SUPPLEMENTARY MATERIALS: QGLAB: A MATLAB PACKAGE FOR COMPUTATIONS ON QUANTUM GRAPHS*

ROY H. GOODMAN†, GRACE CONTE‡, AND JEREMY L. MARZUOLA§

This supplement contains two sections. The first, Section SM1, is devoted to demonstrating both the implementation and efficacy of QGLAB on a variety of examples, including stationary problems—eigenvalue problems, the Poisson equation, and the computation and continuation of standing waves—in Section SM1.1 and evolutionary PDE problems in Section SM1.2. All the examples are included as live scripts (MATLAB `.mlx` files) in the directory `source/examples`. The second part, Sec. SM2, contains a complete listing of user-callable function definitions and explicit instructions for their use.

SM1. Extended examples.

SM1.1. Stationary problems.

SM1.1.1. Eigenproblems. Here, we report in greater detail on the accuracy of the eigenproblem calculated in Sec. 4.2. We find the exact eigenvalues from the zeros of the secular determinant. Then we compute the finite-difference approximation with $h = \frac{1}{40}$ and $h = \frac{1}{80}$. The ratio of these is about 4, which shows the method is second order. Finally, we compute the same eigenvalues using the Chebyshev discretization with 30 points on the long edge and 20 on the short edges. Since all errors are less than 10^{-10}, we conclude the accuracy is spectral.

$k = \sqrt{-\lambda}$	λ	$\mathrm{err}_{\tau=\frac{1}{40}}$	$\mathrm{err}_{\tau=\frac{1}{80}}$	ratio	$\mathrm{err}_{\mathrm{Cheb}}$
$\cos^{-1}\left(\frac{1}{12}\left(\sqrt{33}+3\right)\right)$	-0.569	1.687e-05	4.216e-06	4.000	2.195e-12
$\cos^{-1}\left(\frac{1}{12}\left(\sqrt{33}-3\right)\right)$	-3.246	5.486e-04	1.372e-04	4.000	1.177e-12
π	-9.870	5.072e-03	1.268e-03	3.999	1.506e-11
π	-9.870	5.072e-03	1.268e-03	3.999	3.020e-14
$2\pi - \cos^{-1}\left(\frac{1}{12}\left(\sqrt{33}-3\right)\right)$	-20.085	2.100e-02	5.252e-03	3.999	3.830e-11
$2\pi - \cos^{-1}\left(\frac{1}{12}\left(\sqrt{33}+3\right)\right)$	-30.568	4.864e-02	1.216e-02	3.998	1.669e-11
2π	-39.4784	8.111e-02	2.029e-02	3.998	1.847e-13

SM1.1.2. Nonlinear standing waves and bifurcation diagrams.

Computing individual solutions. We begin with an example computing a single solution to the stationary cubic NLS (1.10) on a dumbbell graph:

```
G = quantumGraphFromTemplate('dumbbell');
fcns = getNLSFunctionsGraph(G);
Lambda = -1;
f = @(z)fcns.f(z,Lambda); M = @(z)fcns.fLinMatrix(z,Lambda);
y0 = G.applyFunctionsToAllEdges({0,@(x)sech((x-2)),0});
y = solveNewton(y0,f,M); G.plot(y)
```

*Supplementary material for SISC MS#M162772.
https://doi.org/10.1137/23M1627729

†Department of Mathematical Sciences, New Jersey Institute of Technology, Newark, NJ 07102 USA (goodman@njit.edu).

‡Johns Hopkins University Applied Physics Laboratory, Laurel, MD 20723 USA (gconte23@unc.edu).

§Department of Mathematics, University of North Carolina, Chapel Hill, NC 27599, USA (marzuola@email.cunc.edu).

The function `getNLSFunctionsGraph` defines the discretized version of the nonlinear functional and several of its partial derivatives and assigns them to a structure array called `fcns`. By default, this uses the function $f(z) = 2z^3$ from Eq. (1.10). The user may provide a symbolic function of one variable as an optional argument, and MATLAB will compute all the required partial derivatives symbolically. The Newton-Raphson solver that is iterated to solve the system requires both the functional and its linearization with respect to Ψ. These are stored in two fields `fcns.f` and `fcns.fLinMatrix`, which are functions of two inputs `z` and `Lambda`. The continuation algorithm considers Eq. (1.10) as a function of both Ψ and Λ, but in this first example, we fix $\Lambda = -1$ and consider only Ψ as unknown. In line 4, *anonymous functions* are used to instruct MATLAB to consider them as functions of Ψ alone. We search for a unimodal solution to Eq. (1.10) with $\Lambda = 1$ centered on the central edge of a dumbbell graph, so we prepare an initial guess in line 5 consisting of a hyperbolic secant centered on the central edge and zeros on the two looping edges. The `solveNewton` command finds the standing wave. The result of the `plot` command is shown in Fig. 2.6(a).

For graphs with a large number of edges, generating an initial guess with the approach of line 6 would be impractical, so QGLAB provides a convenient function `applyGraphicalFunction` which applies a function to the coordinate functions used to plot the graph. In Fig. 2.6(b), we find a standing wave on a spiderweb graph, found in the QGLAB template library, using as an initial guess the function $\operatorname{sech}(r)$ where r is the Euclidean distance from the central point to a point on the graph as laid out in two dimensions.

