Stiff systems

Computational Physics

Adam Reyes

October 31, 2013

1 Introduction

Systems governed by differential equations with time scales that differ by at least an order of magnitude are called Stiff Systems. Because these systems can have some time scales that are much faster than others they can be extremely sensitive to small perturbations. I will investigate the use of numerical solutions to these systems, namely in the under-damped harmonic oscillator and in a reaction network.

Up to this point in this course I have been using exclusively explicit methods to solve differential equations. These solutions involve using information about the derivatives at the current time step to jump to the next. As we will see these methods are unstable at low resolution. Instead if we use implicit methods, those that use the derivatives at future time steps to make the jump, we will see that they are stable even at low resolutions

2 Linear Systems

We now consider the linear system of equations:

$$\begin{pmatrix} \dot{x} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -\omega^2 & -\gamma \end{pmatrix} \begin{pmatrix} x \\ v \end{pmatrix}. \tag{2.1}$$

This is the well-known damped harmonic oscillator. For this my solutions I have chosen $\gamma = 1$ and $\omega = 30$. The analytic solution is given by

$$A(t) = e^{-\frac{\gamma t}{2}}(\cos\omega_1 t + \frac{\gamma}{2\omega}\sin\omega_1 t) \tag{2.2}$$

Where $\omega_1 = \sqrt{\omega^2 - (\frac{\gamma}{2})^2}$. Figure 2.1 shows this plotted along side a high resolution numerical solution to (2.1) using the Mid-Point method described in detail in previous writeups. We can see that at high resolutions our explicit methods converge to the actual solution. Figures 2.2 and 2.3 show the two explicit methods converging to the actual solution as resolution is increased. But as can be seen in Figure 2.6 These methods are very unstable for low resolutions. Both of these solutions have amplitudes that increase with time, instead of being damped out.

The first of our explicit methods is Backwards Euler. The basic idea is to advance the system as follows

$$z_{n+1} = (I - hF)^{-1} z_n \tag{2.3}$$

The convergence of this method with step size can be seen in Figure 2.4. The other method, Crank-Nichelson, boils down to essentially doing a Forward Euler step then a Backward Euler step, each by a half time step.

$$z_{n+1} = (I - \frac{h}{2}F)^{-1}(I + \frac{h}{2}F)z_n \tag{2.4}$$

The convergence of this solution with step size is shown in Figure 2.5. The stability of these two solutions at low resolution can be seen in Figure 2.7. While these solutions are not very accurate, they still manage to damp out unlike the explicit methods. The unfortunate downside to these methods is that, while we have gained stability, we now have to invert a matrix at every time step, which can be computationally expensive.

We can see in Figure 2.8 how the L_2 error, established in previous writeups, propagates with the number of steps taken in a given time interval. We can further see how the explicit methods are unstable at low resolution, as their error is quite large. After they stabilize we see they converge in the usual way with lines of differing slopes. Meanwhile the implicit methods remain fairly level at low resolution, then begin to converge in straight lines on the log-log plot. We can see that Backward Euler converges roughly as Forward Euler, which has been previously established as a first-order method. Crank-Nichelson converges roughly with Mid-Point, already established as a second-order method.

