```
// **********



// shooting.C

// Berechnet die Energieeigenwerte des harmonischen Oszillators,
// -\psi'' = (E -x^2) \psi   mit   \psi(x=-\infty) = \psi(x=+\infty) = 0 .



// **********



#include <math.h>
#include <stdio.h>
#include <stdlib.h>



// **********



// ********************
// Physikalische Parameter, etc.
// ********************

// Schrödinger-Gleichung:
//   y = (\psi , \phi , E)
//   f = (\phi , (x^2-E) \psi , 0)

const int N = 3;  // Anzahl der Komponenten von y bzw. f.


// *****

// Plot 1

// Integration von t_0 bis t_1.
// const double t_0 = 0.0;
// const double t_1 = 10.0;

// Anfangsbedingungen y(t_0).
// double y_0[N] = { 1.0 , 0.0 , 1.0 - 0.000001};
// double y_0[N] = { 1.0 , 0.0 , 1.0 + 0.000001};

// *****

// Plot 2

// Integration von t_0 bis t_1.
const double t_0 = 10.0;
const double t_1 = 0.0;

// Anfangsbedingungen y(t_0).
double y_0[N] = { 1.0 , 0.0 , 5.0 };


// *****


// Berechnet f(y(t),t) * tau.

void f_times_tau(double *y_t, double t, double *f_times_tau_, double tau)
{
  if(N != 3)
    {
      fprintf(stderr, "Error: N != 3!\n");
      exit(EXIT_FAILURE);
    }

  f_times_tau_[0] = y_t[1] * tau;

  f_times_tau_[1] = (t*t - y_t[2]) * y_t[0] * tau;

  f_times_tau_[2] = 0.0;
}



// **********



// ********************
// RK-Parameter.
// ********************


// #define __EULER__
// #define __RK_2ND__
// #define __RK_3RD__
#define __RK_4TH__

#ifdef __EULER__
const int order = 1;
#endif

#ifdef __RK_2ND__
const int order = 2;
#endif

#ifdef __RK_3RD__
const int order = 3;
#endif

#ifdef __RK_4TH__
const int order = 4;
#endif

// Anzahl der RK-Schritte.
const int num_steps = 1000;

double tau = (t_1 - t_0) / (double)num_steps;  // Schrittweite.

double h = 0.000001;  // Diskretisierungsschritt für numerische Ableitung.

double dE_min = 0.0000001;  // Abbruchkriterium für Newton-Raphson.


// **********



#ifdef __EULER__

// Euler-Schritt um Schrittweite tau.

void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
{
  int i1;

  // *****

  // Berechne k1 = f(y(t),t) * tau.

```

```
  double k1[N];
  f_times_tau(y_t, t, k1, tau);

  // *****

  for(i1 = 0; i1 < N; i1++)
    y_t_plus_tau[i1] = y_t[i1] + k1[i1];
}

#endif



#ifdef __RK_2ND__

// RK-Schritt (2nd order) um Schrittweite tau.

void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
{
  int i1;

  // *****

  // Berechne k1 = f(y(t),t) * tau.

  double k1[N];
  f_times_tau(y_t, t, k1, tau);

  // *****

  // Berechne k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau.

  double y_[N];

  for(i1 = 0; i1 < N; i1++)
    y_[i1] = y_t[i1] + 0.5*k1[i1];

  double k2[N];
  f_times_tau(y_, t + 0.5*tau, k2, tau);

  // *****

  for(i1 = 0; i1 < N; i1++)
    y_t_plus_tau[i1] = y_t[i1] + k2[i1];
}

#endif



#ifdef __RK_3RD__

// RK-Schritt (3rd order) um Schrittweite tau.

void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
{
  int i1;

  // *****

  // Berechne k1 = f(y(t),t) * tau.

  double k1[N];
  f_times_tau(y_t, t, k1, tau);

  // *****

  // Berechne k2 = f(y(t)+k1 , t+tau) * tau.

  double y_[N];

  for(i1 = 0; i1 < N; i1++)
    y_[i1] = y_t[i1] + k1[i1];

  double k2[N];
  f_times_tau(y_, t + tau, k2, tau);

  // *****

  // Berechne k3 = f(y(t)+(1/4)*(k1+k2) , t+(1/2)*tau) * tau.

  for(i1 = 0; i1 < N; i1++)
    y_[i1] = y_t[i1] + 0.25 * (k1[i1] + k2[i1]);

  double k3[N];
  f_times_tau(y_, t + 0.5*tau, k3, tau);

  // *****

  for(i1 = 0; i1 < N; i1++)
    y_t_plus_tau[i1] = y_t[i1] + (k1[i1] + k2[i1] + 4.0*k3[i1]) / 6.0;
}

#endif



#ifdef __RK_4TH__

// RK-Schritt (4th order) um Schrittweite tau.

void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
{
  int i1;

  // *****

  // Berechne k1 = f(y(t),t) * tau.

  double k1[N];
  f_times_tau(y_t, t, k1, tau);

  // *****

  // Berechne k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau.

