arXiv:2401.00561v1 [math.NA] 31 Dec 2023

QGLAB: A MATLAB PACKAGE FOR COMPUTATIONS ON QUANTUM GRAPHS*

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

Abstract. We describe QGLAB, a new MATLAB package for analyzing partial differential equations on quantum graphs. The software is built on the existing, object-oriented MATLAB directed-graph class, inheriting its structure and adding additional easy-to-use features. The package allows one to construct a quantum graph and accurately compute the spectrum of elliptic operators, solutions to Poisson problems, the linear and nonlinear time evolution of a variety of PDEs, the continuation of branches of steady states (including locating and switching branches at bifurcations) and more. It uses a unified framework to implement finite-difference and Chebyshev discretizations of differential operators on a quantum graph. For simplicity, the package overloads many built-in MATLAB functions to work on the class.

1. Introduction.

This paper introduces and provides operating instructions for QGLAB, a software package written in MATLAB for computations on quantum graphs, with a special emphasis on solutions of the cubic nonlinear Schrödinger equation (NLS). Many older examples of these remarkable objects—networks of one-dimensional edges interacting via a set of natural vertex conditions—can be found in the literature. However, the modern interest in the subject seems to begin with an analysis of their spectral statistics in the work [54], whose authors seem to have coined the term "quantum graphs." The spectral theory and properties of operators on quantum graphs have further developed rapidly from the pioneering work [40] in which quantum graphs were realized as the limits of quantum equations on thin wire-like domains. See also the works [41, 42, 49, 71] in this direction. Quantum graphs provide effectively one-dimensional model equations that enable explicit calculations that serve as a backbone for representing geometric and spectral theoretic properties of much more complicated higher-dimensional quantum models. An explosion of results exploring their properties has followed. For further introduction to the history and applications of quantum graphs, we recommend references [15, 17].

Numerical packages are essential to facilitate further study for many of the usual reasons: e.g., making progress on larger-scale problems and those with nonlinearities and time dependence all depend on intuition built from numerical experimentation and visualization. This project provides a set of high-level tools that allow users to quickly and easily set up, solve, and visualize the solutions to problems posed on quantum graphs.

At the heart of the package is the definition of a quantum graph class, built on top of MATLAB's directed graph class. While we have striven to write a general-purpose software package for quantum graph computations, the direction of development has been guided by two classes of problems of research interest to its authors:

1. Efficient computation of bifurcation diagrams: standing waves of NLS occur

---

[*]

Funding: R.H.G. acknowledges support from the NSF through NSF Grant DMS-2206016 G.C. and J.L.M. acknowledge support from the NSF through NSF CAREER Grant DMS-1352353, NSF Grant DMS-1909035 and NSF FRG grant DMS-2152289.

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

[‡] Johns Hopkins University Applied Physics Laboratory, Baltimore, MD, (gracie.conte@jhuapl.edu).

[§] Department of Mathematics, University of North Carolina, Chapel Hill, NC (marzuola@email.unc.edu).

along one-parameter families or *branches* rather than at isolated points. To understand the solutions' parameter dependence, it is essential to compute such branches using continuation methods. Further, branches may cross, and the stability of solutions change at isolated *bifurcation points.* The package can detect the most common types of bifurcation points and switch branches at these points; it has been used for this purpose in the publications [18, 47].

2. Spectral spatial accuracy and high-order time stepping: one of the goals, to be described in future publications, is to compute time-periodic and time-relative-periodic orbits of the full time-dependent NLS on a compact quantum graph.

We wrote the code to allow the user to specify the quantum graph object at a high level and insulate them from having to understand the detailed workings of the software. We have overloaded many built-in MATLAB commands for basic calculations and plotting to accomplish this objective. A similar package called *GraFiDi*, written in Python, is described in [20]. While the two packages share many features, QGLAB also includes spectrally accurate solvers, symbolic computation, and numerical continuation capabilities. To achieve spectral accuracy while satisfying sometimes-complex conditions at the vertices, QGLAB adopts several ideas developed for the *Chebfun* package from [35], although QGLAB does not adapt the order of the polynomials used to achieve maximum precision. It would be straightforward, but by no means quick to implement the QGLAB architecture using *Chebfun*, but we have not chosen this route—see [59] for an early implementation of similar ideas. The package uses MATLAB's many built-in graph capabilities, providing a simple and robust user interface.

1.1. Defining a quantum graph. A quantum graph Γ consists of a directed metric graph, considered as a *complex* of edges, on which a function space and differential operators are defined. To be more specific, we define the graph $\Gamma = (\mathcal{V}, \mathcal{E})$ as a set of vertices $\mathcal{V} = \{\mathtt{v}_n,\, n = 1, \ldots, |\mathcal{V}|\,\}$ and a set of directed edges $\mathcal{E} = \{\mathtt{e}_m = (\mathtt{v}_i \to \mathtt{v}_j),\, m = 1, \ldots, |\mathcal{E}|\,\}$ and to each edge assign a positive length ℓ_m and impose upon the edge a coordinate x that increases from 0 to ℓ_m as the edge is traversed from the $\mathtt{v}_i$ to $\mathtt{v}_j$. In general, we may consider the lengths of some or all of the edges infinite, in which case that edge will be connected to a single vertex. For computation, however, we require finite-length edges, so we will not discuss the case of unbounded edges further. Define the *degree* of vertex $\mathtt{v}_n$, denoted d_n to be the number of edges that include that vertex as an initial or final point, counting twice if an edge connects the vertex to itself.

We consider *weighted* graphs, for which there is associated to each edge a positive weight w_m used to define fluxes and, thus, boundary conditions. If the graph is taken to be a model of a network of pipes or wires, we can think of the weight as proportional to the cross-sectional area of a given pipe or wire.

An example graph, with $|\mathcal{V}| = 4$ vertices and and $|\mathcal{E}| = 7$ edges is shown in Fig. 1.1. The degrees of the four vertices are $d_1 = 5$, $d_2 = 4$, $d_3 = 4$, and $d_4 = 1$. Under our definition, there is no problem in taking multiple edges with the same beginning and ending vertex, as seen between vertices $\mathtt{v}_1$ and $\mathtt{v}_2$ in the figure.

A function $\Psi(x)$ defined on Γ can be thought of most simply as a collection of functions defined on each of the edges $\Psi|_{\mathtt{e}_m} = \psi_m(x)$, and a Laplace operator on the graph is defined by

$$\triangle|_{\mathtt{e}_m} = \frac{\mathrm{d}^2}{\mathrm{d}x^2}, \quad \text{for} \quad 0 < x < \ell_m,$$

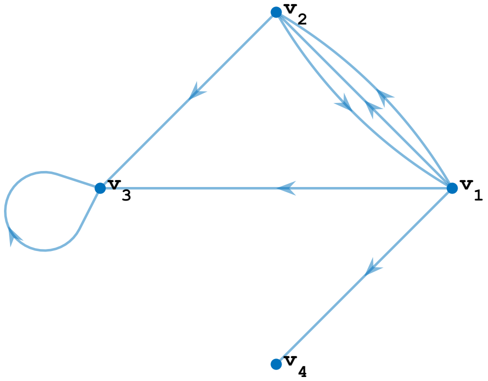


Fig. 1.1: A directed graph with four vertices and seven edges.

subject to appropriate compatibility conditions at the vertices. Of course, defining a Laplacian requires defining a function space. The graph and vertex conditions define a quantum graph, and we are most concerned with vertex conditions giving rise to a self-adjoint operator. We have implemented *weighted Robin-Kirchhoff* vertex conditions and *Dirichlet* boundary conditions, although these are not the most general self-adjoint boundary conditions; see e.g. [17].

The first $d_n - 1$ equalities that define the vertex condition at the vertex $\mathtt{v}_n$ are simply continuity across the vertex. Let $\mathcal{V}_n$ be the set of all edges adjacent to the vertex, double-counting in the case of self-directed edges. Continuity allows us to define a function value at the vertex by

$$\Psi(\mathtt{v}_n) \equiv \psi_i(\mathtt{v}_n) = \psi_j(\mathtt{v}_n), \forall \mathtt{e}_i, \mathtt{e}_j \in \mathcal{V}_n. \tag{1.1}$$

The weighted Robin-Kirchhoff flux condition is then

$$\sum_{\mathtt{e}_m \in \mathcal{V}_n} w_m \psi_m'(\mathtt{v}_n) + \alpha_n \Psi(\mathtt{v}_n) = 0, \tag{1.2}$$

where the derivative is, in all cases, taken in the direction *pointing away* from the vertex. In the case $w_m \equiv 1$, $\alpha_n \equiv 0$, this reduces to the Neumann-Kirchhoff vertex condition, which is the natural generalization of the Neumann boundary condition on a line segment. Interpreting the equation $\triangle \Psi = 0$ as describing a steady state of the heat equation, the Neumann-Kirchhoff vertex condition states that there is zero net heat flux into the vertex. Introducing non-unit weights w_m then generalizes this condition to say that the flux on each edge is proportional to the weight of that edge. Finally, letting $\alpha_n \neq 0$ allows the vertex condition to generalize a Robin boundary condition. The Dirichlet vertex condition is simply

$$\Psi(\mathtt{v}_n) = 0. \tag{1.3}$$

In either case, we may also allow for nonhomogeneous vertex conditions by setting the right-hand side of (1.2) or (1.3) to some value $\phi_n \neq 0$.

