Chapter 9
Numerical Methods for DNLS

Kody J.H. Law and Panayotis G. Kevrekidis

9.1 Introduction

In this section, we briefly discuss the numerical methods that have been used extensively throughout this book to obtain the numerical solutions discussed herein, as well as to analyze their linear stability and to propagate them in time (e.g., to examine their dynamical instability, or to confirm their numerical stability).

Our tool of preference, regarding the numerical identification of solutions consists of the so-called Newton–Raphson (or simply Newton) method. We choose the Newton method because of its quadratic convergence, upon the provision of a suitably good initial guess [1]. It should be clearly indicated here that different groups use different methods to obtain stationary solutions. For instance, methods based on rewriting the standing wave problems of interest in Fourier space and applying Petviashvili's iteration scheme have been proposed [2, 3] and shown to converge for nonlinear Schrödinger-type problems under suitable conditions [4]. Also, methods based on imaginary time integration have been proposed and suitably accelerated [5]; finally, also methods based on constraint minimization of appropriate (e.g., energy) functionals have been developed [6]. However, for the standing wave discrete nonlinear Schrödinger (DNLS) problem, given the existence of the anti-continuum limit of zero coupling, and its analytical tractability (which provides an excellent initial guess and a starting point for parametric continuations), the Newton method posed on the lattice works extremely efficiently. Note that although the Newton method will be presented herein in a simple parametric continuation format, with respect to the coupling parameter ϵ, it is straightforward to combine it also with pseudo-arclength ideas such as the ones discussed in [7], in order to be able to continue the solution past fold points (and to detect relevant saddle-node bifurcations). Although for one-dimensional problems (and even for two-dimensional discrete problems), it is straightforward to use the direct Newton algorithm with full matrices and then perform the linear stability analysis with full eigensolvers,

K.J.H. Law (✉)
University of Massachusetts, Amherst, MA, 01003, USA
e-mail: law@math.umass.edu

in three dimensions (or e.g., in two-dimensional multicomponent systems), the relevant computations become rather intensive. To bypass this problem, we offer a possibility to perform the Newton method (and the subsequent eigenvalue computations) using sparse iterative solvers and sparse matrix eigensolvers that considerably accelerate the computation, based on the work of Kelley [8].

As concerns the direct numerical integration of the DNLS model, our tool of choice for the time stepping herein will be the fourth-order Runge–Kutta method [1]. Although both lower order methods (such as the efficient split-step Fourier method [9]), as well as higher order methods (including even the eighth-order Runge–Kutta method [10]) have been presented and used in the literature, our use of the fourth-order method, we feel, represents a good balance between a relatively high-order local truncation error (accuracy) and stability properties that allow a relatively high value of the time step ($dt = 10^{-3}$ or higher for most cases of interest here) without violating stability conditions.

All of the above methods (existence, linear stability and direct integration) will be presented by means of Matlab [11] scripts in what follows. The scripts will be vectorized to the extent possible to allow for efficient numerical computation and will also be set up to provide "on the fly" visualization of the relevant parametric continuations (for our bifurcation calculations) and the time-stepping evolution (for our direct integrations). The codes presented below can be found at the website [12].

9.2 Numerical Computations Using Full Matrices

We start with a numerical implementation of the one-dimensional Newton algorithm. We recall that the algorithm assumes the simple form $x^{m+1} = x^m - f(x^m)/f'(x^m)$ for approximating the solution x^s such that $f(x^s) = 0$ (m here denotes the algorithm iteration index). The N-dimensional vector generalization for a lattice of N sites in our one-dimensional problem reads

$$\mathbf{J} \cdot \left(\mathbf{x}^{m+1} - \mathbf{x}^m\right) = -\mathbf{F}(\mathbf{x}^m), \tag{9.1}$$