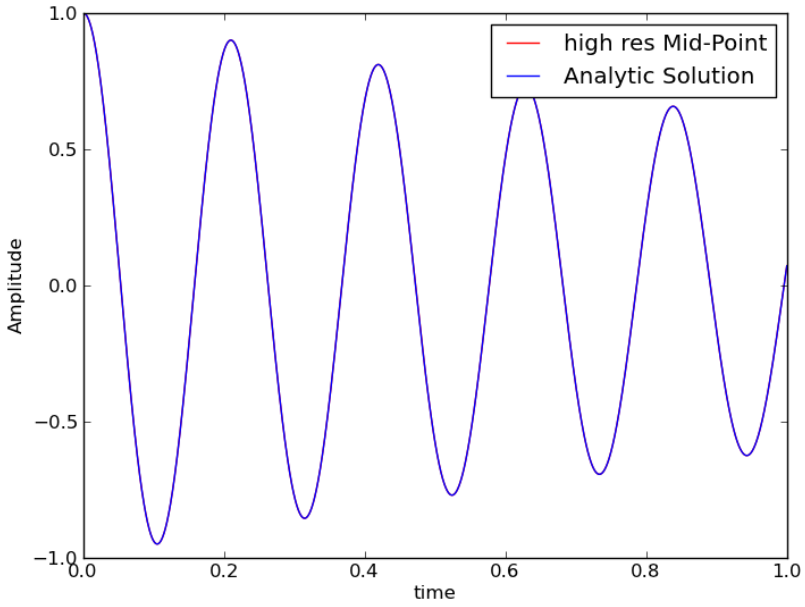

Figure 2.1: High resolution Mid-Point plotted with the analytic solution

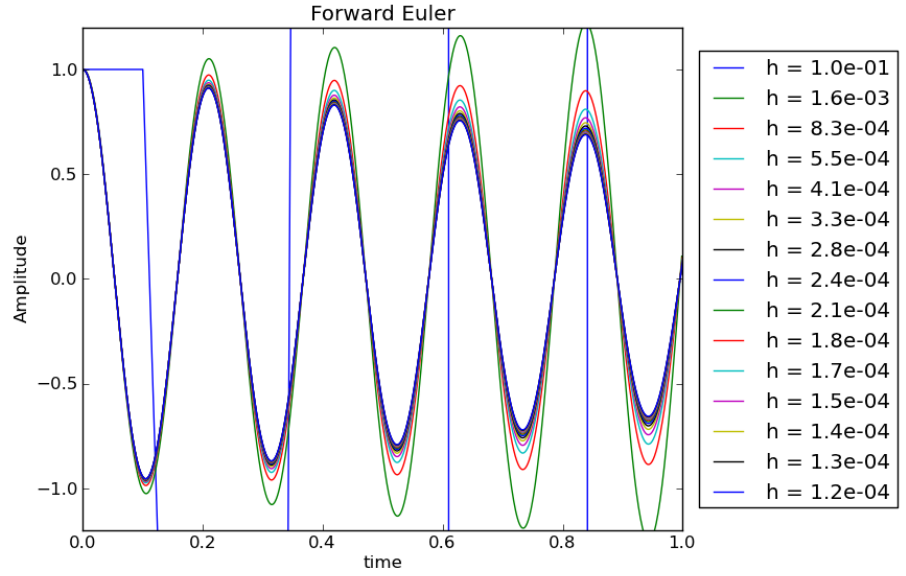

Figure 2.2: Forward Euler at various timesteps

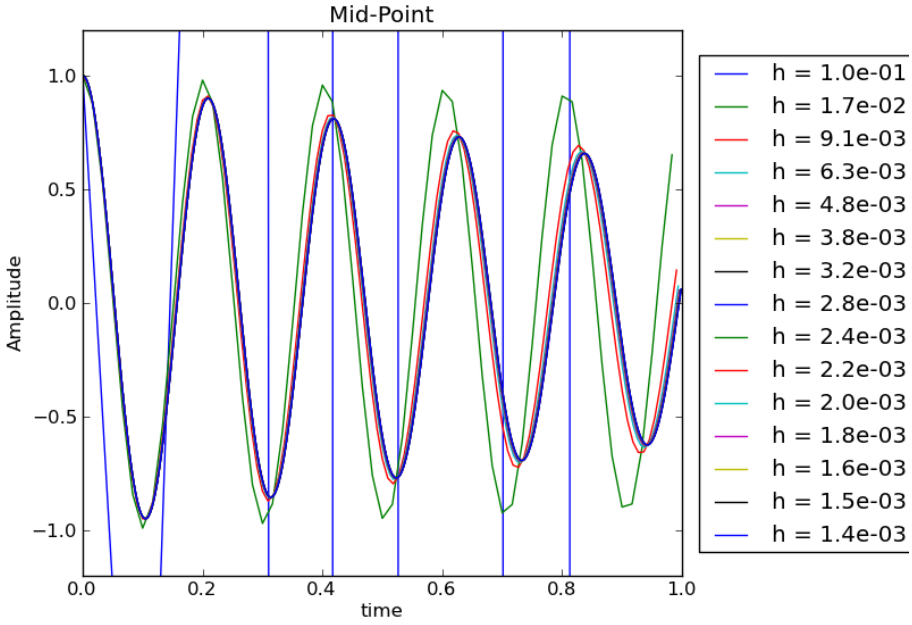

Figure 2.3: MidPoint at various timesteps

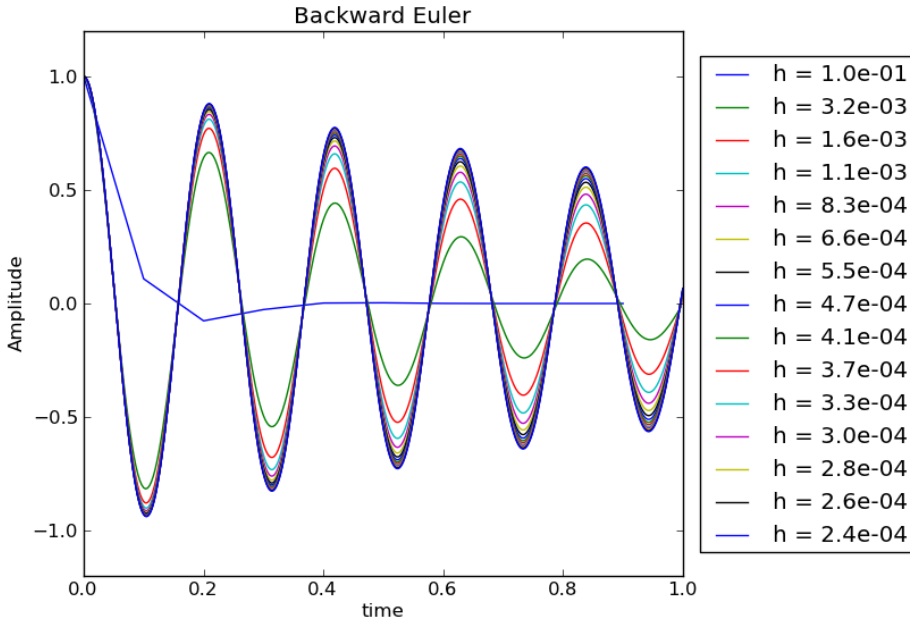

Figure 2.4: Backward Euler at various timesteps

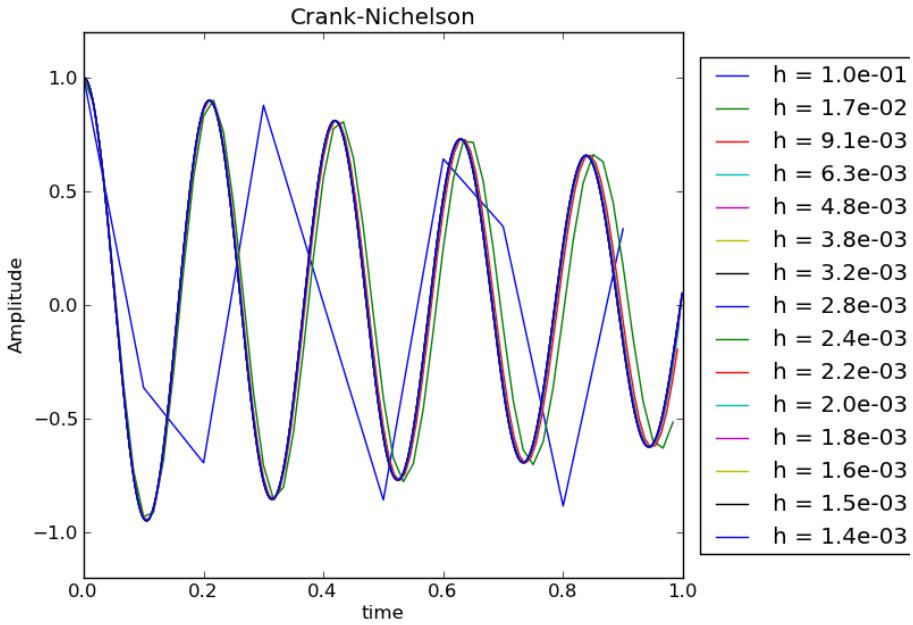

Figure 2.5: Crank-Nichelson at various timesteps

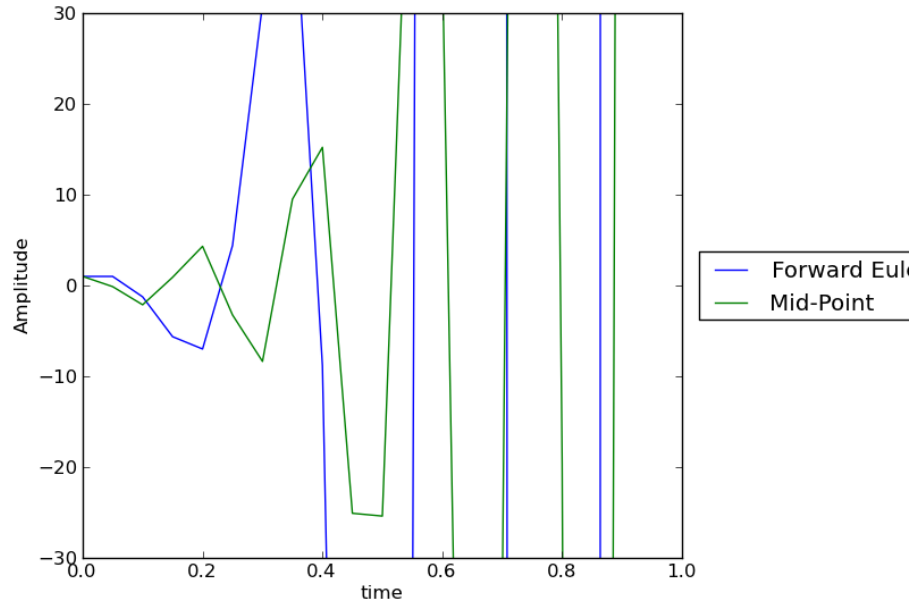

Figure 2.6: Forward Euler and Mid-Point at low resolution

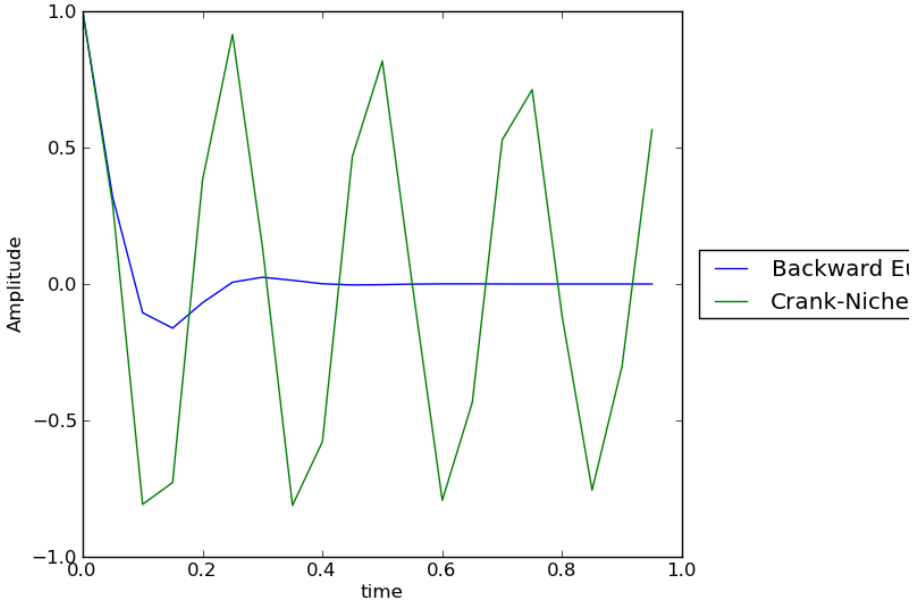

Figure 2.7: Backward Euler and Crank-Nichelson at low resolution

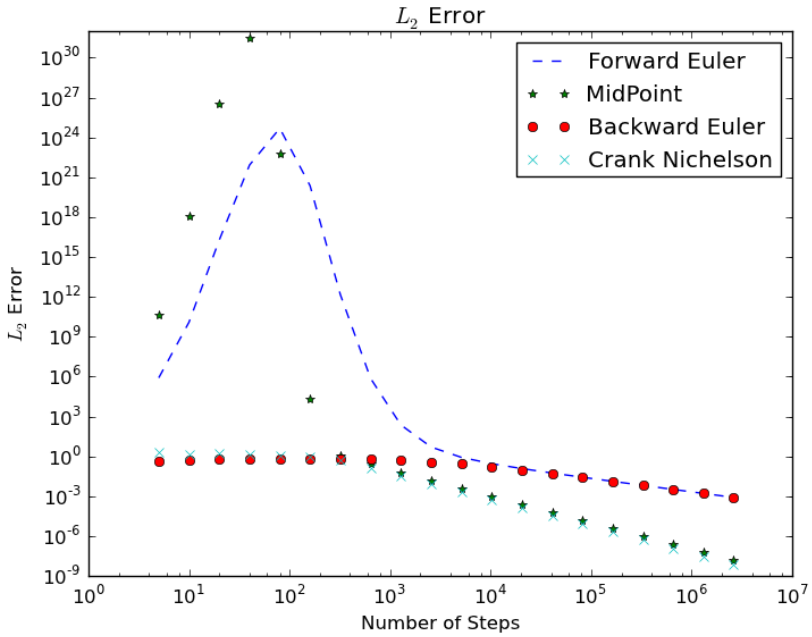

Figure 2.8: L_2 Error plotted against number of time steps over a 5 unit time interval

3 Non-Linear Systems