With these in hand, we may define norms and function spaces, e.g., $L^p(\Gamma)$:

$$\|\Psi\|_{L^p(\Gamma)}^p = \sum_{m=1}^{|\mathcal{E}|} w_m \|\psi_m\|_{L^p}^p \tag{1.4}$$

and the L^2 inner product

$$\langle \Psi, \Phi \rangle = \sum_{m=1}^{|\mathcal{E}|} w_m \int_0^{\ell_m} \psi_m^*(x) \phi_m(x) \, \mathrm{d}x \, . \tag{1.5}$$

The inclusion of the weights w_m in these definitions follows naturally from their appearance in the conservation laws for evolution equations below and from demonstrating that the Laplace operator is self-adjoint. We then define $H^1(\Gamma)$ as the space of square-integrable functions with square-integrable first derivatives. More specifically, we define $L^2(\Gamma)$, $H^1(\Gamma)$, and $H^2(\Gamma)$ to be the space of functions that are, edge by edge, in each of these function spaces, but we define L^2_Γ to be the space $L^2(\Gamma)$ equipped with the inner product (1.5). Similarly, H^1_Γ is defined as the space of functions in $H_1(\Gamma)$ satisfying the continuity condition (1.1), and H^2_Γ consists of functions in $H^2(\Gamma)$ satisfying both Eq. (1.1) and either the Robin-Kirchhoff condition (1.2) or Dirichlet condition (1.3).

1.2. The Eigenvalue Problem. The first natural question to ask about the Laplacian operator defined on Γ is to characterize its spectrum and eigenfunctions. Such properties have been studied rather extensively; the recent works [7, 11, 15–17, 39, 44, 50, 72] provide a broad overview.

The spectrum of a compact graph consists solely of discrete points, so the problem consists of characterizing the set of eigenvalues λ and eigenfunctions Ψ such that

$$\triangle \Psi = \lambda \Psi. \tag{1.6}$$

All but a finite number of the eigenvalues are negative; this set is infinite and unbounded from below. The non-positive eigenvalues $\lambda = -k^2$ can be found by seeking the analytic solution

$$\psi_m(x) = a_m e^{ikx} + b_m e^{ik(\ell_m - x)} \qquad m = 1, 2, \ldots, |\mathcal{E}|. \tag{1.7}$$

The vertex conditions form a homogeneous system of $2|\mathcal{V}|$ linear equations. Its solution requires the vanishing of the determinant of the associated matrix, a function $\Sigma(k)$ known as the *secular determinant*. The authors of [17] demonstrate that $\Sigma(k)$ can be normalized to take real values when $k \in \mathbb{R}$. The recent dissertation [26] shows that the same is true under the more general vertex conditions (1.2). While QGLAB consists primarily of numerical tools for the approximate solution of differential equations on quantum graphs, it can symbolically compute the graph's real-valued secular determinant.

1.3. PDE on a Quantum Graph. Our primary motivating problem for writing QGLAB is the nonlinear Schrödinger equation

$$i \frac{\partial \Psi}{\partial t} = \triangle \Psi + (\sigma + 1) |\Psi|^{2\sigma} \Psi, \tag{1.8}$$

where $\sigma \geq 0$ and $\sigma = 1$ is the most commonly-studied cubic case. We are especially interested in the stationary NLS obtained by assuming $\Psi(x,t) = e^{i\Lambda t} \Psi(x)$,

$$-\Lambda \Psi = \triangle \Psi + (\sigma + 1) |\Psi|^{2\sigma} \Psi. \tag{1.9}$$

We note that the evolution of Eq. (1.8) conserves both the L^2 norm defined in Eq. (1.4) and an energy

$$E(\Psi) = \|\Psi'\|^2_{L^2(\Gamma)} - \|\Psi\|^{2(\sigma+1)}_{L^{2(\sigma+1)}(\Gamma)} + \sum_{\mathtt{v}_n \in \mathcal{V}} \alpha_n |\Psi(\mathtt{v}_n)|^2, \tag{1.10}$$

where Ψ' is defined edge-by-edge. The NLS equation on the real line also conserves a momentum functional. In general, NLS equations on quantum graphs obey no such conservation law unless certain other restrictions to the weights and initial conditions hold; see [52].

Linear and nonlinear PDEs on quantum graphs have a long history, and recent years have seen many advances. The book [63] gives an excellent introduction to various time-dependent equations on graphs. In particular, many groups have studied the solutions of Eq. (1.9)s. We mention the recent survey articles [22, 67] and note the many results about the existence and stability of ground state stationary solutions to the NLS equation (1.8) in references [1, 3–6, 21, 24, 25, 27, 31–33, 74, 75], many from a variational perspective. Similar works have studied the existence and stability of stationary states for Dirac-type equations [21] and KdV-type equations [70]. In addition, recent works including [18, 45, 47, 60, 68] use asymptotic and bifurcation-theoretical approaches to analyze the existence of multiple branches of solutions to (1.9). References including [2, 52, 64, 73] analyze the time-dependent phenomena exhibited by evolution equations of Schrödinger, Dirac, and KdV type on graphs. This is just a short overview of a massive subject. We do not claim to have captured all significant contributions here, but to give a flavor of the type of questions that can be posed on quantum graphs and the breadth of topics yet to be explored.

QGLAB has been explicitly written for PDE with Laplacian spatial derivative terms. In addition to the previously discussed NLS equation, these include the wave equation ([67]), heat equation ([13, 23]), and their nonlinear cousins such as the nonlinear Klein-Gordon equations, including sine-Gordon [46, 58, 73, 78] and the Kolmogorov–Petrovsky–Piskunov (KPP) equation [36], all of which can be defined on a quantum graph. QGLAB provides examples of solving all of these PDEs.

1.4. Organization of the paper.

Section 2 discusses the numerical methods QGLAB uses to discretize and solve equations posed on quantum graphs. The longest part, Subsection 2.1, discusses the overall framework of the discretization and its implementation using both finite-difference and Chebyshev approximations of derivatives and the implementation of the vertex conditions. We apply this framework to discretize eigenvalue problems in Subsection 2.2, where we also discuss the symbolic calculation of the secular determinant for the general class of vertex conditions discussed. Subsection 2.3 describes the nonlinear solvers and continuation algorithms, while Subsection 2.4 describes the implementation of time steppers for evolution equations. Section 3 is devoted to the MATLAB implementation of the tools discussed in QGLAB, including a discussion of MATLAB's directed graph class in Section 3.1 and the QGLAB's quantum graph class, which is built on top of this, in Section 3.2. Section 3.3 discusses basic operations on class objects. We summarize our contributions and give an outlook on potential future features and applications in Section 4. The paper contains extensive supplementary materials in two sections here as an appendix. The first, Sec. A, 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 and evolutionary PDE problems. The second, Sec. B, contains a complete listing of user-callable function definitions and explicit instructions for their use.

2. Numerical Methods.

This section discusses the numerical methods used to implement the quantum graph class and solve various problems. It briefly presents some examples described in detail in Sec. A of the appendix.

2.1. Discretization and vertex conditions. QGLAB grew out of numerical studies in the authors' research [47, 52, 60], but many groups have used finite-difference and finite-element discretizations to approximate the quantum graph Laplacian and solve PDEs, e.g. [8, 18, 20]. As mentioned in the introduction, Malenova built a small quantum graph package using *Chebfun* [59].

The QGLAB package features routines to perform several tasks, including solving nonlinear standing waves and numerically integrating evolution equations posed on a quantum graph. Still, the most central task is to discretize the Laplace operator and to solve the Laplace and Poisson equations posed on the quantum graph. We provide two methods of discretization: centered differences and Chebyshev collocation. Most of the examples provided work using either discretization, although the Chebyshev discretization is, by construction, more accurate.

The two discretizations are implemented using a common framework: the function $\Psi(x)$ is approximated on an *extended grid* $\mathbf{x}^{\text{ext}}$ containing enough points to approximate both the PDE solution and the vertex conditions, while the solution to the PDE is approximated on a smaller *interior* grid containing two fewer points per edge. Thus, the discrete Laplacian matrix is non-square, with $2|\mathcal{E}|$ more columns than rows, mapping from approximations on $\mathbf{x}^{\text{ext}}$ to approximations on $\mathbf{x}^{\text{int}}$. Second, the vertex conditions are implemented as constraints rather than incorporated directly into the discretized Laplacian matrix. This choice has several attractive features that we describe below

Driscoll and Hale introduced non-square differentiation matrices using rectangular discretization matrices for use in the Chebfun package [34, 35]. Aurentz and Trefethen have written an excellent review, developing the theory for the *block operators* that implement these ideas via a sequence of well-chosen examples [10].