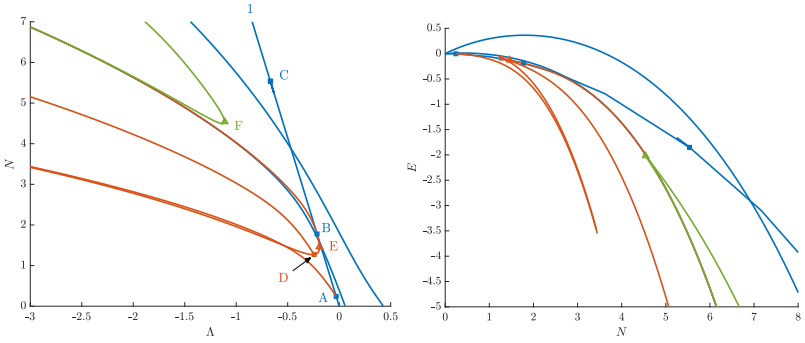
Continuation of solutions. We can learn more about the stationary problem by considering branches of standing waves and their bifurcations than by computing individual solutions. Well-established and sophisticated software packages for such computations include AUTO and MatCont for ODE systems and pde2path for elliptic PDE [SM5, SM6, SM7, SM12]. The capabilities of QGLAB are much more modest but allow for the simple setup and solution to continuation and bifurcation problems on quantum graphs, following branches around folds, detection of bifurcation points, and changing branches at such points.

An extended example of numerical continuation is presented in the live script that is titled `continuationInstructions.mlx`, which presents a computation of a partial bifurcation diagram of the cubic NLS equation on a dumbbell graph in Fig. SM1.1, reproducing a figure from [SM10], which contains far more details and graphs of several of the solutions at various points on the bifurcation diagram.

This figure comprises nine separately-computed curves, each representing dozens of solutions to Eq. (1.10). The curves were initialized in three different ways. The first type, plotted in blue, consists of nonlinear continuations of linear eigenfunctions. We have plotted three such branches but focus on the branch labeled **1**. This branch represents the nonlinear continuation of the null eigenvector of the Laplacian on this quantum graph. The value of Ψ is constant on all solutions on this branch, with

$$\Psi = \sqrt{\frac{-\Lambda}{2}}. \tag{SM1.1}$$

It is straightforward to show that if λ is an eigenvalue of the operator $-\triangle$, then branch **1** has a bifurcation point at $\Lambda = -\lambda/2$ [SM10, SM11]. QGLAB automatically computes the direction in which branches fork from bifurcation points, and the diagram shows two families that emerge from such points. At the points marked **A**, **B**, and **C**, QGLAB has detected bifurcation points on branch **1**, and we have chosen to



(a) (b)

FIG. SM1.1. (a) *A partial bifurcation diagram for the dumbbell graph. The three blue curves are the continuations of linear eigenfunctions. The red curves were computed by continuing from branching bifurcations. The green curve was computed by computing a single large amplitude solution and then continuing it. Branching bifurcations marked with squares and folds with triangles.* (b) *The same diagram, plotted in different variables.*

follow the first two. The branch that bifurcates from branch **A**, which seems to intersect branch **1** transversely, is a pitchfork bifurcation, while the branch that bifurcates from **B** tangentially to branch **1** and extends in both directions is a transcritical bifurcation. This last branch itself has a limit (fold) point at **E** and a pitchfork bifurcation at **D**. The final branch, plotted in green, was generated by first computing a single high-frequency bifurcation with large amplitude pulses on the dumbbell handle and one ring, saving it to a file, and then continuing that solution.

QGLAB stores all the data for branches, bifurcation points, and individual solutions logically and hierarchically and has routines for retrieving and plotting individual solutions and curves of solutions so that the user can largely avoid low-level interactions with the data. By default, it plots the frequency of standing waves versus their power, but it can also plot the energy (1.11), as shown in the right image of Fig. SM1.1.

The nonlinear term in stationary NLS (1.10) can be changed by simply changing the definition of $f(z)$ to any analytic function satisfying $f(0) = 0$ (so that the linearization at zero remains unchanged and the continuation of linear eigenfunctions from zero can be easily computed). In the example `dumbbellcontinuation35.mlx`, we change the right hand side to $f(z) = -2z^3 + 3z^5$ which is defocusing for small values of $|z|$ and focusing for large values. A partial bifurcation diagram for this system is shown in Fig. SM1.2, consisting of three branches that bifurcate from zero in the direction of the eigenfunctions, albeit with a frequency that initially increases with increasing power before changing direction and decreasing. The leftmost branch remains constant in space, and its power increases monotonically along the branch. In contrast, the other two branches have decreasing power as the frequency decreases past a certain point.

Especially interesting is the branch that bifurcates from the point **A** on the middle branch. This middle branch is the continuation of the first excited eigenfunction, which has an odd symmetry about the central point on the dumbbell. At this point, we find a symmetry-breaking pitchfork bifurcation, with two asymmetric branches related by a reflection symmetry. This asymmetric branch continues to the point **B**, at which point it collides again with the same branch from which it bifurcated at **A** and begins retracing its original path. This branch traces out a closed curve in

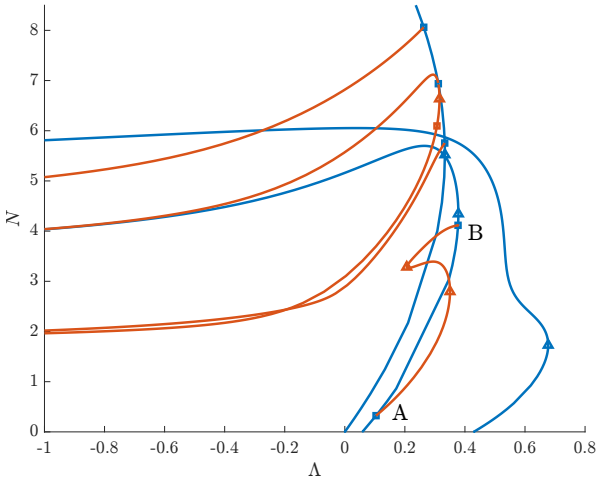


FIG. SM1.2. *A partial bifurcation diagram of the stationary NLS equation on a dumbbell quantum graph with a cubic-quintic nonlinearity.*

solution space, with the sign of the perturbation term flipping each time the branch passes the bifurcation points. Thus, we instructed the continuation program to stop after a finite number of points on the curve are computed by setting the parameter `maxPoints` as described in Sec. SM2.5.