For our non-linear reaction network we will use the following system of equations:

$$\vec{z} = \begin{pmatrix} \dot{A} \\ \dot{X} \\ \dot{Y} \end{pmatrix} = \begin{pmatrix} -k_0 AX \\ k_0 AX - k_1 XY \\ k_1 XY - k_2 Y \end{pmatrix} \tag{3.1}$$

For this example I have chosen the following conditions

$$k_0 = 1 \tag{3.2}$$
$$k_1 = 100 \tag{3.3}$$
$$k_2 = 1000 \tag{3.4}$$
$$\tag{3.5}$$

Since we are now dealing with a non-linear system of equations, in order to solve the system with our previously described methods we must linearize the set of equations. This is done by taking the first order series approximation:

$$\vec{F}(\vec{z}_{n+1}) = \vec{F}(\vec{z}_n) + hJ(\vec{z}_n) \cdot F(\vec{z}_n) + O(h^2). \tag{3.6}$$

Where $J(\vec{z}_n)$ is the Jacobian evaluated at $\vec{z}_n$, and h is the time step.In our case $\vec{F}$ is the vector valued function that takes $\vec{z} \to \vec{z}$. Now with our linearized system of equations we can use our methods to solve numerically for the species amount of our reaction network as a function of time. Figure 3.1 shows the Forward Euler solution at a low and high resolution. As we know Forward Euler will always converge for high enough resolution, but we can see that at low resolution Forward Euler fails to capture a lot of the transient behavior that can be seen with the higher resolution. Further, at low resolution you can see that the species X and Y are tending to diverge towards the end of the time interval, whereas in the high res solution they converge to some values.

By contrast, in Figure 3.2, we can see the Backward Euler solution remains stable even at the lower resolution, although it doesn't quite capture all of the transient behavior. And then again we see the same solution for higher resolution.