2.1.1. Finite-difference discretization. The finite difference method is implemented using standard second-order centered differences with the boundary conditions enforced at so-called ghost points, as discussed, for example, in the textbook [38, Sec. 4.2.2]. We first review this technique for the discretization of the two-point boundary value problem

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = f(x), 0 < x < \ell,\ u'(0) + \alpha_0 u(0) = \phi_0,\ -u'(\ell) + \alpha_\ell u(\ell) = \phi_\ell. \tag{2.1}$$

and then discuss the straightforward extension to the case of quantum graphs. The sign on the u' term in the boundary conditions is chosen to agree with our quantum graph convention that all derivatives are taken in the direction pointing away from a vertex of the quantum graph in the definition of vertex conditions. Given a discretization length $h = \frac{\ell}{N}$, place points not at the usual lattice points but at $x_k = \left(k - \frac{1}{2}\right) h$ for $0 \leq k \leq N+1$ as shown in Fig. 2.1. Note that the first and last points lie outside the interval of interest and that the endpoints of the desired interval do not appear on the list of points. Letting u_k approximate $u(x_k)$, the discretized equation at the interior points is then

$$u''(x_k) \approx \frac{u_{k-1} - 2u_k + u_{k+1}}{h^2} = f_k = f(x_k), \quad \text{for} \quad k = 1, \dots, N,$$

up to an error of $\mathcal{O}\big(h^2\big)$. To approximate the boundary condition at $x = 0$ we use

$$u(0) = \frac{u(h/2) + u(-h/2)}{2} + \mathcal{O}\big(h^2\big) \quad \text{and} \quad u'(0) = \frac{u(h/2) - u(-h/2)}{h} + \mathcal{O}\big(h^2\big),$$

yielding the finite-difference approximation to boundary condition Eq. (2.1):

$$\left(\frac{\alpha_0}{2}-\frac{1}{h}\right)u_0+\left(\frac{\alpha_0}{2}+\frac{1}{h}\right)u_1=\phi_0 \tag{2.2}$$

and a similar approximation for the right boundary condition. In the case of a Dirichlet boundary condition at $x=0$, this boundary condition is replaced by $\frac{1}{2}\left(u_0+u_1\right)=\phi_0$. A standard way to handle the point u_0 is to solve Eq. (2.2) for u_0 in terms of u_1 and thus eliminate u_0 from the system. Instead, we choose to include this equation in our larger system of equations.

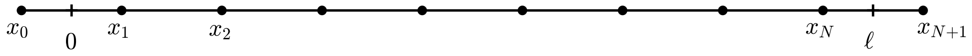


Fig. 2.1: Discretization of the interval $[0,\ell]$ using ghost points.

Assembling the equations for the second derivatives and the boundary conditions into matrix–vector form, we define the vectors

$$\mathbf{u}=\left(u_0,u_1,\ldots,u_{N+1}\right)^{\mathsf{T}},\mathbf{f}=\left(f_0,f_1,\ldots,f_{N+1}\right)^{\mathsf{T}},\quad\text{and}\quad\boldsymbol{\phi}=\left(\phi_0,\phi_L\right)^{\mathsf{T}}, \tag{2.3}$$

as well as the $N\times(N+2)$ *interior Laplacian matrix*

$$\mathbf{L}_{\text{int}}=\frac{1}{h^2}\begin{pmatrix}1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1\end{pmatrix}, \tag{2.4}$$

the $2\times(N+2)$ *boundary condition matrix*

$$\mathbf{M}_{\text{BC}}=\begin{pmatrix}\left(\frac{\alpha_0}{2}-\frac{1}{h}\right) & \left(\frac{\alpha_0}{2}+\frac{1}{h}\right) & \cdots & 0 & 0\\ 0 & 0 & \cdots & \left(\frac{\alpha_\ell}{2}+\frac{1}{h}\right) & \left(\frac{\alpha_\ell}{2}-\frac{1}{h}\right)\end{pmatrix}, \tag{2.5}$$

and the $N\times(N+2)$ *interior projection matrix*

$$\mathbf{P}_{\text{int}}=\begin{pmatrix}\mathbf{0}_{N\times1} & \mathbf{I}_N & \mathbf{0}_{N\times1}\end{pmatrix}, \tag{2.6}$$

where $\mathbf{I}_N$ is the N-dimensional identity matrix and $\mathbf{0}_{M\times N}$ is an $(M\times N)$-dimensional matrix of all zeros. The matrices $\mathbf{L}_{\text{int}}$ and $\mathbf{P}_{\text{int}}$ are linear maps from approximation of Ψ defined on the *extended grid* $\mathbf{x}^{\text{ext}}=\{x_0,\ldots,x_{N+1}\}$ to the approximation of the PDE defined on the *interior grid* $\mathbf{x}^{\text{int}}=\{x_1,\ldots,x_N\}$. This allows us to discretize the second derivative terms as

$$\mathbf{L}_{\text{int}}\mathbf{u}=\mathbf{P}_{\text{int}}\mathbf{f}, \tag{2.7}$$

and the boundary conditions as

$$\mathbf{M}_{\text{BC}}\mathbf{u}=\boldsymbol{\phi}. \tag{2.8}$$

We can then combine these into a single system of $N+2$ equations in $N+2$ unknowns

$$\left(\frac{\mathbf{L}_{\text{int}}}{\mathbf{M}_{\text{BC}}}\right)\mathbf{u}=\left(\frac{\mathbf{P}_{\text{int}}}{\mathbf{0}_{2\times(N+2)}}\right)\mathbf{f}+\left(\frac{\mathbf{0}_{N\times2}}{\mathbf{I}_2}\right)\boldsymbol{\phi}. \tag{2.9}$$

We make three brief remarks on this approach. First, we note that the more common technique is to solve the discretized boundary condition equations for u_0 and u_{N+1} in terms of u_0 and u_N and thereby reduce the number of unknowns to N. We choose not to do so for two reasons: one is that it makes implementing nonhomogeneous boundary conditions slightly more straightforward in time-dependent problems, and the other is that it makes the approach more similar to how we handle boundary conditions using Chebyshev discretization. Second, the null space of the matrix $\mathbf{L}_{\text{int}}$ mimics that of the second derivative: it consists of vectors $\mathbf{v}$ with $v_n = an+b$ and is two-dimensional. Third, ghost points have an important advantage over the standard on-point discretization when enforcing Neumann or Robin boundary conditions. The simplest way to discretize the boundary condition with second-order accuracy at the boundary is to approximate $\frac{\mathrm{d}u}{\mathrm{d}x}\big|_{x=0}$ using a one-sided difference involving the values u_0, u_1, and u_2, and similarly at the right boundary. Solving the two discretized boundary conditions eliminates the values of u_0 and u_{N+1} from the system, leaving a system of N unknowns. The resulting finite-difference matrix is not symmetric, while the differential operator which it approximates is, of course, self-adjoint. The ghost point discretization, by contrast, preserves self-adjointness (after the two ghost points are first eliminated from the system). We will return to this topic after discretizing the Laplacian on the quantum graph.

This scheme extends straightforwardly to the quantum graph. Consider the Poisson problem

$$\triangle\Psi(x) = f(x) \quad \text{meaning} \quad \psi_m''(x) = f_m(x) \quad \text{on} \quad \mathtt{e}_m \quad \text{for} \quad 1 \le m \le |\mathcal{E}| \tag{2.10a}$$

$$\psi_i(\mathtt{v}_n) = \psi_j(\mathtt{v}_n), \forall \mathtt{e}_i, \mathtt{e}_j \in \mathcal{V}_n, \quad \text{i.e., continuity,} \tag{2.10b}$$

$$\sum_{\mathtt{e}_m \in \mathcal{V}_n} w_m \psi_m'(\mathtt{v}_n) + \alpha_n \Psi(\mathtt{v}_n) = \phi_n \quad \text{or} \quad \Psi(\mathtt{v}_n) = \phi_n \quad \text{for} \quad 1 \le n \le |\mathcal{V}|\,. \tag{2.10c}$$