An advantage of the continuation/bifurcation approach is that it illuminates how branches relate to each other. This is well illustrated using the example of a "necklace" quantum graph, also considered by Besse et al. [SM4]. This graph consists of loops alternating with single edges. The necklace graph shown above in Fig. 2.7(a) consists of 54 such alternating pairs, with segments of length 1 and pearls comprised of two edges, each of length $\pi/2$. Fig. 2.7(b) shows a partial bifurcation diagram for the focusing cubic NLS equation on this graph.

We focus on branch **1** and a few branches arising from bifurcations from this branch and its descendants. As in the first example, the constant-valued solution on this branch satisfies Eq. (SM1.1), and bifurcations occur where the frequency is half of an eigenvalue of the linear problem. However, this eigenvalue has a geometric multiplicity of two in this case. In bifurcation theory, the system is said to undergo a *codimension-two* bifurcation at this point. QGLAB has not implemented methods for detecting higher codimension bifurcation points and calculating branches emanating from bifurcations of codimension two or higher. Such methods exist and are implemented in the packages cited above; an approach that obviates the need to calculate higher-order normal forms is the deflated continuation method due to Farrell and collaborators [SM9].

The double-zero eigenvalue at this bifurcation has two orthogonal eigenfunctions plotted in Fig. SM1.3. These may be thought of as the analog of the sine and cosine modes of the second derivative operator on the circle. While any linear combination of these two eigenfunctions is also an eigenfunction, we have chosen the two modes so that one has its maximum at the center of a single strand and the other at the center of a double strand. The nonlinear standing waves that bifurcate from branch **1** at the point **A** do so in the direction of these two eigenfunctions. Close to the bifurcation, the two solution curves are indistinguishable when plotted in these coordinates but separate for more negative frequencies. The standard algorithm that QGLAB uses to detect bifurcations works not by computing all the eigenvalues of the linearization

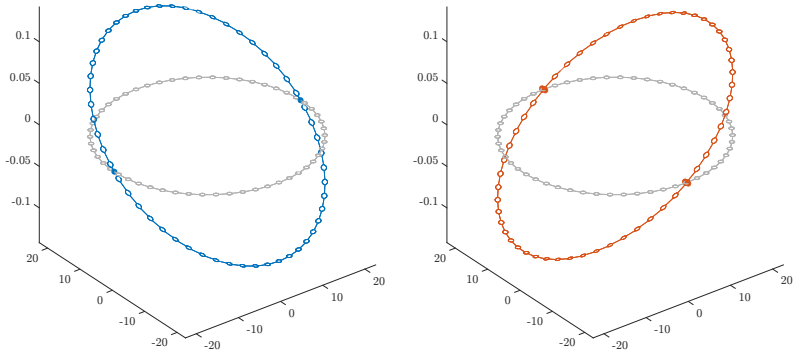


FIG. SM1.3. *The eigenfunctions corresponding to the smallest nonzero eigenvalue on the necklace quantum described in the text. The left eigenfunction has two nodes on "strings" and four local extrema on "pearls", while the right eigenfunction has four nodes on "pearls" and two local extrema on "strings."*

and counting their eigenvalues, which would be slow, but by efficiently calculating the sign of the associated determinant using an LU-decomposition and detecting when it changes. This works efficiently at codimension-one bifurcations but fails at codimension-two bifurcations like this one. As this would predict, the algorithm that detects bifurcations fails to find a bifurcation at **A** and does not compute the branching direction.

The branches **2** and **3** are calculated by first computing a single standing wave with frequency $\Lambda = -4$ and either a single sech-like hump centered on a string or two sech-like humps centered on the two edges on the pearl and then continuing the branches toward the bifurcation point **A**. Branch **4** bifurcates from branch **3** at the point **B**, breaking the symmetry between the two edges of the pearl. By plotting this bifurcation diagram in the same coordinates as in the right image of Fig. SM1.1, we confirm the statement of Ref. [SM4] that this branch represents the ground state at large amplitude. At point **C**, Branch **3** undergoes a second symmetry-breaking bifurcation, giving rise to branch **5**, on which the two-humped standing wave on the pearl moves from the center of the pearl's edges toward either vertex. A similar symmetry bifurcation occurs on Branch **2** at point **D**, giving rise to Branch **6**, along which the standing wave on the string moves away from the string's center and toward a vertex. Branches **5** and **6** appear to converge as Λ is further decreased. Representative standing waves along these five branches of the bifurcation diagram at $\Lambda \approx -4$ are shown in Fig. 2.7(c) above.

Finally, conducting a proper continuation study of standing waves on an infinite necklace is difficult. For a fixed number of pearls, the total width of the standing wave is restricted by the circumference, but in the infinite limit, branches **2** and **3** bifurcate not from the solution of constant amplitude, but from the zero solution, with a width that diverges as the amplitude goes to zero. The limiting behavior exists for the standing waves of the standard cubic NLS problem. However, in that case, a standard method allows the width of the interval to increase, namely using a non-uniform discretization that widens to accommodate the slowing spatial decay rate. Such a trick is unavailable on the quantum graph, where the length scale imposed by the graph's edges precludes this approach.

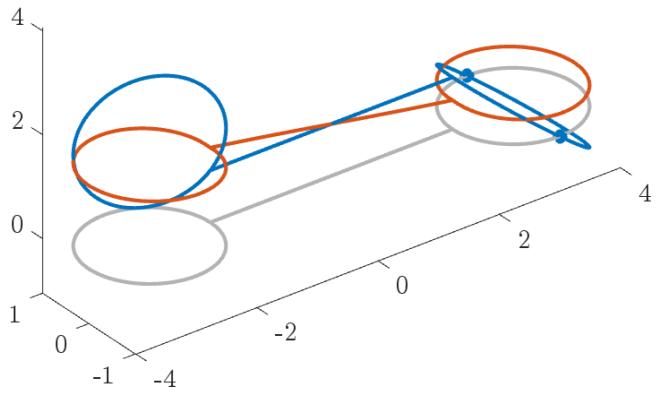


FIG. SM1.4. *The initial (blue) and final (red) states of the heat equation on a dumbbell graph computed using the Crank-Nicholson code in the text.*

SM1.2. Evolutionary PDE. We now discuss the full MATLAB implementation of the heat and sine-Gordon examples constructed in the article.