The true advantage of the implicit method can be seen when we implement a adaptive step size method in the solutions. The step size, h, was determined by

$$h = h_0 \left(\frac{tol}{\delta y} \right)^{\frac{1}{n}} \tag{3.7}$$

Where δy is the difference in y calculated first by taking a step h_0 then two steps of $\frac{1}{2}h_0$. tol is some tolerance and n is the order of the method being used. For this example I have chosen

$$h_0 = 10^{-5} \tag{3.8}$$

$$tol = 10^{-3} \tag{3.9}$$

Figure 3.3 shows the results of this implementation. Notice that in the Forward Euler implementation is almost always at a high resolution, but still does not quite manage to capture all of the detail, and is not stable as species X is tending to diverge at the end. The criterion had wanted to use an even higher resolution on the order of 10^{-8}, and became to computationally taxing so I enforced a minimum time step of 10^{-5}, and still took more than a few seconds to run.

On the other side of things the Backwards Euler implementation ran in about a second, and as can be seen very well captures the behavior of the solution. Notice that the resolution greatly increases at the intervals of transient behavior.

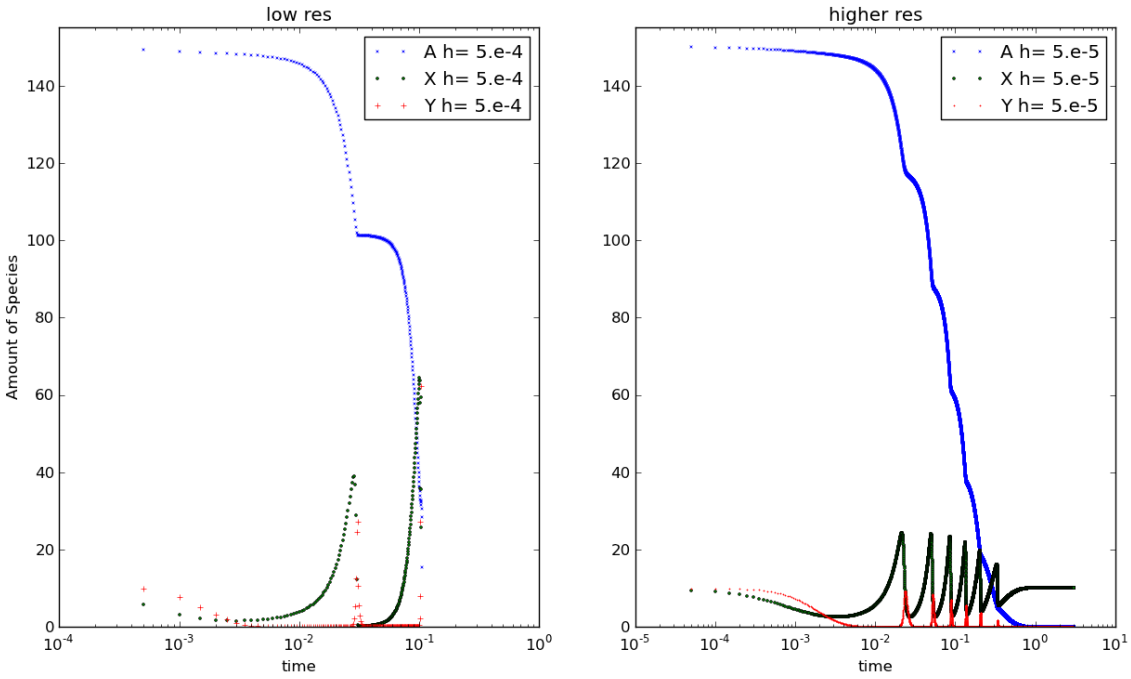

Figure 3.1: Forward Euler at two resolutions

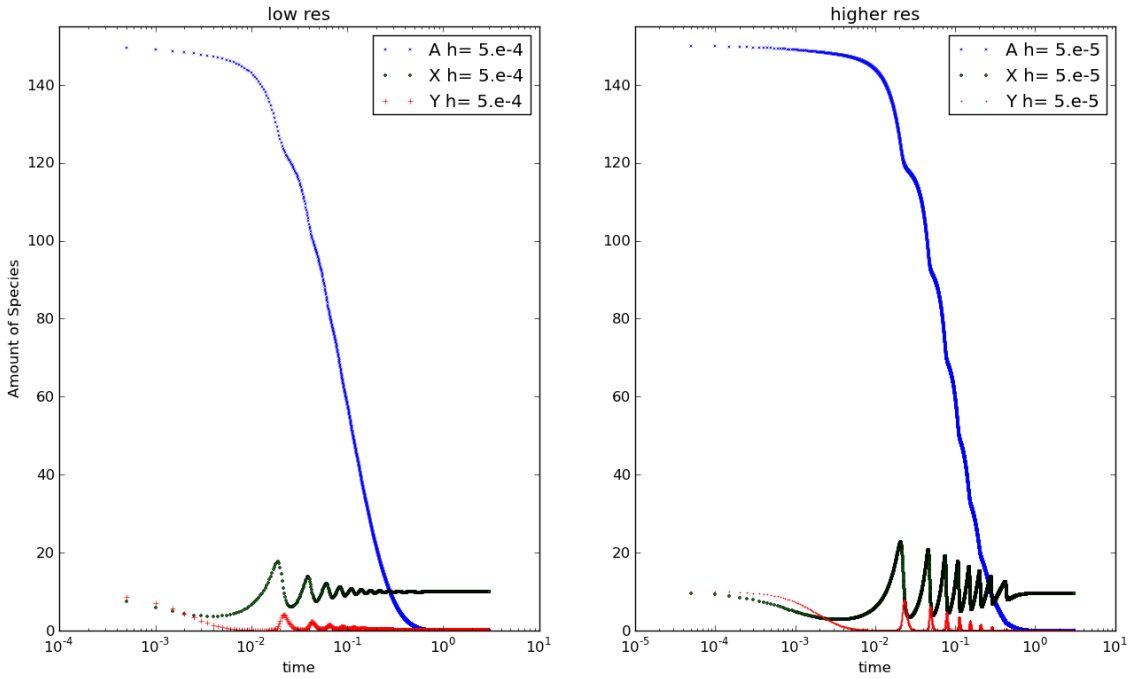

Figure 3.2: Backward Euler at two resolutions

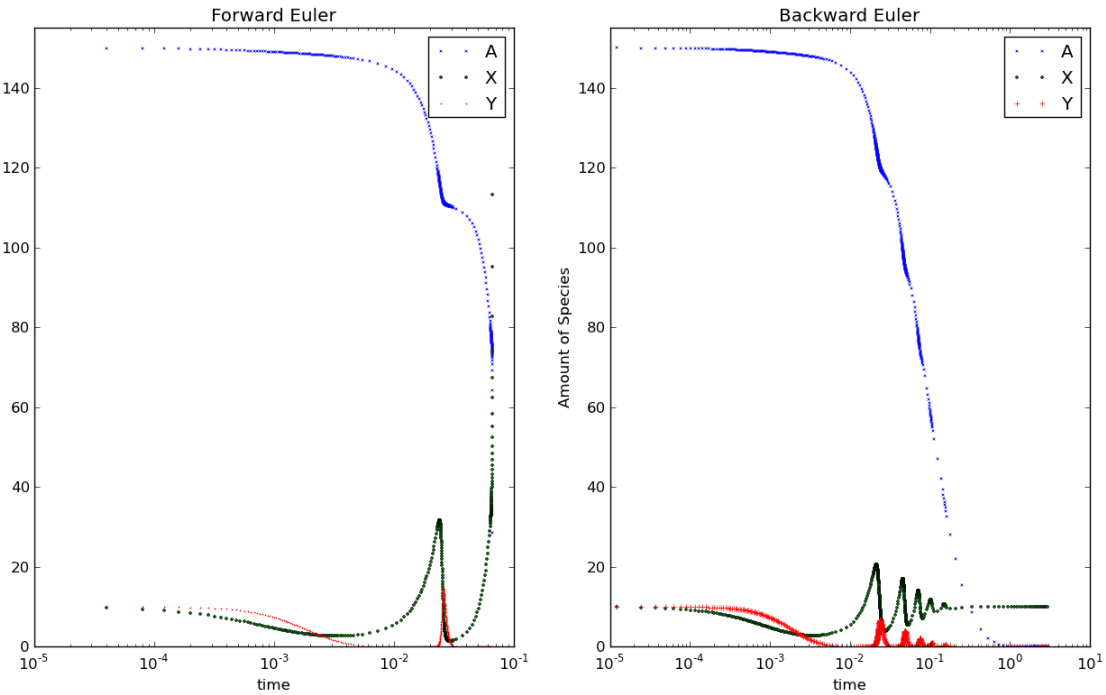