Each edge $\mathtt{e}_m$ is discretized using the ghost-point formulation, using N_m+2 discretization points to define $\mathbf{x}_m^{\text{ext}}$, and a mesh size $h_m = \ell_m/N_m$, generating a $(N_m+2)\times N_m$ matrix $\mathbf{L}_{\text{int}}^{(m)}$ of the same form as matrix (2.4), and a matrix $\mathbf{P}_{\text{int}}^{(m)}$ of the same dimension of the form as matrix (2.6). Thus letting $N_{\text{int}} = \sum_{m=1}^{|\mathcal{E}|} N_m$ be the total number of interior discretization points and $N_{\text{ext}} = N_{\text{int}} + 2|\mathcal{E}|$ be the total number of discretization points including boundary points, this discretization has resulted in N_{ext} unknowns arranged in a single vector as

$$\boldsymbol{\psi} = \begin{pmatrix} \boldsymbol{\psi}^{(1)} \\ \vdots \\ \boldsymbol{\psi}^{(|\mathcal{E}|)} \end{pmatrix}, \quad \text{where} \quad \boldsymbol{\psi}^{(m)} = \begin{pmatrix} \boldsymbol{\psi}_0^{(m)} \\ \vdots \\ \boldsymbol{\psi}_{N_m+1}^{(m)} \end{pmatrix}.$$

The vector $\mathbf{f}$ is assigned similarly, and the vector of nonhomogeneous boundary terms is $\boldsymbol{\phi} = \left(\phi_0, \ldots, \phi_{|\mathcal{V}|}\right)^{\mathsf{T}}$. Enforcing the continuity condition (2.10b) at the vertex $\mathtt{v}_n$ requires $(d_n - 1)$ rows and the Robin-Kirchhoff condition (2.10c) vertex $\mathtt{v}_n$ involves the $2d_n$ adjacent discretization points in one row. The derivative and function values of the vertex are approximated to second order using a straightforward generalization of the reasoning that leads to Eqs. (2.2) and (2.5). Altogether, these form a matrix

$\mathbf{M}_{\text{VC}}^{(n)}$ of dimension $(2d_n) \times N_{\text{ext}}$. We let

$$\mathbf{L}_{\text{VC}} = \left(\begin{array}{c} \mathbf{L}_{\text{int}} \\ \hline \mathbf{M}_{\text{VC}} \end{array} \right) = \left(\begin{array}{ccc} \mathbf{L}_{\text{int}}^{(1)} & & \\ & \ddots & \\ & & \mathbf{L}_{\text{int}}^{(|\mathcal{E}|)} \\ \hline & \mathbf{M}_{\text{VC}}^{(1)} & \\ & \vdots & \\ & \mathbf{M}_{\text{VC}}^{(|\mathcal{V}|)} & \end{array} \right), \tag{2.11}$$

and

$$\mathbf{P}_0 = \left(\begin{array}{c} \mathbf{P}_{\text{int}} \\ \hline \mathbf{0}_{2|\mathcal{E}| \times N_{\text{ext}}} \end{array} \right) = \left(\begin{array}{ccc} \mathbf{P}_{\text{int}}^{(1)} & & \\ & \ddots & \\ & & \mathbf{P}_{\text{int}}^{(|\mathcal{E}|)} \\ \hline & \mathbf{0}_{2|\mathcal{E}| \times (N_{\text{ext}})} & \end{array} \right). \tag{2.12}$$

We also define and define the *nonhomogeneity matrix* $\mathbf{M}_{\text{NH}}$ in two steps. First define a matrix $\mathbf{M}$ of size $2|\mathcal{E}| \times |\mathcal{V}|$ such that

$$\mathbf{M}(i,j) = \begin{cases} 1 & \text{If the } j\text{th Neumann-Kirchhoff condition is enforced by row } i \text{ of } \mathbf{M}, \\ 0 & \text{otherwise} \end{cases}$$

and then define

$$\mathbf{M}_{\text{NH}} = \left(\begin{array}{c} \mathbf{0}_{N_{\text{int}},|\mathcal{V}|} \\ \hline \mathbf{M} \end{array} \right). \tag{2.13}$$

This assigns the appropriate entry of the nonhomogeneous term $\boldsymbol{\phi}$ to the correct row of the matrix enforcing the vertex conditions. With these matrix definitions, the problem can be represented in the compact form

$$\mathbf{L}_{\text{VC}} \boldsymbol{\psi} = \mathbf{P}_0 \mathbf{f} + \mathbf{M}_{\text{NH}} \boldsymbol{\phi}. \tag{2.14}$$

We introduce some notation to simplify our discussion of the numerical problems addressed below. The matrices $\mathbf{P}_{\text{int}}$ and $\mathbf{L}_{\text{int}}$, of dimension $N_{\text{int}} \times N_{\text{ext}}$, represent linear maps from the function space $\mathbb{F}^{\text{ext}}$, of functions defined on the extended grid $\mathbf{x}^{\text{ext}}$, to the function space $\mathbb{F}^{\text{int}}$, of functions defined on the interior grid $\mathbf{x}^{\text{int}}$. Of course, $\mathbb{F}^{\text{ext}} = \mathbb{R}^{N_{\text{ext}}}$ and $\mathbb{F}^{\text{int}} = \mathbb{R}^{N_{\text{int}}}$, but simply thinking of these spaces merely as high-dimensional Euclidean spaces neglects the meaning to which we have assigned the elements of each space. Further, we denote by $\mathbb{F}_{\boldsymbol{\phi}}^{\text{ext}}$ the set of functions in $\mathbb{F}^{\text{ext}}$ which, in addition, satisfy the discretized boundary conditions represented by the final $2|\mathcal{E}|$ rows of system (2.14). Note that if $\boldsymbol{\phi} \neq \mathbf{0}$, i.e., for nonhomogeneous vertex conditions, then $\mathbb{F}_{\boldsymbol{\phi}}^{\text{ext}}$ is an affine space of dimension N_{int}, while for $\boldsymbol{\phi} = \mathbf{0}$, $\mathbb{F}_{\mathbf{0}}^{\text{ext}}$ is a linear vector space of dimension N_{int}. Thus the first N_{int} rows of Eq. (2.14) use the points from $\mathbf{x}^{\text{ext}}$ to approximately evaluate the underlying Laplace equations at the points in $\mathbf{x}^{\text{int}}$, while the remaining $2|\mathcal{E}|$ rows ensure that the solution lies on $\mathbb{F}_{\boldsymbol{\phi}}^{\text{ext}}$.

In what follows, we will apply similar reasoning to discretize other problems on the quantum graph. As above, we apply the differential equations at the interior points and supplement these equations with $2|\mathcal{E}|$ additional equations representing the vertex conditions, which, together, suffice to specify a unique solution. In addition to the matrices $\mathbf{L}_{\mathrm{VC}}$ and $\mathbf{P}_0$ defined in Eq. (2.11) and (2.12), we will also need

$$\mathbf{L}_0 = \left(\frac{\mathbf{L}_{\mathrm{int}}}{\mathbf{0}_{2|\mathcal{E}|\times N_{\mathrm{ext}}}}\right) \quad \text{and} \quad \mathbf{P}_{\mathrm{VC}} = \left(\frac{\mathbf{P}_{\mathrm{int}}}{\mathbf{M}_{\mathrm{VC}}}\right). \tag{2.15}$$

More generally, if $\mathbf{B}_{\mathrm{int}}$ is any matrix of the same dimension of $\mathbf{L}_{\mathrm{int}}$ and $\mathbf{P}_{\mathrm{int}}$, then we will define

$$\mathbf{B}_0 = \left(\frac{\mathbf{B}_{\mathrm{int}}}{\mathbf{0}_{2|\mathcal{E}|\times N_{\mathrm{ext}}}}\right) \quad \text{and} \quad \mathbf{B}_{\mathrm{VC}} = \left(\frac{\mathbf{B}_{\mathrm{int}}}{\mathbf{M}_{\mathrm{VC}}}\right). \tag{2.16}$$

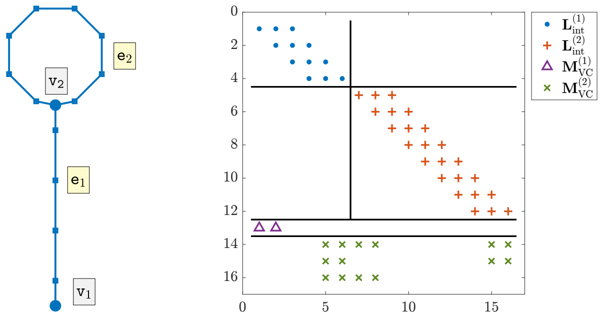
In practice, we will construct such a matrix $\mathbf{B}_{\mathrm{int}}$ as a linear combination of $\mathbf{L}_{\mathrm{int}}$ and $\mathbf{P}_{\mathrm{int}}$. Such examples arise in constructed time-stepping algorithms for evolutionary PDE in Secs. 2.4.2 and A.2.

We return here to the reasoning for choosing a discretization using ghost points, which comes from an attempt to maintain the self-adjointness of the Laplacian after discretizing. Suppose that, instead of including the vertex conditions as part of an extended linear system, we first use them to eliminate the function values at the ghost points, resulting in a smaller linear operator represented by a matrix $\mathbf{A}$. In the case of problem (2.1), using ghost points results in a symmetric matrix $\mathbf{A}$, while the matrix $\mathbf{A}$ that results from using second-order one-sided differences is asymmetric.