SM1.2.1. The heat equation. In Sec. 2.4.1 we derived Eq. (2.26) to evolve the solution of the heat equation over one time step. We apply this code to a dumbbell graph in the live script `heatOnDumbbell`. After removing the code for plotting and calculating the conserved total heat, the code reads

```
G = quantumGraphFromTemplate('dumbbell');
y=G.applyFunctionsToAllEdges({@(x)(2-2*cos(x-pi/3)),1,@cos
    });
dt=0.01; tFinal=10; nStep=tFinal/dt;
L0 = Phi.laplacianMatrixWithZeros;
P0 = Phi.interpolationMatrixWithZeros;
LVC = Phi.laplacianMatrixWithVC;
PVC = Phi.interpolationMatrixWithVC;
LPlus =  P0  + (h/2)*L0;
LMinus = PVC - (h/2)*LVC;
for k=1:nStep
    y = LMinus \ (LPlus*y);
end
```

This solution's initial and final states are shown in Fig. SM1.4. The total heat is conserved to twelve digits by this calculation.

SM1.2.2. The sine-Gordon equation. The sine-Gordon equation on the line supports solitons, traveling solutions of the form

$$\psi(x,t) = 4\tan^{-1}\left(e^{(x-ct)/\sqrt{1-c^2}}\right), \quad \text{for any} \quad -1 < c < 1.$$

Following [SM8], we initialize kinks on three edges of the graph formed by the edges of a regular tetrahedron, heading away from their common vertex. We consider two initial conditions: the first with $c = 0.9$ and the second with $c = 0.95$. These are plotted in Fig. SM1.5, with the tetrahedron flattened into the shape of a wheel with three spokes (thus, distance in the plot does not uniformly represent distance on the metric graph). The top row shows the first case, in which the three solitons are

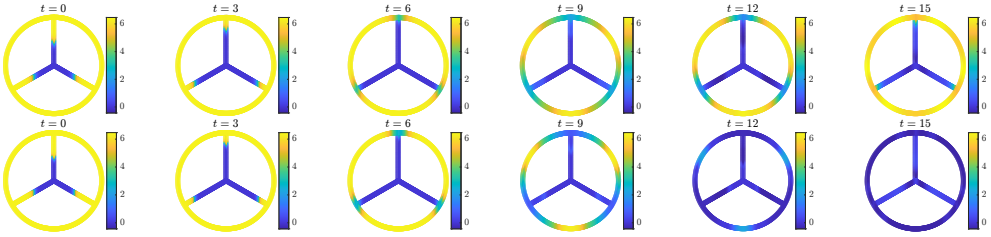


FIG. SM1.5. *Evolution of sine-Gordon solitons propagating along the edges of a tetrahedron (deformed for plotting). (Top) the vertices reflect solitons with* $c = 0.9$ *while (Bottom) those with* $c = 0.95$ *are transmitted.*

```
Quantum-Graphs
├── data
├── documentation
│   ├── Tutorial
│   └── figures_from_the_paper
├── source
│   ├── @quantumGraph
│   ├── chebyshev
│   ├── continuation
│   ├── examples
│   │   ├── chebyshev
│   │   ├── evolution
│   │   └── stationary
│   ├── startup_shutdown
│   ├── templates
│   ├── user
│   └── utilities
├── tests
└── tmp
```

FIG. SM2.1. *Directory structure of QGLAB*

reflected after encountering vertices, while in the second case, the faster solitons can pass through the vertices.

SM2. Function Listing and Detailed Instructions. QGLAB is implemented as a MATLAB *Project*. After starting MATLAB, the user should open the folder titled `Quantum-Graphs`, whose subfolder structure is shown in Fig. SM2.1. Among the files listed in the MATLAB Desktop's `Current Folder` pane is the *project file* `QGobject.prj`, which can be opened by double clicking. This opens the Project Window, adds the necessary QGLAB directories to MATLAB's search path, and changes the plotting preferences needed to render the graphics correctly. To end the QGLAB session, close the Project Window or quit MATLAB. This will remove the QGLAB directories from the search path and restore the user's default plotting preferences, which are held in the folder `tmp` while `QGLAB` is running.

The MATLAB code is contained in the subfolders of the folder `source`. Most importantly, the folder `@quantumGraph` contains the *constructor* file `quantumgraph.m`, which defines the class and initiates an instance, as well as all the class methods, i.e., the functions that act on quantum graph objects. As their first input argument, all MATLAB methods must have a `qg` object `G`. For example, the overloaded eigensolver method `eigs` is defined as `function [v,d]=eigs(G,n)`, where `n` is the number of eigenvalues to calculate. It can be called using either the standard function syntax `[v,d]=eigs(G,n)` or with the preferred syntax for methods `[v,d]=G.eigs(n)`.

SM2.1. The Quantum Graph Constructor. The first step to working with QGLAB is initializing a quantum graph object using its constructor function titled `quantumGraph`. As detailed in Sec. 3.2, it takes three required arguments

- `source` and `target` are two vectors of positive integers. The entries `source(m)` and `target(m)` represent the initial and final nodes of the edge $\mathtt{e}_m$. Thus, these two vectors must be of the length $|\mathcal{E}|$ and each integer m satisfying $1 \le m \le |\mathcal{V}|$ must appear in at least one of the two vectors to guarantee that the graph is connected. MATLAB's `digraph` constructor automatically sorts the edges to avoid confusion, `quantumGraph`checks to make sure the edges are sorted the same way and throws an error if they are not.
- `L` May be either a positive real number or a vector of length $|\mathcal{E}|$ of positive real numbers. If `L` is scalar, the constructor assumes all edges are the same length.