where $\mathbf{J}$ is the Jacobian of the (vector of) N equations $\mathbf{F}$ with respect to the (vector of) N unknowns $\mathbf{x}$, i.e., $J_{ij} = \partial F_i/\partial x_j$. We are writing Eq. (9.1) as indicated above for a reason, namely to highlight that it is far less expensive to perform the Newton algorithm iteration step as a solution of a linear system (for the vector $\mathbf{x}^{m+1}$), rather than through the inversion of the Jacobian. The vector $\mathbf{x}$ in the computations below consists of the lattice field variables u_n, satisfying the vector of equations

$$F_n = \epsilon\Delta_2 u_n + u_n^3 - u_n = 0, \tag{9.2}$$

where we have taken advantage of the real nature of the one-dimensional solutions (although the algorithm can be straightforwardly generalized to complex solutions as needed in higher dimensions). Once the solutions of Eq. (9.2) are identified to a prescribed accuracy (set below to 10^{-8}), linear stability analysis is performed around the solution as is explained in Chap. 2. The code detailing these bifurcation computations, along with relevant commenting of each step is given below.

```
clear; format long;
% number of sites;
n=100;
% initial coupling; typically at AC-limit eps=0;
eps=0;
% propagation constant; typically set to 1.
l=1;
% field initialization
u1=zeros(1,n);
u1(n/2)=sqrt(l);
u1(n/2+1)=sqrt(l);
% iteration index
it=1;
% lattice index
x=linspace(1,n,n)-n/2;
% continuation in coupling epsilon
while (eps<0.101)
u=zeros(1,n);
while (norm(u-u1)>1e-08)
u=u1;
% evaluation of second difference with free boundaries
sd2=diff(u,2); sd1=u1(2)-u1(1); sdn=u1(n-1)-u1(n);
sd=[sd1,sd2,sdn];
% equation that we are trying to solve
f=-l*u+eps*sd+(u1.^2).*u1;
% auxiliary vectors in Jacobian
ee = eps*ones(1,n);
ee0=ee; ee0(1)=ee(1)/2; ee0(n)=ee(n)/2;
ee1=(-2*ee0-l*ones(1,n)+3*(u1.^2));
%tridiagonal Jacobian
jj1 = spdiags([ee' ee1' ee'], -1:1, n, n);
% Newton correction step
cor=( jj1 \ f' )'; u1=u-cor;
% convergence indicator: should converge quadratically
norm(cor)
end;
% auxiliary vector for stability
ee1=(-2*ee0-l*ones(1,n)+2*abs(u1.^2));
% construction of stability matrix
jj2=spdiags([ee' ee1' ee'], -1:1, n, n);
jj3=diag(u1.^2);
jj4=[ jj2, jj3;
-conj(jj3) -conj(jj2)];
% eigenvalues d and eigenvectors v of stability matrix
[v,d]=eig(full(jj4));
d1=diag(d);
% store solution and stability
u_store(:,it)=u;
d_store(:,it)=d1;
e_store(it)=eps;
% visualize the continuation profiles and stability on the fly
```

```
subplot(2,1,1)
plot(x,u,'-o')
drawnow;
subplot(2,1,2)
plot(imag(d1),real(d1),'o')
drawnow;
% increment indices and epsilon
it=it+1;
eps=eps+0.001;
end;
% save the profiles and other data
save('sol_eps_1d.mat','u_store','d_store','e_store')
```

A prototypical result of the continuation of the above code has been given below (the code addresses the unstable case with two excited sites – the inter-site mode) using the command

```
imagesc(0.001*linspace(0,100,101),linspace(1,100,100)-50,u_store)
```

to spatially visualize the branch for different values of ϵ. Also the dominant stability eigenvalues of this unstable branch are shown (more specifically, λ^2) via the commands

```
d2=sort(real(d_store.^2));
plot(e_store,d2(1,:),e_store,d4(3,:),'b--',e_store,d4(5,:),'b-.')
```

The plots for the solution continuation and its stability are depicted in Fig. 9.1. We now turn to the numerical integration of one of the unstable solutions of the above branch (namely, of the solution for $\epsilon = 0.1$) that we saved at the end of the