We now consider the discretization of the Laplacian on a quantum graph subject to the vertex conditions (1.1) and (1.2). If all edges are discretized with the same stepsize h, the discretization matrix constructed above is symmetric. However, choosing all discretization lengths to be equal may be impossible or inconvenient. In that case, we may measure the magnitude of the asymmetry by considering the largest element, in absolute value, of the asymmetric part $\frac{1}{2}\left(\mathbf{A}-\mathbf{A}^{\mathsf{T}}\right)$. Assume that edge $\mathtt{e}_m$ is discretized with a stepsize $h_m = h + \delta_m$ with $\delta_m = O(\delta) \ll h$. Then using one-sided centered differences introduces terms of $O(\frac{1}{h^2})$ into the asymmetric part of A. By contrast, using ghost points introduces terms of $O(\frac{1}{h^2}\cdot\delta)$. Therefore, if all the discretization lengths h_m are roughly equal, the non-symmetric part of A will be significantly smaller so that the matrix A is "more symmetric." Since A is not symmetric, we cannot guarantee that all of our eigenvalues are purely real as for the underlying differential operator.

An example. To demonstrate the structure of the discretized Laplacian matrix, we consider a lollipop graph consisting of two vertices and two edges shown in Fig. 2.2, the edge $\mathtt{e}_1$ points from $\mathtt{v}_1$ to $\mathtt{v}_2$ and the edge $\mathtt{e}_2$ points from $\mathtt{v}_2$ to itself. We have chosen a coarse discretization with $N_1 = 4$ and $N_2 = 8$ discretization points on the two edges. The interior points of the discretization and the vertices are shown in the figure, but not the ghost points. The figure also shows the structure of the nonzero entries of the matrix $\mathbf{L}_{\mathrm{VC}}$. The matrix $\mathbf{M}_{\mathrm{NH}}$ is 16×2 and is nonzero in the positions $(13,1)$ and $(14,2)$. Numerical convergence is demonstrated for an example Poisson problem in Sec. A.1.2 of the appendix.

2.1.2. Chebyshev discretization. To achieve spectral accuracy, QGLAB allows for discretization using a method based on Chebyshev polynomials due to Driscoll and Hale called *rectangular collocation* [34]. Further details necessary to accurately implement this method are described by Xu and Hale [80]. This method, based on a non-square differentiation matrix, allows for the straightforward implementation of non-trivial boundary conditions. To introduce the idea behind this method, we again


e2
v2
e1
v1
0
2
4
6
8
10
12
14
16
0
5
10
15
$\mathbf{L}_{\text{int}}^{(1)}$
$\mathbf{L}_{\text{int}}^{(2)}$
$\mathbf{M}_{\text{VC}}^{(1)}$
$\mathbf{M}_{\text{VC}}^{(2)}$

(a) (b)

Fig. 2.2: (a) The lollipop graph with interior discretization points. (b) The structure of the nonzero entries in the matrix $\mathbf{L}_{\text{VC}}$, the Laplacian matrix extended with vertex conditions.

consider the Robin problem on a line segment defined in Eq. (2.1). Consider the exterior grid $\mathbf{x}^{\text{ext}}$ given by the $N+2$ Chebyshev points of the second kind,[1]

$$x_k^{\text{ext}} = \frac{\ell}{2}\left(1 - \cos\left(\frac{k\pi}{N+1}\right)\right), \qquad k = 0, 1, \ldots, N, N+1. \tag{2.17}$$

We adapt the notation of Eq. (2.3) to define the vectors $\mathbf{u}$, $\mathbf{f}$, and $\boldsymbol{\phi}$ on the discretization points defined in Eq. (2.17). The main observation motivating rectangular collocation is that applying a second derivative matrix defined over a finite space of polynomials should reduce the order of that space by two, naturally leading to matrices of size $N \times (N+2)$. This is realized by first operating on the vector $\mathbf{u}$ with the standard $(N+2) \times (N+2)$ Chebyshev derivative matrix $\mathbf{D}^2$ and then resampling these polynomials onto the interior grid $\mathbf{x}^{\text{int}}$ consisting of Chebyshev points of the first kind

$$x_k^{\text{int}} = \frac{\ell}{2}\left(1 - \cos\left(\frac{(2k-1)\pi}{2N}\right)\right) \qquad k = 1, 2, \ldots, N. \tag{2.18}$$

As in the finite-difference discretization described above, we have defined exterior and interior grids $\mathbf{x}^{\text{ext}}$ and $\mathbf{x}^{\text{int}}$, where the approximate solutions are ultimately defined on $\mathbf{x}^{\text{ext}}$, but derivatives, and thus approximations to the differential equations, are evaluated at the points of $\mathbf{x}^{\text{int}}$. In contrast to the finite difference case in which $\mathbf{x}^{\text{int}} \subsetneq \mathbf{x}^{\text{ext}}$, the two grids, in this case, are disjoint sets. Further, note that the endpoints of the interval, rather than ghost points, here are given as the first and last points of the exterior grid $\mathbf{x}^{\text{ext}}$.

Resampling is a linear operation and is represented by an $N \times (N+2)$-dimensional *barycentric resampling matrix* $\mathbf{P}_{\text{int}}$ whose construction uses the barycentric interpolation formula proposed in [19] and is the basis of the Chebyshev implementation of generic boundary value problems in *Chebfun*. Given the set of points $\mathbf{x}^{\text{ext}} =$

---

[1] The Chebyshev points are defined in a slightly nonstandard way so that the coordinates in the vector $\mathbf{x}$ increase with increasing values of k.

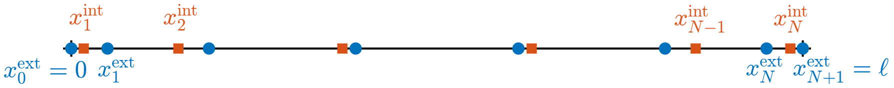


Fig. 2.3: Discretization of the interval $[0, \ell]$ using a grid $\mathbf{x}^{\text{ext}}$ of Chebyshev points of the second kind (in blue) and a grid $\mathbf{x}^{\text{int}}$ first kind Chebyshev points (in red).

$\{x_k^{\text{ext}}\}_{k=0}^{N+1}$, the barycentric weights are

$$w_k = \prod_{\substack{l=0 \\ l \neq k}}^{N+1} (x_k^{\text{ext}} + x_l^{\text{ext}})^{-1}, \qquad k = 0, \ldots, N+1. \tag{2.19}$$

These are used to construct a unique interpolating polynomial,

$$p_{N+1}(x) = \frac{\sum_{k=0}^{N+1} (w_k/(x - x_k^{\text{ext}})) f_k}{\sum_{l=0}^{N+1} (w_l/(x - x_l^{\text{ext}}))}, \tag{2.20}$$

which interpolates the set of data points $\{(x_k^{\text{ext}}, f_k)\}_{k=0}^{N+1}$. The polynomial is evaluated at both $\{x_k^{\text{ext}}\}_{k=0}^{N+1}$ and $\{\chi_k\}_{k=1}^{N}$ so that the barycentric resampling matrix is given by

$$(\mathbf{P}_{\text{int}})_{j,k} = \begin{cases} \frac{w_k}{x_j^{\text{int}} - x_k^{\text{ext}}} \left(\sum_{l=0}^{N+1} \frac{w_l}{x_j^{\text{int}} - x_l^{\text{ext}}} \right)^{-1} & x_j^{\text{int}} \neq x_k^{\text{ext}}, \\ 1 & x_j^{\text{int}} = x_k^{\text{ext}}, \end{cases} \tag{2.21}$$

and satisfies

$$p_{N+1}(\mathbf{x}^{\text{int}}) = \mathbf{P}_{\text{int}}\, p_{N+1}(\mathbf{x}^{\text{ext}}). \tag{2.22}$$

The barycentric resampling matrix efficiently and stably evaluates an interpolating polynomial defined on a given set of points (in this case, on Chebyshev points of the second kind) at a different set of points (in this case Chebyshev points of the first kind). A more generalized construction is given by Driscoll and Hale in [34, Sec. 3.1].

