h_step*ff_x[i+2*N];
    Gs_x[i+4*N]     += A[9]*h_step*ff_x[i+3*N];
    Gs_y[i+4*N]     += A[6]*h_step*ff_y[i];
    Gs_y[i+4*N]     += A[7]*h_step*ff_y[i+1*N];
    Gs_y[i+4*N]     += A[8]*h_step*ff_y[i+2*N];
    Gs_y[i+4*N]     += A[9]*h_step*ff_y[i+3*N];
    Gs_theta[i+4*N] += A[6]*h_step*ff_theta[i];
    Gs_theta[i+4*N] += A[7]*h_step*ff_theta[i+1*N];
    Gs_theta[i+4*N] += A[8]*h_step*ff_theta[i+2*N];
    Gs_theta[i+4*N] += A[9]*h_step*ff_theta[i+3*N];
  }

  // Computing f(G5):
  if (enable_BH) smart_compute_xx(t+C[4]*h_step, (Gs_x+4*N), (Gs_y+4*N), (Gs_theta+4
*N), (ff_x+4*N), (ff_y+4*N), (ff_theta+4*N));
  else compute_xx(t+C[4]*h_step, (Gs_x+4*N), (Gs_y+4*N), (Gs_theta+4*N), (ff_x+4*N),
 (ff_y+4*N), (ff_theta+4*N));
}

void rk4::compute_y1y1h(double t, double* Gs_x, double* ff_x, double* Gs_y, double*
ff_y, double* Gs_theta, double* ff_theta){
  double sc_x[N];
  double sc_y[N];
  double sc_theta[N];
  for (int i=0; i<N; i++){
    sc_x[i] = 0.;
    sc_y[i] = 0.;
    sc_theta[i] = 0.;
    x1[i] = 0.;
    y1[i] = 0.;
    theta1[i] = 0.;
    x1h[i] = 0.;
    y1h[i] = 0.;
    theta1[i] = 0.;
  }
```

```
  for (int i=0; i<N; i++){
    x1[i]      = x0[i];
    y1[i]      = y0[i];
    theta1[i]  = theta0[i];
    x1h[i]     = x0[i];
    y1h[i]     = y0[i];
    theta1h[i] = theta0[i];
  }
  for (int i=0; i<N; i++){
    for (int j=0; j<4; j++){
      x1[i] += B[j]*h_step*ff_x[i+j*N];
      y1[i] += B[j]*h_step*ff_y[i+j*N];
      theta1[i] += B[j]*h_step*ff_theta[i+j*N];
    }
  }
  // We comment the adaptive timestep part
/*
  for (int i=0; i<N; i++){
    for (int j=0; j<5; j++){
      x1h[i] += B[j+4]*h_step*ff_x[i+j*N];
      y1h[i] += B[j+4]*h_step*ff_y[i+j*N];
      theta1h[i] += B[j+4]*h_step*ff_theta[i+j*N];
    }
  }
  for (int i=0; i<N; i++){
    sc_x[i] = Atol + Rtol * std::max(std::abs(x0[i]), std::abs(x1[i]));
    sc_y[i] = Atol + Rtol * std::max(std::abs(y0[i]), std::abs(y1[i]));
    sc_theta[i] = Atol + Rtol * std::max(std::abs(theta0[i]), std::abs(theta1[i]));
  }

  double err=0.0;

  for (int i=0; i<N; i++){
    err += ((x1[i]-x1h[i])/sc_x[i])*((x1[i]-x1h[i])/sc_x[i]);
    err += ((y1[i]-y1h[i])/sc_y[i])*((y1[i]-y1h[i])/sc_y[i]);
    err += ((theta1[i]-theta1h[i])/sc_theta[i])*((theta1[i]-theta1h[i])/sc_theta[i])
;
  }

  err *= (1./float(N));
  err = sqrt(err);
  last_h_step = h_step;
  h_step = h_step * std::min(facmax, std::max(facmin, fac*pow((1./err), (1./4.))));


  if (err>1){
    compute_Gs(t, Gs_x, ff_x, Gs_y, ff_y, Gs_theta, ff_theta);
    compute_y1y1h(t, Gs_x, ff_x, Gs_y, ff_y, Gs_theta, ff_theta);
  }
  else if (t+last_h_step+h_step>T_final){
    h_step = T_final-(t+last_h_step);
  }
  */
}