It also may take the following optional arguments

- `Discretization` One may take the values `'Uniform'` (default), `'Chebyshev'`, or `'None'`. If `'None'`, then no discretization is constructed, and the only available method, besides simple methods that query the graph's properties, is `secularDet`, which computes the secular determinant.
- `nxVec` Defines the number of points used to discretize the edges. A vector value gives the number of discretization points on each edge, but if scalar, its behavior depends on the discretization; if `'Uniform'`, then it gives the approximate number of points per unit edge length, while if `'Chebyshev'`, then it gives the number of discretization points on each edge. Default: 20.
- `RobinCoeff` The vector of Robin coefficients α_n in Eq. (1.3). Use the value `NaN` to indicate the Dirichlet boundary condition (1.4). If scalar, apply the same value at all vertices. Default: 0.
- `Weight` The vector of weights w_m in Eq. (1.3). If scalar, apply the same value at all vertices. Default: 1.
- `nodeData` The vector of nonhomogeneous vertex terms ϕ_n in the Poisson problem (2.10c). If scalar, apply the same value at all vertices. Default: 0.
- `plotCoordinateFcn` The handle of a function defining the layout of the edges and vertices for plotting. Associates coordinate arrays `x1`, `x2`, and (optionally) `x3` to each edge and to the vertices. If left unset, then plotting is not possible. It can be set later using the function `addPlotCoordinates`.

The constructor runs several checks on the inputs to ensure they are consistent and meaningful, returning descriptive error messages if these checks fail.

SM2.2. Properties of a `quantumGraph` object. Many of the properties of a designated `quantumGraph` object are detailed in Sec. 3.2, a complete list is given here, filling in some additional details

- `qg` The `digraph` object, consisting of `Edge` and `Node` tables, each of which has the additional required fields described in Sec. 3.2 as well as the optional fields `x1`, `x2`, and `x3` used for plotting.

- `discretization` A string labeling the discretization type is used to choose between uniform and Chebyshev algorithms.
- `wideLaplacianMatrix` The Laplacian matrix $\mathbf{L}_{\text{int}}$, with discretized boundary condition rows at the bottom, defined in Eq. (2.12) and illustrated by the two upper matrix blocks in Figs. 2.2(b) and 2.4(b).
- `interpolationMatrix` The matrix $\mathbf{P}_{\text{int}}$ that interpolates from the extended grid to the interior grid as defined in Eq. (2.13), as illustrated by the two upper matrix blocks in Figs. 2.2(c) and 2.4(c).
- `discreteVCMatrix` The matrix $\mathbf{M}_{\text{VC}}$ containing the discretization of the vertex conditions, as defined in Eq. (2.12), (2.13) and illustrated by the two lower matrix blocks in Figs. 2.2(b) and 2.4(b).
- `nonhomogeneousVCMatrix` The matrix $\mathbf{M}_{\text{NH}}$ defined in Eq. (2.14) used to define nonhomogeneous terms in the vertex condition to the correct rows.
- `derivativeMatrix` The square first derivative matrix which does not include boundary conditions. This is used for calculating integrals, including the energy and momentum, which may or may not be conserved based on the vertex conditions.

SM2.3. Methods defined for a `quantumGraph` object.

SM2.3.1. MATLAB `digraph` methods overloaded for `quantumGraph` objects. MATLAB features many functions for analyzing, querying, and manipulating directed graphs. The command `indegree(G,1)` returns the incoming degree of the vertex $\mathtt{v}_1$ of a graph `G`. This could be applied to the `qg` field of a quantum graph Φ by using the command `indegree(G.qg,1)`, but it is preferable in object-oriented programming to *overload* this function so that can be applied directly as `indegree(Phi,1)` Several other low-level directed graph functions have been similarly overloaded:

- `Edges, Nodes, indegree, outdegree, numedges, numnodes, rmnode.`

SM2.3.2. Other `quantumGraph` methods. The following provide directed graph related functionality not in MATLAB's digraph toolbox:

- `source, target, follows, sharednode, incomingedges, outgoingedges, isleaf.`

The following functions query specific properties of quantum graphs, edges, or vertices:

- `nx, dx, weight, L, robinCoeff, isUniform, isChebyshev, isDirichlet.`

The following are utilities for working with `quantumGraph` objects:

- `addPlotCoords` Given a user-provided script defining the plotting coordinates `x1`, `x2`, and, optionally, `x3`, runs the script and associates the coordinates to both the edge and vertex tables.
- `graph2column` and `column2graph` transfer data back and forth between the edge-vertex representation and a single-column vector. The latter function uses the discretized vertex conditions to interpolate the data at the vertices.
- `applyFunctionToEdge` The call `G.applyfunctionToEdge(fhandle,m)` applies the function represented by the function handle `fhandle` to the edge $\mathtt{e}_m$ and stores the result in `G.Edges.y{m}`. If `fhandle` is a number `c`, then the output `G.Edges.y{m}` will be a constant-valued vector of the appropriate length.
- `applyFunctionsToAllEdges` If `handleArray` is a cell array containing $|\mathcal{E}|$ function handles and constants, this function applies `applyFunctionToEdge` to each function/constant and edge in the quantum graph. If an output argument is specified, then `graph2column` is used to assign the function to a column vector.
- addPotential Adds a potential to the graph structure using the same syntax as `applyFunctionsToAllEdges`.

- `applyGraphicalFunction` This applies a function, input as its function handle, to the plotting coordinates `x1`, `x2`, and (optionally) `x3` defined for each function and edge. This convenience function creates initial guesses for the nonlinear standing wave solvers.
- `addEdgeField` and `addNodeField` can be used to assign other fields to the `Edge` and `Node` tables.