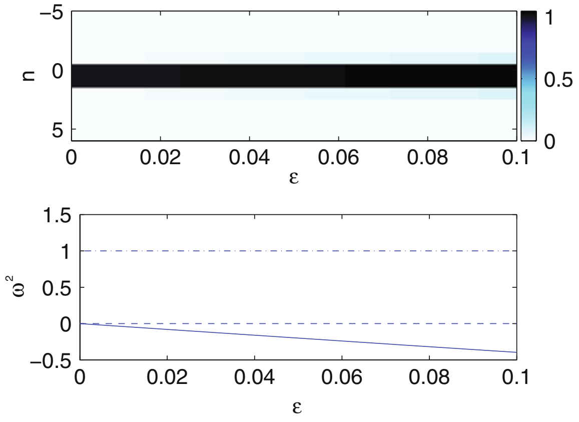

Fig. 9.1 The *top panel* shows the result of continuation as a function of ϵ of the solution profile (shown in contour plot). The *bottom panel* shows the result of the linear stability analysis, indicating the instability of this inter-site mode, through the presence of a negative squared eigenfrequency (*solid line*). The *dashed* pair of eigenvalues at the origin is due to the phase invariance, while the *dash–dotted* pair at 1 (due to the choice of propagation constant $\Lambda = 1$) indicates the lower edge of the continuous spectrum

previous bifurcation code. As indicated above, we use the fourth-order Runge–Kutta method whose four steps and subsequent integration step, we now remind, for the solution of the vector of ordinary differential equations $\mathbf{x}' = \mathbf{f}(t, \mathbf{x})$ with initial condition $\mathbf{x}(t_0) = \mathbf{x_0}$:

$$\mathbf{k}^{(1)} = dt\,\mathbf{f}\left(t^m,\, \mathbf{x}^m\right), \tag{9.3}$$

$$\mathbf{k}^{(2)} = dt\,\mathbf{f}\left(t^m + \frac{dt}{2},\, \mathbf{x}^m + \frac{dt}{2}\,\mathbf{k}^{(1)}\right), \tag{9.4}$$

$$\mathbf{k}^{(3)} = dt\,\mathbf{f}\left(t^m + \frac{dt}{2},\, \mathbf{x}^m + \frac{dt}{2}\,\mathbf{k}^{(2)}\right), \tag{9.5}$$

$$\mathbf{k}^{(4)} = dt\,\mathbf{f}\left(t^m + dt,\, \mathbf{x}^m + dt\,\mathbf{k}^{(3)}\right), \tag{9.6}$$

$$\mathbf{x}^{m+1} = \mathbf{x}^m + \frac{1}{6}\left(\mathbf{k}^{(1)} + 2\mathbf{k}^{(2)} + 2\mathbf{k}^{(3)} + \mathbf{k}^{(4)}\right). \tag{9.7}$$