Putting this all together, the product

$$\mathbf{L}_{\text{int}} = \mathbf{P}_{\text{int}} \mathbf{D}^2 \tag{2.23}$$

defines a $N \times (N+2)$ differentiation matrix. The right-hand side of the differential equation (2.1) must be resampled to the same set of points, so the differential equation is discretized by the N equations $\mathbf{L}_{\text{int}} \mathbf{u} = \mathbf{P}_{\text{int}} \boldsymbol{\phi}$, leaving two more equations to define the boundary conditions of Eq. (2.1). These may be compactly rewritten as

$$\mathbf{M}_{\text{BC}} \mathbf{u} = \begin{pmatrix} \phi_0 \\ \phi_L \end{pmatrix},$$

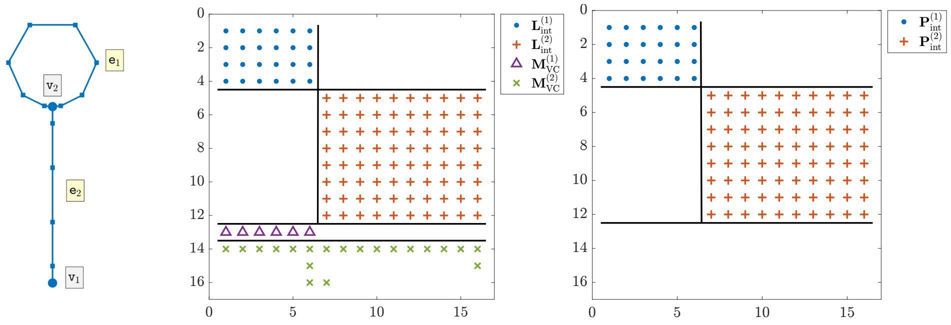
where $\mathbf{M}_{\text{BC}}$ is a matrix of size $2 \times (N+2)$ conveniently expressed using unit vectors

$$\mathbf{M}_{\text{BC}} = \begin{pmatrix} \mathbf{e}_1^{\mathsf{T}} \mathbf{D} + \alpha_0 \mathbf{e}_1^{\mathsf{T}} \\ -\mathbf{e}_{N+2}^{\mathsf{T}} \mathbf{D} + \alpha_L \mathbf{e}_{N+2}^{\mathsf{T}} \end{pmatrix}. \tag{2.24}$$

This matrices $\mathbf{L}_{\text{int}}$ and $\mathbf{P}_{\text{int}}$ are dense, in contrast to the equivalent matrices in the uniform discretization, defined in Eq. (2.5), which are banded. With that, we have constructed all the necessary elements to reinterpret Eq. (2.9) as a spectral collocation of Eq. (2.1).

It remains to extend this construction to the Poisson problem on the quantum graph, given in Eq. (2.10). Letting $\mathbf{x}^{\text{ext}}_{(m)}$ and $\mathbf{x}^{\text{int}}_{(m)}$ be the be, respectively, the extended discretization vector and the interior discretization vectors on edge e_m, then these are concatenated, respectively, into vectors $\mathbf{x}^{\text{ext}}$ and $\mathbf{x}^{\text{int}}$ on which the space $\mathbb{F}^{\text{ext}}$ and $\mathbb{F}^{\text{int}}$ are defined.

Defining the matrices $\mathbf{L}_{\text{VC}}$, $\mathbf{P}_0$, and $\mathbf{M}_{\text{NH}}$ in Eq. (2.14) is straightforward once we construct the submatrix $\mathbf{M}^{(n)}_{\text{VC}}$ defining the discretized vertex condition (2.11) and (2.12), extending the construction in (2.24). Fig. 2.4 shows a coarse discretization of the example shown in Fig. 2.2, in which the blocks defining the second derivative are dense, as are the rows defining the Robin-Kirchhoff vertex condition, whereas the rows enforcing continuity on the boundaries contain only two nonzero entries.



(a) (b) (c)

Fig. 2.4: (a) The lollipop graph, shown with interior discretization points in the Chebyshev discretization. (b) The structure of the nonzero entries in $\mathbf{L}_{\text{VC}}$, the Laplacian matrix extended with vertex conditions. (c) The structure of $\mathbf{P}_0$, the barycentric resampling matrix extended with zeros.

An early version of this Chebyshev discretization method was used in [12] to calculate the eigenvalues and eigenvectors of the related problem of a Laplacian on an interval perturbed by a large number of delta function potentials.

2.2. Numerical and Symbolic Eigenproblems. Following the steps used above to discretize the Poisson problem leads to a discretized form of the eigenvalue problem (1.6)

$$\mathbf{L}_{\text{VC}}\mathbf{u} = \lambda \mathbf{P}_0 \mathbf{u} \tag{2.25}$$

in both the finite difference and Chebyshev discretizations. Since $\mathbf{P}_0$ is not the identity matrix and is singular, this is a *generalized eigenvalue problem*. MATLAB has two built-in solvers for eigenvalue problems, `eig` and `eigs`, but only the latter is defined for generalized eigenvalue problems. QGLAB has overloaded the `eigs` command so that `[d,v]=G.eigs(m)` returns the m eigenvalues of the smallest absolute value. Its use on a Y-shaped graph with Dirichlet conditions at the ends of the two shorter edges is shown in Fig. 2.5. QGLAB's plotting features are described in Sec. 3.3.1.

The secular determinant, described above, is a function $\Sigma(k)$ whose zeros k_n correspond to eigenvalues $-k_n^2$ of the Laplacian operator. While not as powerful

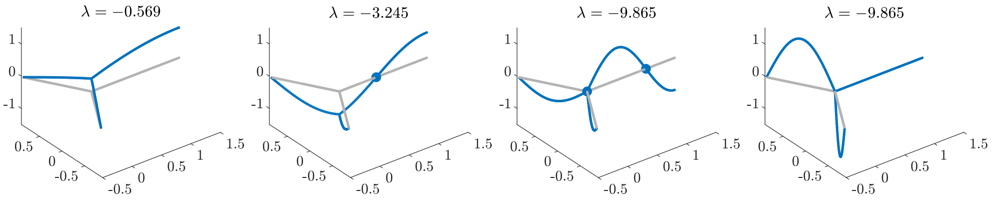
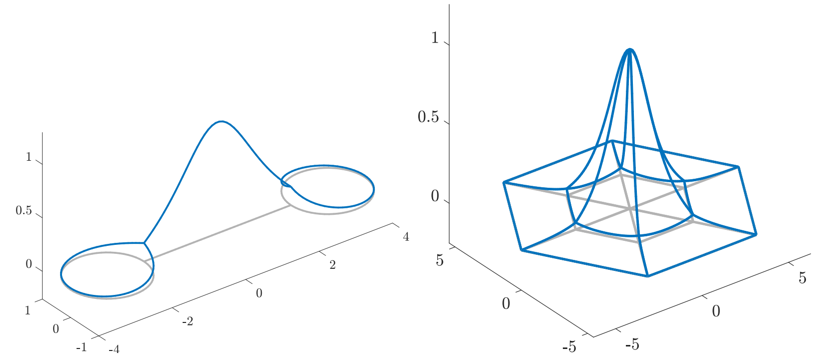


Fig. 2.5: Four eigenfunctions of a Y-shaped quantum graph.



(a) (b)

Fig. 2.6: (a) A standing wave of cubic NLS on a dumbbell graph with $\Lambda = -1$. (b) A standing wave on a spiderweb graph with $\Lambda = -1$.

or full-featured as Mathematica and Maple, MATLAB'S Symbolic Math Toolbox can manipulate and simplify complex exponentials and trigonometric functions sufficiently to construct $\Sigma(k)$ and simplify it to a form that we can typeset and plot. QGLAB has implemented the calculation of the secular determinant for the boundary conditions (1.2) with $w_j \equiv 1$ and for Dirichlet boundary conditions using a call of the form `f = G.secularDet`, described in detail in the dissertation [26]. This provides an independent check on the numerical calculation of eigenvalues for the discretized problem. Applying this routine to the graph shown in Fig. 2.5 yields the symbolic secular determinant

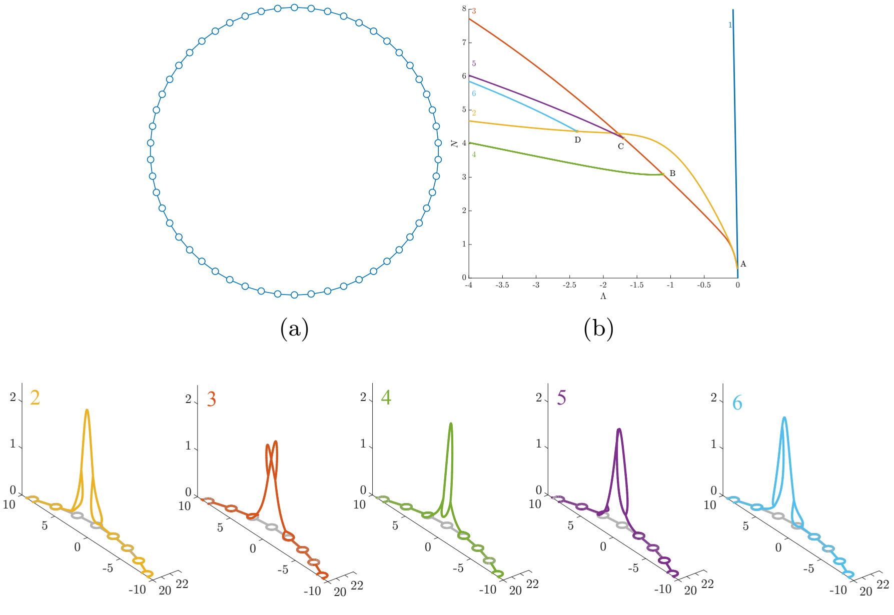
$$\Sigma(k) = \frac{16}{3} \sin\frac{k}{2} \left(1 - \sin^2\frac{k}{2}\right)\left(1 - 9\sin^2\frac{k}{2} + 12\sin^4\frac{k}{2}\right)$$

whose zeros are consistent with the numerically determined eigenvalues.