  double y_[N];

  for(i1 = 0; i1 < N; i1++)
    y_[i1] = y_t[i1] + 0.5*k1[i1];

  double k2[N];
  f_times_tau(y_, t + 0.5*tau, k2, tau);

  // *****

  // Berechne k3 = f(y(t)+(1/2)*k2 , t+(1/2)*tau) * tau.

  for(i1 = 0; i1 < N; i1++)
    y_[i1] = y_t[i1] + 0.5*k2[i1];

  double k3[N];
  f_times_tau(y_, t + 0.5*tau, k3, tau);

  // *****

  // Berechne k4 = f(y(t)+k3 , t+tau) * tau.

  for(i1 = 0; i1 < N; i1++)
    y_[i1] = y_t[i1] + k3[i1];

  double k4[N];
```

```
  f_times_tau(y_, t + tau, k4, tau);

  // *****

  for(i1 = 0; i1 < N; i1++)
    {
      y_t_plus_tau[i1] =
        y_t[i1] + (k1[i1] + 2.0*k2[i1] + 2.0*k3[i1] + k4[i1]) / 6.0;
    }
}

#endif



// **********



// RK-Entwicklung der "Wellenfunktion".

double t[num_steps+1];  // Diskretisierte Zeitachse.
double y[num_steps+1][N];  // Diskretisierte "Bahnkurven".

double RK(bool output = false)
{
  double d1;
  int i1, i2;

  // *****

  // Führe Euler/RK-Schritte aus.

  for(i1 = 0; i1 < num_steps; i1++)
    {
      // y(t)   -->   y(t+\tau)
      RK_step(y[i1], t[i1], y[i1+1], tau);

      t[i1+1] = t[i1] + tau;
    }

  // *****

  if(output == true)
    {
      // Ausgabe.

      for(i1 = 0; i1 <= num_steps; i1++)
        {
          printf("%9.6lf %.5e %.5e %9.6lf\n",
                 t[i1], y[i1][0], y[i1][1], y[i1][2]);
        }
    }

  // *****

  // return y[num_steps][1];  // Fuer P = + (0., 2., 4., ... Zustand)
  return y[num_steps][0];  // Fuer P = + (1., 3., 5., ... Zustand)
}



// **********



int main(int argc, char **argv)
{
  int i1;

  // *****

  // Initialisiere "Bahnkurven" mit Anfangsbedingungen.

  t[0] = t_0;

  for(i1 = 0; i1 < N; i1++)
    y[0][i1] = y_0[i1];

  // **********
  // **********
  // **********

  // Einmalige RK-Entwicklung.

  // **********
  // **********
  // **********

  /*
  RK(true);
  */

  // **********
  // **********
  // **********

  // **********
  // **********
  // **********

  // Grobes Scannen der moeglichen Energieeigenwerte.

  // **********
  // **********
  // **********

  /*
  double E_min = 0.0;
  double E_max = 12.0;

  double E_step = 0.1;

  for(double E = E_min; E <= E_max; E += E_step)
    {
      // Modifiziere Anfangsbedingungen.
      y[0][N-1] = E;

      // Führe Euler/RK-Schritte aus.
      double psi_1 = RK(false);

      // *****

      printf("%+.5e %+.5e.\n", E, psi_1);
    }
  */

  // **********
  // **********
  // **********

  // **********
  // **********
  // **********

  // Newton-Raphson zur präzisen Bestimmung einzelner Energieeigenwerte.

  // **********
  // **********
  // **********

  // /*
```

```
427   // double E = 0.9;  // Initialer Schätzwert für Energieeigenwert.
428   // double E = 2.9;  // Initialer Schätzwert für Energieeigenwert.
429   // double E = 4.9;  // Initialer Schätzwert für Energieeigenwert.
430   double E = 6.9;  // Initialer Schätzwert für Energieeigenwert.
431
432   fprintf(stderr, "E_num = %+10.6lf .\n", E);
433
434   while(1)
435     {
436       // Modifiziere Anfangsbedingungen.
437       y[0][N-1] = E;
438
439       // Führe Euler/RK-Schritte aus.
440       double psi_1_E = RK(false);
441
442       // *****
443
444       // Modifiziere Anfangsbedingungen.
445       y[0][N-1] = E-h;
446
447       // Führe Euler/RK-Schritte aus.
448       double psi_1_E_mi_h = RK(false);
449
450       // Modifiziere Anfangsbedingungen.
451       y[0][N-1] = E+h;
452
453       // Führe Euler/RK-Schritte aus.
454       double psi_1_E_pl_h = RK(false);
455
456       double dpsi_1_E = (psi_1_E_pl_h - psi_1_E_mi_h) / (2.0 * h);
457
458       // *****
459
460       // Newton-Raphson-Schritt.
461
462       double dE = psi_1_E / dpsi_1_E;
463
464       if(fabs(dE) < dE_min)
465         break;
466
467       E = E - dE;
468
469       // *****
470
471       fprintf(stderr, "E_num = %+10.6lf.\n", E);
472     }
473
474   // Ausgabe der Wellenfunktion.
475   RK(true);
476
477   // */
478
479   // **********
480   // **********
481   // **********
482
483   return EXIT_SUCCESS;
484 }
485
486
487
488 // **********
```