The commented version of the Matlab script that implements this algorithm for the DNLS equation is given below.

```
% parameters
n=100; eps=0.1;
% load solutions from Newton
load sol_eps_1d.mat
u_num=10
u=u_store(:,u_num)'+1e-04*rand(1,n);
x=real(u); y=imag(u);
% spatial lattice index
sp=linspace(1,n,n)-n/2;
% iteration indices and time step
it=1;
dt=0.001;
it1=1;
it2=1;
% integration up to t=100
while ((it-1)*dt<100)
% computation of second differences and 1st RK integration step
d2y=diff(y,2); ad1y=(y(2)-y(1)); ad3y=(y(n-1)-y(n));
d2x=diff(x,2); ad1x=(x(2)-x(1)); ad3x=(x(n-1)-x(n));
p1=[ad1y,d2y,ad3y]; p2=[ad1x,d2x,ad3x];
k1x=dt*(-eps*p1-y.*(x.^2+y.^2));
k1y=dt*(eps*p2+x.*(x.^2+y.^2));
a=x+k1x/2;
b=y+k1y/2;
% computation of second differences and 2nd RK integration step
d2y=diff(b,2); ad1y=(b(2)-b(1)); ad3y=(b(n-1)-b(n));
d2x=diff(a,2); ad1x=(a(2)-a(1)); ad3x=(a(n-1)-a(n));
p1=[ad1y,d2y,ad3y]; p2=[ad1x,d2x,ad3x];
k2x=dt*(-eps*p1-b.*(a.^2+b.^2));
k2y=dt*(eps*p2+a.*(a.^2+b.^2));
```

```
a=x+k2x/2;
b=y+k2y/2;
% computation of second differences and 3rd RK integration step
d2y=diff(b,2); ad1y=(b(2)-b(1)); ad3y=(b(n-1)-b(n));
d2x=diff(a,2); ad1x=(a(2)-a(1)); ad3x=(a(n-1)-a(n));
p1=[ad1y,d2y,ad3y]; p2=[ad1x,d2x,ad3x];
k3x=dt*(-eps*p1-b.*(a.^2+b.^2));
k3y=dt*(eps*p2+a.*(a.^2+b.^2));
a=x+k3x;
b=y+k3y;
% computation of second differences and 4th RK integration step
d2y=diff(b,2); ad1y=(b(2)-b(1)); ad3y=(b(n-1)-b(n));
d2x=diff(a,2); ad1x=(a(2)-a(1)); ad3x=(a(n-1)-a(n));
p1=[ad1y,d2y,ad3y]; p2=[ad1x,d2x,ad3x];
k4x=dt*(-eps*p1-b.*(a.^2+b.^2));
k4y=dt*(eps*p2+a.*(a.^2+b.^2));
% completion of integration from t -> t+dt
x1=x+(k1x+2*k2x+2*k3x+k4x)/6;
y1=y+(k1y+2*k2y+2*k3y+k4y)/6;
% square modulus profile
uu=x1.^2+y1.^2;
% evaluate energy and (l^2 norm)^2 & visualize the solution every
few steps
if (mod(it,100)==0)
% time counter
tim(it1)=(it-1)*dt;
% square l^2 norm; should be conserved
l2(it1)=sum(uu);
% energy calculation; energy should also be conserved
% although to lower accuracy than l2
gr=[diff(x1,1),0]; gr1=[diff(y1,1),0];
ener(it1)=sum(eps*(gr.^2+gr1.^2)-uu.^2/2);
% plot the solution and its energy and l2 norm
subplot(2,1,1)
plot(sp,uu,'-')
drawnow
subplot(2,1,2)
plot(tim,l2,tim,ener,'--')
drawnow
if (mod(it,1000)==0)
u_store(:,it2)=x1+sqrt(-1)*y1;
it2=it2+1;
end;
it1=it1+1;
end;
it=it+1;
x=x1;
y=y1;
end;
```

The result of the integration for the unstable evolution of the solution is shown in Fig. 9.2 (note that in the initial condition the exact solution was perturbed by a random uniformly distributed noise field of amplitude 10^{-4} in order to seed the

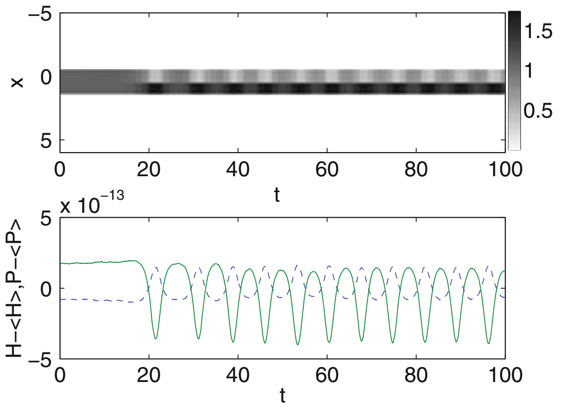

Fig. 9.2 The *top panel* shows the result of the direct integration of the unstable inter-site-centered solution for $\epsilon = 0.1$, with the instability being seeded by a random (uniformly distributed) perturbation of amplitude 10^{-4}. The space–time contour plot of the solution shows how it results into a breathing mode oscillating between the initial condition and a single-site-centered mode. The bottom panel of the figure shows the deviation from the energy H (*solid line*) and the squared l^2 norm P (*dashed line*) conservation. The average energy during the simulation is ≈ -1.007, while the mean of the squared l^2 norm is ≈ 2.199. In both cases, we can see that the deviations from this conservation law are of $O(10^{-13})$