2.3. Nonlinear Solvers, Continuation, and Bifurcation Algorithms. After discretizing the spatial derivatives, QGLAB implements the Newton-Raphson method to solve for standing wave solutions of the stationary NLS (1.9). Two examples of such solutions are shown in Fig. 2.6.

Solutions to Eq. (1.9) do not occur at isolated points but along one parameter families that, away from singularities, can be parameterized by the frequency Λ. Pseudo-arclength continuation provides a way to follow this family as it traces a smooth path. It is due originally to Keller [53] and is well summarized in the textbook of Nayfeh and Balachandran [65], who cite many additional contributors to the method. Importantly, this method allows one to continue the curve around a fold singularity, and other techniques allow one to detect branch points, which include both pitchfork and transcritical bifurcations, and to determine the direction at which new families of solutions branch off from the branch being followed; see also Govaerts [48]. More rudimentary bifurcation calculations were performed in our previous works [18, 47, 60].

Fig 2.7 shows the so-called necklace graph, similar to an example in Ref. [20], except that all solutions are calculated on the same compact domain. The cited paper allows the numerical domain to widen as the amplitude decreases, demonstrating the ground state scaling at small amplitude. However, because it does not employ continuation and branch-switching, it does not demonstrate the relationships between the branches.



(a) (b)

(c)

Fig. 2.7: (a) Layout of the necklace graph, with 54 “strings” and 54 “pearls.” (b) Partial bifurcation diagram. (c) Solutions along the color-coded branches with frequency $\Lambda \approx -4$.

2.4. Time-stepping for evolution problems posed on a quantum graph. In this section, we consider two methods for approximating a function $\Psi(t) \in H^1(\Gamma)$ that evolves according to the PDE

$$\frac{\partial \Psi}{\partial t} = \mu \triangle \Psi + F(t, \Psi), \tag{2.26}$$

subject to a time-dependent version of vertex condition (2.10c), where ϕ_n is allowed to vary in time, and such that $F(t, \Psi)$ contains any terms involving a potential or nonlinearity. Depending on the constant μ, which could be real, imaginary, or complex, this formulation includes heat, Schrödinger, Ginzburg-Landau, and scalar reaction-diffusion equations.

We may consider the discretized evolution equation and vertex conditions as a system

$$\mathbf{P}_{\text{int}} \frac{\mathrm{d}\boldsymbol{\psi}}{\mathrm{d}t} = \mu \mathbf{L}_{\text{int}} \boldsymbol{\psi} + \mathbf{P}_{\text{int}} \mathbf{F}(t, \boldsymbol{\psi}), \tag{2.27a}$$

$$\mathbf{M}_{\text{VC}} \boldsymbol{\psi}(t) = \mathbf{M}_{\text{NH}} \boldsymbol{\phi}(t), \tag{2.27b}$$

where $\mathbf{F}(t, \boldsymbol{\psi})$ is considered as a vector-valued function $\mathbf{F} : \mathbb{R} \times \mathbb{F}^{\text{ext}} \to \mathbb{F}^{\text{ext}}$, which must then be projected onto $\mathbb{F}^{\text{int}}$ by $\mathbf{P}_{\text{int}}$. Here, Eq. (2.27a) discretizes the differential equations on $\mathbb{F}^{\text{int}}$, while Eq. (2.27b) enforces the vertex conditions, ensuring that the solution remains in $\mathbb{F}^{\text{ext}}_{\boldsymbol{\phi}(t)}$ at all times. This is a system of differential-algebraic equations since it contains some components that are differential equations and others that are algebraic.

In QGLAB, we have implemented two approaches to solve (2.28), described below. First, we combine the two parts of system (2.27) into a standard DAE form and pass it to appropriate built-in MATLAB ODE solvers. In the second approach, we adapt a Runge-Kutta (RK) algorithm to work directly with system (2.27). Because the ODE system derived in this manner is stiff and nonlinear, we use an implicit-explicit (IMEX) RK scheme.

The methods described in this section have been implemented only for scalar equations with a single time derivative. In the numerical examples of Sec. A.2, we also include a simple implementation of a leapfrog method to solve a nonlinear wave equation with second-order time derivatives.

2.4.1. DAE formulation for use with MATLAB's ODE suite. The two components of system (2.27) may be combined into a single equation by first multiplying Eq. (2.27b) by μ and using the vertex conditions to extend the $\mathbf{L}_{\text{int}}$ operator, while the $\mathbf{P}_{\text{int}}$ operator is extended with zeros, yielding a system of the form

$$\mathbf{P}_0 \frac{\mathrm{d}\boldsymbol{\psi}}{\mathrm{d}t} = \mu \mathbf{L}_{\text{VC}} \boldsymbol{\psi} + \mathbf{P}_0 \mathbf{F}(t, \boldsymbol{\psi}) - \mu \mathbf{M}_{\text{NH}} \boldsymbol{\phi}(t). \tag{2.28}$$

Since the final $2|\mathcal{E}|$ rows of $\mathbf{P}_0$ are identically zero, it is singular, and Eq. (2.28) is an index-one DAE. The MATLAB ODE solvers `ode23t` and `ode15s` can solve such DAEs, given in the general form

$$\mathbf{M}(t, \boldsymbol{\psi}) \dot{\boldsymbol{\psi}} = \mathbf{f}(t, \boldsymbol{\psi}),$$

in which $\mathbf{M}$, the so-called *mass matrix* given by $\mathbf{P}_0$ in Eq. (2.28), may be singular.

We have defined two solvers `qgde23t` and `qgde15s` that, in turn, call the two similarly named MATLAB solvers. The first uses an implicit trapezoidal rule and is recommended for problems of moderate stiffness where the user wishes to avoid numerical damping. The second uses the Klopfenstein-Shampine family of numerical differentiation formulas of orders one through five and is recommended for highly stiff problems [61, 76]. Both methods adaptively choose the step size based on user-specified tolerances, although the user may select a sequence of times to output the solution. As such, it is difficult to measure their order of convergence through numerical tests. However, we may use proxies such as the numerical maintenance of conserved quantities that are necessary but insufficient for this purpose. For moderately stiff problems, such as cubic NLS, `qgde23t` performs much faster. Figure 2.8

shows the collision of a soliton with a vertex on a star graph with both so-called balanced and unbalanced Kirchhoff conditions, as considered in Ref. [52]. Here, balance is defined by a restriction on the weights w_j in Eq. (1.2) with $\alpha_n \equiv 0$. The soliton passes through the balanced vertex but is largely reflected by the unbalanced vertex.

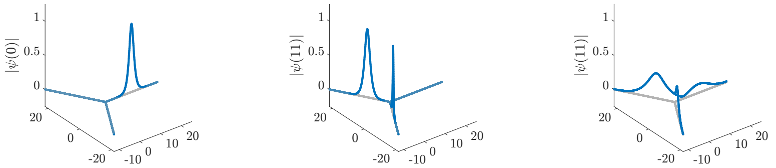


Fig. 2.8: Collision of an NLS soliton with the vertex of a star graph. (Left) Initial time. (Center) Final time, balanced graph. (Right) Final time, unbalanced graph.

For a stiff problem, such as the Fisher-KPP equation,

$$u_t = \mu \triangle u + u(1-u),$$

the stiff solver `qgde15s` proves more efficient. A computation on a honeycomb graph is shown in Fig. 2.9.

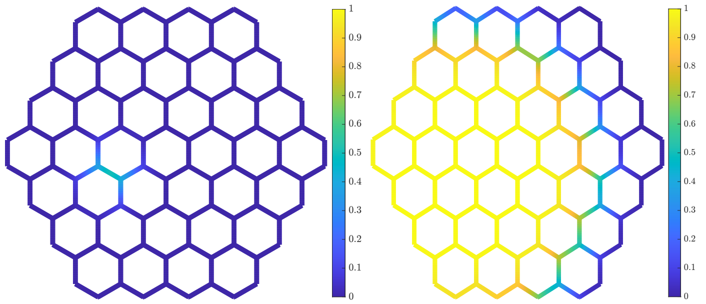


Fig. 2.9: Solution of the KPP equation, with the initial condition on the left and the solution at $t = 12$ on the right.