Figure 3.3: Backward and Forward Euler using variable time step

4 Source Code

Listing 1: DampedOscillator

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

double gam = 1.;
double omega = 30.0;
int N_x = 2;
int N_y = 2;

////////////Physics///////
double  exact(double t){
  double om = sqrt(omega*omega - 0.25*gam*gam);
  double ans = exp(-0.5*gam*t)*(cos(om*t) + 0.5*gam/om*sin(om*t));
  //printf("%e %e\n", t, ans);
  return(ans);
}
void osc(double * F){
  F[0] = 0; //F00
  F[2] = 1; //F01
  F[1] = -pow(omega,2); //F10
  F[3] = -gam; //F11
}
////////////local functions////////////
void mult2x2(double * F, double* a){
  double a_n[2] = {a[0], a[1]};
  a_n[0] = F[0]*a[0] + F[2]*a[1];
  a_n[1] = F[1]*a[0] + F[3]*a[1];
  a[0] = a_n[0];
  a[1] = a_n[1];
}
void invert2x2( double *F, double * Fi){
  double det = F[0]*F[3] - F[1]*F[2];
  Fi[0] =  F[3]/det;
  Fi[1] = -F[1]/det;
  Fi[2] = -F[2]/det;
  Fi[3] =  F[0]/det;
}
void inverse33( double ** a , double ** inv ){

   int i;
   double det = 0.0;
   for( i=0 ; i<3 ; ++i )
      det += a[0][i]*(a[1][(i+1)%3]*a[2][(i+2)%3] - a[1][(i+2)%3]*a[2][(i+1)%3])
          ;

   int j;
   for(i=0;i<3;i++){
      for(j=0;j<3;j++)
           inv[j][i] = ((a[(i+1)%3][(j+1)%3] * a[(i+2)%3][(j+2)%3]) - (a[(i+1)
               %3][(j+2)%3]*a[(i+2)%3][(j+1)%3]))/det ;
   }
}
void Feuler2x2( double dt, double * z, double * F){
  double z_n[2] = {z[0], z[1]};
  mult2x2(F, z_n);
  z[0] += dt*z_n[0];
  z[1] += dt*z_n[1];
}
```