instability). It can be seen how the two-site solution transforms itself into a breathing mode oscillating between a two-site and a single-site solution. On the other hand, the bottom panel of the figure shows the deviation from the relevant (for the DNLS) conservation laws of the energy and the squared l^2 norm. Both of these deviations are of $O(10^{-13})$ as can be seen in the panel while the means of these quantities are of $O(1)$ for the presented simulation. This confirms the good preservation by the proposed scheme of the important conservation laws of the underlying physical model. The above panels are created in Matlab through the use of the commands:

```
subplot(2,1,1)
imagesc(linspace(1,100,100),sp,abs(u_store.^2))
subplot(2,1,2)
plot(tim,l2-mean(l2),'--',tim,ener-mean(ener))
```

Extending the above type of numerical considerations to higher dimensions is conceptually straightforward, although both computationally tedious and obviously far more numerically intensive. We briefly indicate how the above considerations would generalize in two-dimensions (three-dimensional generalizations would naturally extend along the same vein); however, we focus in the next section on how to render these computations more efficient in higher dimensional settings by means of the use of sparse matrix computations and iterative linear solvers.

In the two-dimensional case, the Newton iteration has to extend over a grid of $N \times N$ points, hence the relevant vectors have N^2 elements (more generally N^d for d-dimensional computations). The key realization concerning the performance of operations such as those of Eq. (9.1) with such vectors stemming from higher dimensional grids is that not all points should be treated on the same footing. Taking perhaps the simplest case of vanishing Dirichlet boundary conditions at the edges of our two-dimensional domain, it should be appreciated that while the "inner" $(N-2)\times(N-2)$ nodes of the domain are "regular" points possessing all 4 of their neighbors, there exist an additional $4\times(N-2)$ "edge" points with only 3 neighbors, while the 4 corner points only have 2 neighbors. That is to say, these points should be treated separately regarding both the equation they satisfy and the nature of their corresponding Jacobian elements. Upon this realization, one can treat the two-dimensional grid as a one-dimensional vector whose elements $(1, 1), \ldots, (1, N)$ become elements $1, \ldots, N$, elements $(2, 1), \ldots, (2, N)$ becomes $N+1, \ldots, 2N$, and so on. Then, when constructing the full Jacobian, one should test whether the vector index i running from 1 to N^2 lies at the corners (1, N, N^2-N+1 and N^2), or at the edges $1<i<N$, $N^2-N+1<i<N^2$, $\mathrm{mod}(i, N)=1$ or $\mathrm{mod}(i, N)=0$). This testing can be constructed through appropriate `if` statements, or equivalently by more clever vector manipulations particularly well suited for Matlab. If none of the above happens, then one has all four neighbors (which for the element i are $i+1$, $i-1$, $i+N$, and $i-N$ in the quasi-one-dimensional vector implementation of the grid). Using these considerations one can construct the corresponding Jacobian and perform the same bifurcation computations as above.

As regards the two-dimensional Runge–Kutta simulations, things are in fact a bit simpler, as no Jacobian evaluations are needed. Then assuming that the field (and its second differences) are vanishing at the boundaries, which is a reasonable assumption, for the vast majority of the configurations considered in this book, we can construct the second difference operators in a simple vectorized manner as follows:

```
e=zeros(n,1);
Dxmm = diff([e';x;e'],2,1);
Dxnn = diff([e,x,e],2,2);
Dymm = diff([e';y;e'],2,1);
Dynn = diff([e,y,e],2,2);
```