void rk4::hermite(double actual_t, double myTheta, char* filenameDense){
  std::ofstream myDense;
  myDense.open(filenameDense);
  double f0_x[N];
  double f0_y[N];
  double f0_theta[N];
  double f1_x[N];
  double f1_y[N];
  double f1_theta[N];

  if (enable_BH){
    smart_compute_xx(actual_t, x0, y0, theta0, f0_x, f0_y, f0_theta);
    smart_compute_xx(actual_t + last_h_step, x1, y1, theta1, f1_x, f1_y, f1_theta);
  }
  else {
    compute_xx(actual_t, x0, y0, theta0, f0_x, f0_y, f0_theta);
    compute_xx(actual_t + last_h_step, x1, y1, theta1, f1_x, f1_y, f1_theta);
```

```
  }

  for(int i=0; i<N; i++){
    double u_x = (1-myTheta)*x0[i] + myTheta*x1[i] + myTheta*(myTheta-1)*((1-2*myThe
ta)*(x1[i]-x0[i]) + (myTheta-1)*last_h_step*f0_x[i]+myTheta*last_h_step*f1_x[i]);
    double u_y = (1-myTheta)*y0[i] + myTheta*y1[i] + myTheta*(myTheta-1)*((1-2*myThe
ta)*(y1[i]-y0[i]) + (myTheta-1)*last_h_step*f0_y[i]+myTheta*last_h_step*f1_y[i]);
    double u_theta = (1-myTheta)*theta0[i] + myTheta*theta1[i] + myTheta*(myTheta-1)
*((1-2*myTheta)*(theta1[i]-theta0[i]) + (myTheta-1)*last_h_step*f0_theta[i]+myTheta*
last_h_step*f1_theta[i]);
    myDense<<(actual_t + myTheta*last_h_step)<<" "<<u_x <<" "<< u_y<<" "<<u_theta<<s
td::endl;
  }
  myDense.close();
}

void rk4::dense_output(double t_){
  int m1=0;
  if (float(int(t_/dense_stpsze)) == t_/dense_stpsze){
    m1 = int(t_/dense_stpsze)-1;
  }
  else{
    m1 = int(t_/dense_stpsze);
  }
  int m2 = int((t_+last_h_step)/dense_stpsze);
  for (int k=(m1+1); k<(m2+1); k++){
    char filenameDense[32];
    sprintf(filenameDense, "Dense%04d.txt", k);
    hermite(t_, (dense_stpsze*k-t_)/last_h_step, filenameDense);
  }
}

void rk4::nextStep(){
  for(int i=0; i<N; i++){
    x0[i]=x1[i];
    y0[i]=y1[i];
    theta0[i]=theta1[i];
  }
}

void rk4::compute_solution(double T_final_){
  T_final = T_final_;
  dense_stpsze = double(T_final/double(n_intvls));
  double t=0;
  double Gs_x[5*N];
  double ff_x[5*N];
  double Gs_y[5*N];
  double ff_y[5*N];
  double Gs_theta[5*N];
  double ff_theta[5*N];
  initialize();
  while(t<T_final){
    compute_Gs(t, Gs_x, ff_x, Gs_y, ff_y, Gs_theta, ff_theta);
    compute_y1y1h(t, Gs_x, ff_x, Gs_y, ff_y, Gs_theta, ff_theta);
    dense_output(t);
    t += last_h_step;
    //printf("t=%f, next_step=%f\n", t, h_step);
    step_counter += 1;
    nextStep();
  };
  //printf("Done! It tool us %d steps to perform the entire integration.", step_coun
ter);
}

void rk4::initialize(){
  srand (static_cast <unsigned> (time(0)));
  float max_xy = 2;
  float max_angle = 2*M_PI;
  //printf("N=%d\n", N);
  for(int i=0; i<N; i++){
    float x;
    float y;
    float phse;
```

```
    do {
      x = static_cast <float> (rand()) / (static_cast <float> (RAND_MAX/max_xy));
      y = static_cast <float> (rand()) / (static_cast <float> (RAND_MAX/max_xy));
      phse = static_cast <float> (rand()) / (static_cast <float> (RAND_MAX/max_angle
));
      x-=1;
      y-=1;
    } while ((x*x + y*y)>1);
    x0[i] = x;
    y0[i] = y;
    theta0[i] = phse;
    vx0[i] = 0;
    vy0[i] = 0;
    omega0[i] = 0;
  }
}
```