```
void Mid2x2(double dt, double * z, double * F){
  double z_half[2] = {z[0], z[1]};
  Feuler2x2(0.5*dt, z_half, F);
  mult2x2(F, z_half);
  z[0] += dt*z_half[0];
  z[1] += dt*z_half[1];
  }
void Beuler2x2(double dt, double *z, double * F){
  double T[4]= {0}; //* T = (double*)malloc(N_x*N_y*sizeof(double));
  double Ti[4]= {0};// = (double*)malloc(N_x*N_y*sizeof(double));
  double y = 0.0;
  int i;
  int j;
  for( i=0; i<2; ++i){
    for( j=0; j<2; ++j){
      if(i==j){
  y = 1.0;
      }
      else{
  y = 0.0;
      }
      T[i + 2*j] = y - dt*F[i + 2*j];
    }
  }
  invert2x2(T, Ti);
  double z_n[2] = {z[0], z[1]};
  mult2x2(Ti, z_n);
  z[0] = z_n[0];
  z[1] = z_n[1];
  //free(T);
  //free(Ti);
}
void Crank(double dt, double *z, double * F){
  double T[4] = {0};// = (double*)malloc(N_x*N_y*sizeof(double));
  double Ti[4]= {0};// = (double*)malloc(N_x*N_y*sizeof(double));
  double  z_n[2] = {z[0], z[1]};
  double y;
  int i, j;
  for( i=0; i<2; ++i){
    for( j=0; j<2; ++j){
      if(i==j){
  y = 1.0;
      }
      else{
  y = 0.0;
      }
      T[i + 2*j] = y + 0.5*dt*F[i + 2*j];
    }
  }
  mult2x2(T, z_n);
  for( i=0; i<2; ++i){
    for( j=0; j<2; ++j){
      if(i==j){
  y = 1.0;
      }
      else{
  y = 0.0;
      }
      T[i + 2*j] = y - 0.5*dt*F[i + 2*j];
    }
  }
  invert2x2(T, Ti);
  mult2x2(Ti, z_n);
```

```
  z[0] = z_n[0];
  z[1] = z_n[1];
  //free(T);
  //free(Ti);
}
void advance(double dt, double * z, int method){
  double F[4] = {0};// = (double*)malloc(N_x*N_y*sizeof(double));//F[i + j*N_y]
  osc(F);
  //printf("%e   %e\n%e    %e\n\n", F[0], F[2], F[1], F[3]);
  if( method == 1){
    Feuler2x2(dt, z, F);
  }
  else if(method ==2){
    Mid2x2(dt, z, F);
    }
  else if(method == 3){
    Beuler2x2(dt, z, F);
  }
  else if(method ==4){
    Crank(dt, z, F);
  }
  //free(F);
}

double error( z0, t, h ){
  double diff = pow((z0-exact(t)),2);
  return(h*diff);
}
////////////////Main//////////////////
int main(int argc, char **argv){
  // /a.out T N method

  FILE* fid,* fa;
  fid = fopen("oscdata.dat", "w");
  fa = fopen("L21.dat","a");
  double T = atof(argv[1]);
  int N = atoi(argv[2]);
  int method = atoi(argv[3]);
  double dt = T/(double)N;
  double t = 0.0;
  double z[2] = {1., 0.};
  double sum = 0.;

  int i;
  for(i = 0; i<N; ++i){
    //printf("%e %e %e\n", t, z[0], exact(t));
    fprintf(fid, "%e %e %e\n", t, z[0], exact(t));
    sum += dt*pow(fabs(z[0]-exact(t)),2);
    //printf("%e\n", dt*pow(z[0]-exact(t),2));
    //printf("%e %e %e\n", t, z[0], z[1]);
    advance(dt, z, method);
    t += dt;
  }
  double L2 = sqrt(sum);
  fprintf(fa, "%d %e\n", N, L2);
}
```

Listing 2: Reaction Network

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