The following functions perform mathematical operations on `quantumGraph`objects, automatically choosing the appropriate program for the discretization method used:

- `integral` Computes the weighted integral $\int_\Gamma \Psi \, \mathrm{d}x = \sum_{m=1}^{|\mathcal{E}|} w_m \int_{\mathbf{e}_m} \psi_m(x) \, \mathrm{d}x$.
- `norm` Uses `integral` to compute the L^p norm (1.5).
- `dot` Uses `integral` to compute the L^2 inner product (1.6).
- `energyNLS` Uses `integral` to compute the NLS energy (1.11).
- `eigs` Computes n eigenvalues closest to zero.
- `secularDet` Computes the real-valued secular determinant defined briefly in Sec. 1.2 using the MATLAB Symbolic Mathematics Toolbox. This works for all the boundary conditions discussed in this article but requires the edge weights to satisfy $w_m \equiv 1$.
- `solvePoisson` Solves the Poisson problem (2.10).

The following functions are for visualizing `quantumGraph` objects:

- `plot` The call `G.plot` plots the data currently stored in the yentries of the `Edges` and `Nodes` tables, using the coordinates stored in the `x1`, `x2` and `x3` table entries. If `x3` is not defined, then it plots the function in three dimensions over the skeleton of the graph. If it is defined, then the function is plotted in false color. The call `G.plot(z)` first calls `G.column2graph(z)` and then plots.
- `pcolor` Plots the function in false color on the quantum graph in two dimensions. It is useful for visualizing highly complex graphs, as seen by comparing the two plots of MATLAB's `peaks` function defined over the edges of a randomly generated Delaunay triangulation, shown in Fig. SM2.2.

```
G = delaunaySquare('n',8);
f = @(x1,x2)peaks(6*x1-3,6*x2-3);
G.applyGraphicalFunction(f);
G.plot; figure; G.pcolor
```

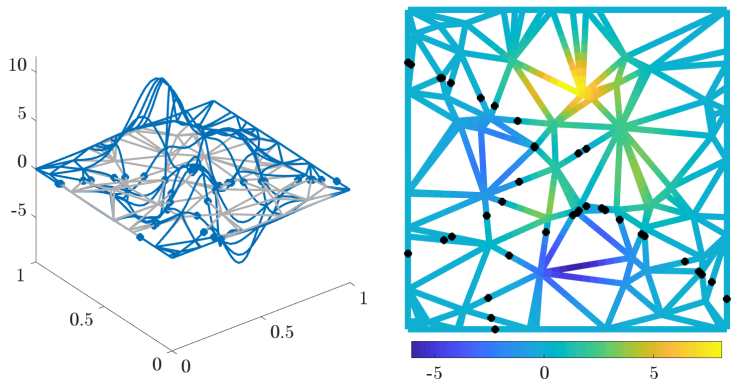


FIG. SM2.2. *Visualization of a function defined on a random graph using (left)* `plot` *and (right)* `pcolor`*, where zeros are indicated with black dots.*

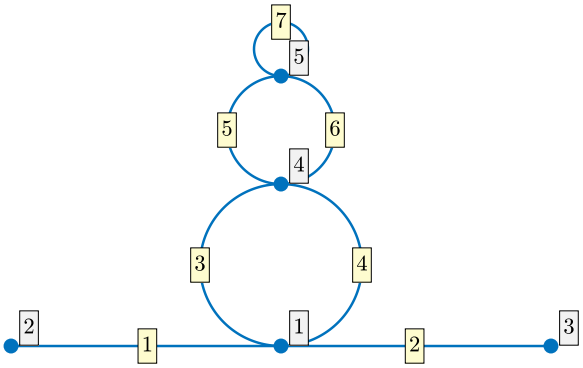


FIG. SM2.3. *The default* `bubbleTower` *quantum graph, with five vertices and seven edges.*

- `spy` The call `G.spy` uses the MATLAB `spy` function to plot the nonzero entries in the three matrices `G.wideLaplacianMatrix`, `G.interpolationMatrix`, and `G.nonhomogeneousVCMatrix`.
- `animatePDESolution` Given a vector of times `t` and an array `u` whose columns give the numerical solution to a PDE at those times, animates the solution, taking special care that the viewing axes are fixed throughout the visualization. Automatically uses false color to plot graphs with a three-dimensional layout. To animate a PDE solution using false color on a two-dimensional layout, use `animatePDESolution2DColor`.

Additional programs not called by the end-user exist, which we do not document.

SM2.4. The template library. The package features a library of graphs, many of which have been studied in the quantum graph literature, which is stored in the folder `source/templates`. Their use is demonstrated in the live script that is titled `templateGallery.mlx`. These fall into a few groups. Almost all depend on several user-provided parameters for which default values are provided.

Individual graphs. Several simple graphs are provided in the template library and are called using the command `G=quantumGraphFromTemplate(tag,varargin)`, where `tag` is the name of the template and `varargin` is used by MATLAB to indicate a variable-length input argument list, and is here used to enter using the same key-value syntax as the `quantumGraph` command. The graph produced by running:

```
G=quantumGraphFromTemplate('bubbleTower','L',10,'circumferences',[6 4 2]*pi)
```

is shown in Fig. SM2.3. The default graph in this family has five vertices and seven edges. Bubble tower graphs with infinite-length base edges have featured extensively in the quantum graph literature as examples where one can still find a ground state even though a certain graph topology condition is satisfied by this family that would normally preclude the existence of a ground state, see [SM1, SM2, SM3]. The underlying symmetry of the construction here is crucial to the analysis.