Using those second differences along the two lattice directions, it is straightforward to again perform the steps used to obtain the intermediate integration vectors $\mathbf{k}^{(j)}$ $j=1, \ldots, 4$, e.g., as follows:

```
k1x=dt*(-eps*(Dymm+Dynn)-y.*(x.^2+y.^2));
k1y=dt*(eps*(Dxmm+Dxnn)+x.*(x.^2+y.^2));
```

Modulo this small modification, the one-dimensional Runge–Kutta realization given above can be essentially immediately transferred into a two-dimensional integrator, upon suitable provision $N\times N$ vector initial conditions $\mathbf{x}_0$, $\mathbf{y}_0$. The same type of considerations can be immediately extended to three-dimensional computations with `Dxmm, Dxnn, Dxll` computed similarly using the `diff` command.

9.3 Numerical Computations Using Sparse Matrices/Iterative Solvers

We discuss two among the many methods for efficiently computing the solution of the Newton fixed point in two dimensions using finite difference derivatives. The more straightforward method is to utilize the sparse banded structure of the Jacobian and the efficiency of the Matlab command "backslash," which will automatically recognize the banded structure and use a banded solver. If memory is not the main consideration, this method is faster. However, one can save a fraction of the memory at the cost of a slightly slower computation utilizing a Newton–Krylov GMRES scheme as implemented by the Matlab script `nsoli` [8]. The memory is minimized by using an Arnoldi iterative algorithm to solve the linear system at each iteration of the Newton method and approximating the Jacobian only in the direction of the Krylov subspace. We note that beyond the standard case outlined here this has far-reaching benefits, particularly when the explicit form of the Jacobian is unknown, or when employing a pseudo-arclength method, for instance, which spoils the banded structure of the Jacobian.