```
void advance(double  h, double * z, int method);
double TOL = 1.e-3;
double h0 = 1.e-5;
double K[3] = {1. , 100. , 1000.};
////////////PHYSICS///////////////
void F(double * z, double * z_n){
  double An = -K[0]*z[0]*z[1];
  double Xn = K[0]*z[0]*z[1] - K[1]*z[1]*z[2];
  double Yn = K[1]*z[1]*z[2] - K[2]*z[2];
  z_n[0] = An;
  z_n[1] = Xn;
  z_n[2] = Yn;
}
void DF(double * z, double ** dF){
  dF[0][0] = -K[0]*z[1];
  dF[0][1] = -K[0]*z[0];
  dF[0][2] = 0.0;
  dF[1][0] = K[0]*z[0];
  dF[1][1] = K[0]*z[0] - K[1]*z[2];
  dF[1][2] = -K[1]*z[1];
  dF[2][0] = 0.0;
  dF[2][1] = K[1]*z[2];
  dF[2][2] = K[1]*z[1] - K[2];
  //this is a -dF thats why theres a plus later
}
////////////MATH/////////////////
void inverse33( double ** a , double ** inv ){

  int i;
  double det = 0.0;
  for( i=0 ; i<3 ; ++i )
    det += a[0][i]*(a[1][(i+1)%3]*a[2][(i+2)%3] - a[1][(i+2)%3]*a[2][(i+1)%3]) ;

  int j;
  for(i=0;i<3;i++){
    for(j=0;j<3;j++)
      inv[j][i] = ((a[(i+1)%3][(j+1)%3] * a[(i+2)%3][(j+2)%3]) - (a[(i+1)%3][(j
          +2)%3]*a[(i+2)%3][(j+1)%3]))/det ;
  }
}
void mult33(double ** A, double ** B, double ** C){
  int i, j, k;
  for(i=0;i<3;++i){
    for(j=0;j<3;++j){
      C[i][j] = 0.;
      for(k=0;k<3;++k){
  C[i][j] += A[i][k]*B[k][j];
      }
    }
  }
}
void dot33(double ** A, double * z, double * z_n){
  int i,j;
  double zn[3] = {0,0,0};
  for(i=0;i<3;++i){
    for(j=0;j<3;++j){
      zn[i] += A[i][j]*z[j];
    }
  }
  z_n[0] = zn[0];
  z_n[1] = zn[1];
  z_n[2] = zn[2];
```

```
}
void init33(double *** A){
  *A = (double**)malloc(3*sizeof(double*));
  int i, j;
  for (i=0;i<3;++i){
    (*A)[i] = (double*)malloc(3*sizeof(double));
    for (j=0; j<3;++j){
      (*A)[i][j] = 0.;
    }
  }
}
double getstep(double * z, int method){
  double z1[3];
  double z2[3];
  int i;
  for(i=0;i<3;++i){
    z1[i] = z[i];
    z2[i] = z[i];
  }
  double h1 = h0;
  double h2 = 2.*h0;
  double order = 1.0;

  advance( h1, z1, method);
  advance( h1, z1, method);
  advance( h2, z2, method);

  double delta = 0.;
  double del;
  for(i=0;i<3;++i){
    del = fabs(z1[i] - z2[i]);
    if(del>delta){
      delta = del;
    }
  }
  double h_n = pow(TOL/delta, 1./order)*h0;
  if(h_n > 0.1){
    h_n = 0.1;
  }
  else if(h_n < 4.e-5){
    h_n = 4.e-5;
  }
  return(h_n);
}
////////////INTEGRATION FUNCTIONS////////////
void Feuler(double * z, double h){
  double zn[3];
  F(z, zn);
  z[0] += h*zn[0];
  z[1] += h*zn[1];
  z[2] += h*zn[2];
}
void Beuler(double * z, double h){
  double ** dF, **Ti, **T;
  init33(&dF);
  init33(&Ti);
  init33(&T);
  DF(z, dF);
  double y = 0.0;
  int i, j;
  for( i=0; i<3; ++i){
    for( j=0; j<3; ++j){
      if(i==j){
```

```
    y = 1.0;
        }
        else{
    y = 0.0;
        }
        T[i][j] = y - h*dF[i][j];
      }
    }
    inverse33(T, Ti);
    double zo[3], Fz[3];
    F(z, Fz);
    dot33(Ti, Fz, zo);
    for ( i = 0;i<3;++i){
      z[i]+= h*zo[i];
    }
  }
  void test(double * z, double  h){
    int i;
    for (i=0;i<3;++i){
      z[i] = z[i] - 2*h*z[i];
    }
  }

  void advance(double  h, double * z, int method){
    if (method == 1){
      Feuler(z, h);
    }
    else if (method == 2){
      Beuler(z, h);
    }
    else if (method == 7){
      test(z, h);
    }
  }
  ///////////////MAIN//////////////
  int main(int argc, char **argv){
    // ./a.out method h
    double T = 3.0;
    double A = 150.;
    double X = 10.;
    double Y = 10.;
    double z[3] = {A, X, Y};
    double t = 0.;
    FILE * fid;
    double h;
    //h = atof(argv[2]);
    int method = atoi(argv[1]);
    fid = fopen("best.dat", "w");
    while(t<T){
      fprintf(fid, "%e %e %e %e\n", t, z[0], z[1], z[2]);
      h = getstep(z, method);
      //printf("%e\n", h);
      advance(h, z, method);
      printf("%e\n", t);
      t+= h;
      }
  }
```