The `quantumGraphFromTemplate` function calls two separate functions

- **A template function**, here `bubbleTower.m`, that builds the quantum graph, setting the lengths of the two straight line segments to 10 and the circumferences of the three bubbles to $[6\pi, 4\pi, 2\pi]$, setting the discretization, and building the necessary matrices.
- **A plot coordinates function**, here `bubbleTowerPlotCoords.m`, that places the vertices at locations consistent with the above-defined lengths. In this

example, two edges are laid out as line segments, created using the command `straightEdge`, four edges are laid out as semicircular edges by using the command `semicircularEdge`, and there is one circular edge, created using the command `circularEdge`. A fourth function `circularArcEdge` can connect two nodes by a circular arc subtending a central angle `theta`.

SM2.4.1. Two-dimensional lattices. The following templates exist to create two-dimensional lattices. All have default values and can be customized to change the number of cells per side. These programs are called directly and set plotting coordinates without calling `quantumGraphFromTemplate`.

- `rectangularArray` creates a rectangular array. By default, the sides have unit length but can be customized.
- `triangularArray` creates a triangular array. The unit cell is an equilateral triangle by default, but the period vectors can be customized.
- `hexagonalArray` creates a hexagonal array, forming a parallelogram, the default shown in Fig.
- `hexGrid` creates a rectangular array of hexagons.
- `hexGridPeriodic` identifies the left edge with the right and the top edge with the bottom to create a periodic array.
- `hexOfHexes` A hexagonal array of hexagons.
- `triangularArray` A triangular array.

Three-dimensional geometric templates. The program `solidTemplate` constructs quantum graphs whose vertices and edges are the vertices and edges of geometric solids, including the five Platonic solids (tetrahedron, cube, octahedron, dodecahedron, and icosahedron), as well as the cuboctahedron, which has 24 edges and 12 vertices, and the buckyball (or truncated icosahedron) which has 90 edges and 60 vertices. This is called directly and sets up the plot coordinates. Sec. 3.3.1 gives an example of constructing a tetrahedron.

SM2.5. Continuation and bifurcation routines. The live script that is titled `continuationInstructions.mlx` in the `documentation` directory uses all the following subroutines in the given order after constructing a `quantumGraph` object named `Phi`. We refer to line numbers in this live script to describe the steps taken to compute the bifurcation diagrams. To run the continuation software, the user must use a template from the `source/templates` or create one themselves, including a properly named function to create the plotting coordinates. We will assume that the template's name is stored in a variable named `tag`. In the example `tag='dumbbell'`. As explained below, the results of the computation will be stored in the directory `dataDir='data/dumbbell/001'` with the trailing number incremented each time a bifurcation diagram is created. Each computed branch of solutions is stored in its own subdirectory, with consecutively labeled names, beginning `branch001`, etc. Most of the programs given below add a line to a log file named `logfile.txt` that resides in the data directory.

- `makeContinuationDirectory` After initializing the discretized quantum graph on which families of solutions are to be computed, create a sequentially named directory to hold the data; see line `5`. Saves a file `template.mat` containing the `qg` object.
- `saveEigenfunctions` Calculate some eigenvalues and eigenfunctions of the Laplacian matrix and save them to the data directory with names `lambda.001` and `eigenfunction.001`.

- `saveNLSFunctionsGraph` Saves a file named `fcns.mat` to the data directory. This file contains one variable: a structure `x` whose fields contain a function handle to the discretized form of Equation (1.10), as well as several derivatives of this function and the antiderivative of the nonlinearity, used in computing the energy.
- `continuerSet` This function sets several parameters the continuation algorithms use. It assigns them to a structure, usually named `options`, which is then passed to the various `continueFrom` programs described below. It takes as input a sequence of name-value pairs, imitating the programs `odeset` and `optimset` used in MATLAB's ODE and optimization routines. The parameters it sets are:
  - `maxTheta` The maximum angle, in degrees, between two consecutive segments on a branch of solutions. Default: 4°.
  - `minNormDelta` The minimum step length below which the continuation solver does not attempt to refine the branch further. Default: 10^{-3}.
  - `beta` The weight in the inner product defined by
    $$\left\langle \Phi_1(x)e^{i\Lambda_1 t}, \Phi_2(x)e^{i\Lambda_2 t}\right\rangle = \langle \Phi_1, \Phi_2\rangle + \beta\langle \Lambda_1, \Lambda_2\rangle,$$
    used in defining angles and distances in the above two variables. Default: 0.1.
  - `NThresh` Threshold for the power N, i.e., the squared L^2-norm, so the continuation routine terminates when this value is crossed. Default: 4.
  - `LambdaThresh` Threshold for the frequency Λ. The continuation routine terminates when this value is crossed. Default: -1.
  - `maxPoints` The maximum number of points to compute on a given branch. Default: 999.
  - `saveFlag` A boolean variable. If true, then data is saved to files. Default: `true`.
  - `plotFlag` A boolean variable. If true, then data is plotted to screen. Default: `true`.
  - `verboseFlag` A boolean variable. If true, then some information is printed on the MATLAB Desktop. Default: `true`.
- Four continuation programs that are initiated from different starting points.
  - `continueFromEig` Compute a branch of stationary solutions that bifurcates from $\Psi = 0$ with a frequency given by an eigenvalue, in the direction of an eigenfunction, using the data saved by the above command `saveEigenfunctions`; cf. lines `13-15` of the live script.
  - `continueFromBranchPoint` Compute a branch of stationary solutions that bifurcates from a branch point. While computing a curve of solutions, the continuation routines monitor for branching bifurcations (pitchfork and transcritical, which are mathematically equivalent in the pseudo-arclength formulation). When it detects a bifurcation between two computed solutions, it computes the exact frequency at which the bifurcation occurs and the solution at the bifurcation point.
  - `continueFromSaved` Continue from a previously-computed solution to the stationary computed using `saveHighFrequencyStandingWave` (called here), which computes and saves a solution with an initial guess built from sech-like functions defined on the edges,
    `saveHighFrequencyStandingWaveGraphical`,
    which computes a solution based on an initial guess that places a "bump"

somewhere on the graph defined by its plotting coordinates or a user-written function.
On line `35` of the example, a solution with positive sech pulses of edges 1 and 2 of the dumbbell is saved to files:
`savedFunction.001` and `savedFunction.001`
in the folder `data/dumbbell/001`. A branch continuing from this solution is computed at line `38`.