First, we will outline the analogous two-dimensional standard Newton solver (without the tedium of `if` statements) utilizing the structure and sparsity of the Jacobian.

```
clear; format long;
% number of sites;
n=100;
% initial coupling; typically at AC-limit eps=0;
eps=0;
% propagation constant; typically set to 1.
l=1;
% field initialization
u1t=zeros(n,n);
u1t(n/2,n/2)=sqrt(l);
u1t(n/2+1,n/2)=sqrt(l);
% reshape the field into a column vector with real and
% imaginary parts separated for solving
u1=[reshape(real(u1t),n*n,1);reshape(imag(u1t),n*n,1)];
% iteration index
it=1;
% lattice indices as vectors
x=linspace(1,n,n)-n/2;
y=linspace(1,n,n)-n/2;
% lattice indices as matrices
[X,Y]=meshgrid(x,y);
% continuation in coupling epsilon
eps=0;
% define increment
inc = .001;
while (eps<0.101)
eps=eps+inc;
u=zeros(2*n*n,1);
```

```
% BEGIN - see alternative method below
%
while (norm(u-u1)>1e-08)
u=u1;
% fill in the real and imaginary parts of the original 2d field
uur = u(1:n*n);
ur = reshape(uur,n,n);
uui = u(n*n+1:2*n*n);
ui = reshape(uui,n,n);
% evaluation of second difference with zero fixed boundaries
e=zeros(n,1);
Durmm = diff([e';ur;e'],2,1);
Durnn = diff([e,ur,e],2,2);
Duimm = diff([e';ui;e'],2,1);
Duinn = diff([e,ui,e],2,2);
Dur = Durmm + Durnn;
Dui = Duimm + Duinn;
% Other boundary conditions can be implemented similarly,
% for instance use the following for free boundaries
% diff([ur(1,:);ur;ur(n,:)],2,1); ... etc.
% equation that we are trying to solve (vectorized)
f=[reshape(-l*ur+eps*Dur+(ur.^2+ui.^2).*ur,n*n,1); ...
   reshape(-l*ui+eps*Dui+(ur.^2+ui.^2).*ui,n*n,1)];
% auxiliary vectors for linear part of the Jacobian
ee = eps*ones(n,1);
ee0 = ee; ee0(1) = ee(1)/2; ee0(n) = ee(n)/2;
ee1 = -2*ee0;
% tridiagonal linear component for 1d
jj1 = spdiags([ee ee1 ee], -1:1, n, n);
% quick conversion of the 1d $n \times n$ tridiagonal
% into the 2d $n^2 \times n^2$ quintidiagonal via the Kronecker product.
% [ note that the linear component of the Jacobian only needs to
% be constructed once at the beginning of the code, but is left
% here for clarity. ]
jj22 = kron(speye(n),jj1) + kron(jj1,speye(n))-l*speye(n*n,n*n);
% nonlinear component of the Jacobian
n1 = sparse(n*n,n*n);
n2 = sparse(n*n,n*n);
n3 = sparse(n*n,n*n);
n1 = spdiags(uui.^2 + 3*uur.^2,0,n1);
n2 = spdiags(uur.^2 + 3*uui.^2,0,n2);
n3 = spdiags(2*uur.*uui,0,n3);
nn = [n1 n3;n3 n2];
clear n1 n2 n3
% completion of the Jacobian
jj2 = nn + [ jj22, sparse(n*n,n*n);sparse(n*n,n*n), jj22];
clear nn
% Newton correction step
cor = jj2 \ f ;
clear jj2
u1 = u-cor;
% convergence indicator: should converge quadratically
norm(cor)
```

```
end;
% END - see alternative method below
%
% convert back into a complex valued vector
uu1 = u1(1:n*n)+sqrt(-1)*u1(n*n+1:2*n*n);
% and it’s 2d representation
uu2 = reshape(uu1,n,n);
% construct nonlinear part of stability matrix
n1 = sparse(n*n,n*n);
n2 = sparse(n*n,n*n);
n1 = spdiags(uu1.^2,0,n1);
n2 = spdiags(2*abs(uu1).^2,0,n2);
jj2 = jj22 + n2;
jj3=[ jj2, n1;
-conj(n1) -conj(jj2)];
% smallest magnitude (’SM’) 100 eigenvalues d and
% eigenvectors v of stability matrix
% (utilizing a shift in order to improve stability of the
% algorithm)
d1=eigs(jj3+sqrt(-1)*3*speye(2*n*n),100,’SM’)-sqrt(-1)*3;
% store solution and stability
u_store(:,it)=uu1;
d_store(:,it)=d1;
e_store(it)=eps;
% visualize the continuation profiles and stability on the fly
subplot(2,2,1)
imagesc(x,y,abs(uu2).^2)
drawnow;
subplot(2,2,2)
imagesc(x,y,angle(uu2))
drawnow;
subplot(2,2,3)
imagesc(x,y,imag(uu2))
drawnow;
subplot(2,2,4)
plot(imag(d1),real(d1),’o’)
drawnow;
% increment indices and epsilon
it=it+1;
eps=eps+0.001;
end;
% save the data
save(’sol_eps_2d.mat’,’u_store’,’d_store’,e_store’);
```

We should make a few comments here about the code given above. We have illustrated in the existence portion (the Newton method) and the stability section the respective formulations of the complex-valued problem in terms of the real and imaginary components of the field and the field and its complex conjugate (rotation of the former), respectively. The formulations can be interchanged, and it is unnecessary to represent both components in the existence section for real-valued solutions (as with one-dimensional solutions) or equivalently purely imaginary ones, while it is always necessary in the linearization problem because we must consider complex valued perturbations even of real solutions. There are many alternative options in eigs other than "SM," which the interested readers should perhaps explore. On the other hand, the latter is particularly efficient here because the relevant eigenvalues bifurcate from the origin in the anticontinuum limit. The reader should be aware, however, that eigs uses an Arnoldi iterative method to calculate the eigenvalues and the (unshifted) linearization system is singular due to phase invariance. Therefore, some care has to be taken, by a shift or otherwise.