- `continueFromEnd` Extends a previously-computed branch.

- `bifurcationDiagram` Draws a bifurcation diagram from the data in a given directory and its subdirectories. By default, it plots the frequency on the x-axis and the squared L^2-norm on the y-axis, but these defaults can be overwritten.
- `rmBranch` Removes the subdirectory containing a given branch from the bifurcation diagram directory.
- `plotSolution` Plots a single solution from a given diagram and branch.
- `animateBranch` Animates how the individual solutions change as a branch of the bifurcation diagram is traversed.
- `addComment` Adds a string to the log file `logfile.txt` in the given directory.

We examine the files contained in the directory `branch001`, which was created online `13` of the live script by `continueFromEig`.

- `PhiColumn.xxx` Where `xxx` is a three-digit integer n. The nth solution on the branch.
- `NVec`, `LambdaVec`, and `energyVec` Column vectors containing the squared L^2-norm, the frequency, and the energy, which are the three variables that can be plotted using the `bifurcationDiagram` program. The nth entry in each vector corresponds to the nth solution in the previous bullet point.
- `k` The number of `PhiColumn` files and the length of the vectors of integrals.
- `initialization` A one-word text file denoting which of the four continuation programs `coninueFromXXX` was used to initialize the branch, in this case `Eigenfucntion`.
- `eignumber` The number of the eigenfunction from which the solution was continued.
- `options.mat` The options structure set by the `continuerSet` program.
- `bifTypeVec` A column vector of integers, with the value 0 if solution n is a regular point on the branch, the value 1 at branching bifurcations, and the value -1 at folds.
- `phiPerturbationXXX.mat` and `LambdaPerturbationXXX.mat` Here `xxx` is a three-digit number at which a branching bifurcation has been detected, and the files contain the directions in which the new branch points from the bifurcation location, used by the function `continueFromBranchPoint`.

SM2.6. Other folders.

- `data` An empty folder where the continuation routines store the data they produce.
- `documentation` Contains live scripts demonstrating the main features entitled `quantumGraphRoutines.mlx`, `continuationInstructions.mlx`, and `continuationInstructionsChebyshev.mlx`.
- `source/chebyshev` Contains many programs used to construct the Chebyshev discretization.
- `source/examples` Contains example programs sorted into three further subfolders:
  - `source/examples/chebyshev` Contains examples involving the Chebyshev discretization, all of which are minor modifications of examples from the `stationary` folder.
  - `source/examples/evolution` Examples illustrating the solution to time-dependent problems.

  - `source/examples/stationary` Examples of time-independent problems: eigenproblems, Poisson problems, and continuation problems.
- `source/startup_shutdown` Contains programs that are run upon starting up and shutting down QGLAB.
- `source/user` An empty folder intended to give end-users a place to store code they write without mixing it with package code.
- `source/utilities` Some utilities used for file management and formatting plots.
- `tmp` A temporary folder created at startup and removed at shutdown, where the user's plotting preferences are stored to be automatically restored upon shutting down `quantumGraph`.

REFERENCES

[SM1] R. Adami, *Ground states for NLS on graphs: a subtle interplay of metric and topology*, Math. Modell. Nat. Phenom., 11 (2016), pp. 20–35.

[SM2] R. Adami, E. Serra, and P. Tilli, *Negative energy ground states for the L^2-critical NLSE on metric graphs*, Commun. Math. Phys., 352 (2017), pp. 387–406.

[SM3] R. Adami, E. Serra, and P. Tilli, *Nonlinear dynamics on branched structures and networks*, Riv. Math. Univ. Parma, 8 (2017), pp. 109–159.

[SM4] C. Besse, R. Duboscq, and S. Le Coz, *Numerical simulations on nonlinear quantum graphs with the GraFiDi library*, SMAI J. Comput. Math, 8 (2021), pp. 1–47.

[SM5] A. Dhooge, W. Govaerts, and Y. A. Kuznetsov, *MATCONT: a MATLAB package for numerical bifurcation analysis of ODEs*, ACM T. Math. Software, 29 (2003), pp. 141–164.

[SM6] A. Dhooge, W. Govaerts, Y. A. Kuznetsov, H. G. E. Meijer, and B. Sautois, *New features of the software MatCont for bifurcation analysis of dynamical systems*, Math. Comp. Model. Dyn., 14 (2008), pp. 147–175.

[SM7] E. J. Doedel, B. E. Oldeman, A. R. Champneys, F. Dercole, T. Fairgrieve, Y. Kuznetsov, B. Sandstede, X. Wang, and C. Zhang, *AUTO*-07P: Continuation and bifurcation software for ordinary differential equations, tech. report, Concordia University, 2007.

[SM8] D. Dutykh and J.-G. Caputo, *Wave dynamics on networks: Method and application to the sine-Gordon equation*, Appl. Numer. Math., 131 (2018), p. 54.

[SM9] P. E. Farrell, A. Birkisson, and S. W. Funke, *Deflation techniques for finding distinct solutions of nonlinear partial differential equations*, SIAM J. Sci. Comput., 37 (2015), pp. A2026–A2045.

[SM10] R. H. Goodman, *NLS bifurcations on the bowtie combinatorial graph and the dumbbell metric graph*, Discrete Contin. Dyn. Syst. A, 39 (2019), pp. 2203–2232.

[SM11] J. L. Marzuola and D. E. Pelinovsky, *Ground state on the dumbbell graph*, Appl. Math. Res. eXpress, 1 (2016), pp. 98–145.

[SM12] H. Uecker, D. Wetzel, and J. D. Rademacher, *pde2path–A MATLAB package for continuation and bifurcation in 2D elliptic systems*, Numer. Math. Theory Methods Appl., 7 (2014), pp. 58–106.