Now, we briefly outline the alternative method using the Matlab script nsoli [8] for the existence portion of the above routine. The reader should note that in addition to the benefits mentioned above, it may be attractive as a black box since it slims down the code. Consult the help file or [8] for more details about it.

```
% setup the parameters for nsoli
max_iter=200;
max_iter_linear=100;
etamax=0.9;
lmeth=1;
restart_limit=20;
sol_parms=[max_iter,max_iter_linear,etamax,lmeth,restart_limit];
error_flag=0;
tolerance=1e-8*[1,1]
% solve with nsoli the equation F(u_)
[u1,iter_hist,error_flag]=nsoli(u1,@(u_)F(u_,eps,n,l),
tolerance,sol_parms);
if error_flag==0
else
disp('there was an error')
iter_hist
break
end
% Now define the function F as a new script F.m in the
% same directory. (Functions cannot be defined within scripts,
% but can be defined within functions, so if the whole code is
% made into a function, then this routine can be embedded.)
function f = F(u,eps,n,l)
ur = reshape(u(1:n*n),n,n);
ui = reshape(u(n*n+1:2*n*n),n,n);
% evaluation of second difference with zero fixed boundaries
e=zeros(n,1);
Durmm = diff([e';ur;e'],2,1);
Durnn = diff([e,ur,e],2,2);
Duimm = diff([e';ui;e'],2,1);
Duinn = diff([e,ui,e],2,2);
Dur = Durmm + Durnn;
Dui = Duimm + Duinn;
% equation that we are trying to solve (vectorized)
f(1:n*n)=reshape(-l*ur+eps*Dur+(ur.^2+ui.^2).*ur,n*n,1);
f(n*n+1:2*n*n)=reshape(-l*ui+eps*Dui+(ur.^2+ui.^2).*ui,n*n,1);
```

9.4 Conclusions

In this section, we have presented an overview of numerical methods used in order to perform both bifurcation, as well as time-stepping computations with the discrete nonlinear Schrödinger equation. We have partitioned our presentation into roughly two broad classes of such methods, namely the ones that use full matrices and direct linear solvers, and ones that use sparse matrices and iterative linear solvers. Our tool of choice for bifurcation calculations has been the Newton method, especially due to the existence of the so-called anti-continuum limit, whereby configurations of interest can be constructed in an explicit form and subsequently continued via either parametric or pseudo-arclength continuation techniques. On the other hand, for the direct dynamical evolution aspects of the problem, we advocated that the use of Runge–Kutta methods affords us the possibility to use relatively large time steps, while at the same time preserving to a satisfactory degree the conservation laws (such as energy and l^2 norm) associated with the underlying Hamiltonian system. The codes given above can be found at the website [12].

References

1. Atkinson, K.: An Introduction to Numerical Analysis. John Wiley, New York (1989) 191, 192
2. Ablowitz, M.J., Musslimani, Z.H.: Opt. Lett. **30**, 2140 (2005) 191
3. Lakoba, T.I., Yang, J.: J. Comp. Phys. **226**, 1668 (2007) 191
4. Pelinovsky, D.E., Stepanyants, Yu.A.: SIAM J. Numer. Anal. **42**, 1110 (2004) 191
5. Yang, J., Lakoba, T.I.: Stud. Appl. Math. **120**, 265 (2008) 191
6. Garcia-Ripoll, J.J., Perez-Garcia, V.M.: SIAM J. Sci. Comput. **23**, 1315 (2001) 191
7. http://en.wikipedia.org/wiki/Numerical_continuation 191
8. Kelley, C.T.: Iterative methods for linear and nonlinear equations. Frontiers in Applied Mathematics, vol. 16, SIAM, Philadelphia (1995) 192, 199, 202
9. Taha, T.R., Ablowitz, M.J.: J. Comp. Phys. **55**, 203 (1984) 192
10. Hairer, E., Norsett, S.P., Wanner, G.: Solving ordinary differential equations. I: Nonstiff Problems. Springer-Verlag, Berlin (1987) 192
11. http://www.mathworks.com 192
12. http://www.math.umass.edu/~law/Research/DNLS_codes/ 192, 203