2.4.2. Building time-stepping algorithms from scratch. The methods described above run very slowly because problem (2.26) is *stiff* and *nonlinear*. The vertex-condition constraints make the straightforward application of standard methods somewhat difficult. We here construct a method that overcomes these problems.

The main issues in constructing a time-stepping algorithm for an evolutionary PDE defined on a quantum graph using the spatial discretization described in Sec. 2.1 can be illustrated using Euler methods. These ideas then extend straightforwardly to Runge-Kutta algorithms. To derive these methods, we first discretize in time, fixing a time step h, defining a sequence of discretization times $t_n = nh$, and denoting by $\psi_n(x)$ the approximate solution to Eq. (2.26) at $t = t_n$.

The forward Euler method is then

$$\psi_{n+1} = \psi_n + h \left(\alpha \triangle \psi_n + f(\psi_n) \right)$$

subject to vertex conditions on ψ_{n+1}. After we discretize in space and enforce the vertex conditions, this yields a time-stepper

$$\mathbf{P}_{\mathrm{VC}}\boldsymbol{\psi}_{n+1} = \mathbf{P}_0 \cdot (\boldsymbol{\psi}_n + h\mathbf{f}(\boldsymbol{\psi}_n)) + h\alpha\mathbf{L}_0\boldsymbol{\psi}_n. \tag{2.29}$$

The matrix $\mathbf{P}_{\mathrm{VC}}$ on the left means this method is implicit, but the implicitness is linear and does not require Newton iterations. However, the Laplacian on the right is evaluated explicitly, imposing a step-size restriction, so we do not consider this a practical method.

Similarly, the backward Euler method is

$$\psi_{n+1} = \psi_n + h\left(\alpha\triangle\psi_{n+1} + f(\psi_{n+1})\right)$$

subject to vertex conditions on ψ_{n+1}. After we discretize in space and enforce the vertex conditions, this yields a time-stepper

$$\begin{bmatrix} \mathbf{P}_{\mathrm{int}} - h\alpha\mathbf{L}_{\mathrm{int}} \\ \mathbf{M}_{\mathrm{VC}}, \end{bmatrix} \boldsymbol{\psi}_{n+1} - h\mathbf{P}_0\mathbf{f}(\boldsymbol{\psi}_{n+1}) = \mathbf{P}_0\boldsymbol{\psi}_n. \tag{2.30}$$

This method is fully implicit. It has no time-step restrictions but is implicit in a nonlinear term, requiring Newton iterations at each step. The simplest generalizations of the backward Euler method to Runge-Kutta schemes are so-called *diagonally implicit* (DIRK) schemes. These require several substeps that require the solution of equations similar to Eq. (2.30), which are then combined to produce a higher-order approximation to $\boldsymbol{\psi}(t_n + h)$. We have implemented Nørsett's three-stage fourth-order DIRK method as the quantum graph method `qgdeN34DIRK`. We have found it performs very slowly because it must solve a nonlinear equation at each substep.

To resolve this difficulty, we may treat the stiff term involving the Laplacian implicitly and the nonstiff term involving the nonlinearity explicitly. This idea was introduced for Runge-Kutta methods, which include the Euler method, by Ascher et al. [9]. There exist several such implicit-explicit (IMEX) Euler methods, including one they call forward-backward Euler $(1,1,1)$:

$$\psi_{n+1} = \psi_n + h\left(\alpha\triangle\psi_{n+1} + f(\psi_n)\right)$$

subject to vertex conditions on ψ_{n+1}. After we discretize in space and enforce the vertex conditions, this yields a time-stepper

$$\begin{bmatrix} \mathbf{P}_{\mathrm{int}} - h\alpha\mathbf{L}_{\mathrm{int}} \\ \mathbf{M}_{\mathrm{VC}} \end{bmatrix} \boldsymbol{\psi}_{n+1} = \mathbf{P}_0 \cdot (\boldsymbol{\psi}_n + h\mathbf{f}(\boldsymbol{\psi}_n)). \tag{2.31}$$

This method combines the best aspects of the forward and backward Euler methods. Moving the operator $h\alpha\mathbf{L}_{\mathrm{int}}$ to the left-hand side resolves the stiffness issue without requiring a small step size h. Keeping the nonlinear term on the right-hand side eliminates the need to solve a nonlinear equation on each step. However, the method is only first order in time, requiring small time steps for accuracy. We therefore turn to higher-order Runge-Kutta methods.

Ref. [9] applies similar ideas to derive IMEX Runge-Kutta methods, which at each stage handle the stiff part of the evolution equation implicitly and the nonstiff part explicitly. QGLAB comes with a four-stage third-order Runge Kutta method `qgdeSDIRK443`, based on the method denoted $(4,4,3)$ in [9]. As described in Sec. A.2.3 of the appendix, this final method solves the example shown in Fig. 2.8 faster than any other option.

3. Understanding the MATLAB implementation.

3.1. MATLAB's digraph class. The MATLAB numerical computing environment provides software tools for working with undirected and directed graphs. The main component of QGLAB is a class `qg` which builds upon MATLAB's `digraph` class used for defining directed graph objects. We illustrate the construction of a directed graph with a simple example, in which we initialize a `digraph` object `G` composed of four vertices and seven edges:

```
source=[1 1 1 1 2 2 3]; target=[2 2 3 4 1 3 3];
G = digraph(source, target);
plot(G)  % Plot called with additional figure-formatting options
```

The vector `source` specifies the initial vertices of the edges, and the vector `target` specifies the final vertices. The resulting graph is shown in Fig. 1.1. The digraph object `G` contains two fields: `G.Edges` and `G.Nodes`, each is in the form of a table, a MATLAB array type that holds column-oriented data, each column stored as a variable. The methods `G.numnodes` and `G.numedges` return the number of vertices (nodes) and edges, respectively. The array of edges contains one variable `G.Edges.EndNodes`, an array of size $|\mathcal{E}| \times 2$ whose two columns contain, respectively, the indices of the source vertices and the target vertices. The nodes table is initially empty. We add fields to the two tables to create the quantum graph class.

3.2. Understanding the quantum graph class and initializing a quantum graph object. To set a concrete example, the graph shown in Fig. 2.2 and the associated figure were generated with the code:

```
source=[1 2]; target=[2 2]; L=[4 2*pi]; nx=[4 8];
G=quantumGraph(source,target,L,'nxVec',nx)
```

The last line initializes a `qg` object. The three required arguments `source`, `target`, and `L` must be entered in that order. The last, `nx`, is an optional argument. If `nx` is a vector of length `G.numedges`, it defines the number of interior points on each edge. If it is a scalar, the constructor will assign `nx` points per unit length to each edge, rounding if necessary.

There exist several other optional arguments a user may set, which will be discussed below, some of which will be set to default values if not specified in the function call. Optional arguments are listed in the function call after required arguments using a key/value syntax. In older releases of MATLAB, this is entered as G= quantumGraph(source,target,L,'key1',value1,'key2',value2) while more recent releases allow the more compact syntax G=quantumGraph(source,target,L,key1=value1,key2 =value2) In the interest of compatibility, we use the former syntax in this document. Complete instructions, including all the optional initialization arguments, are presented in Sec. B.

The above commands return the following in the MATLAB command window:

```
G =

  QuantumGraph with properties:

              discretization: 'Uniform'
         wideLaplacianMatrix: [12x16 double]
         interpolationMatrix: [12x16 double]
```

```
          discreteVCMatrix: [4x16 double]
    nonhomogeneousVCMatrix: [16x2 double]
          derivativeMatrix: [16x16 double]
```

The most important property of this `quantumGraph` object is the property that specifies the quantum graph itself and its discretization. It is not visible in the above listing because it has been declared a *private property* of the object and can not be directly accessed by the user, only acted upon by class methods. We will discuss it last. The remaining properties are publicly viewable but can only be set by class methods. They are:

- `discretization` may take three values, `'Uniform'`, `'Chebyshev'`, or `'None'`. If the property `nx` is left undefined, then this defaults to `'None'`. Otherwise, it defaults to `'Uniform'`.
- `wideLaplacianMatrix` The rectangular Laplacian matrix $\mathbf{L}_{\text{int}}$ defined in Eq. (2.11) for either the uniform or Chebyshev discretization.
- `interpolationMatrix` The interpolation or resampling matrix $\mathbf{P}_{\text{int}}$ as defined in Eq. (2.12) for either the uniform or Chebyshev discretization.
- `discreteVCMatrix` The matrix $\mathbf{M}_{\text{VC}}$ defining the discretization of the vertex conditions defined in Eq. (2.11) and (2.12) for either the uniform or Chebyshev discretization.
- `nonhomogeneousVCMatrix` The matrix $\mathbf{M}_{\text{NH}}$ defined in Eq. (2.13) which maps nonhomogeneous terms in the vertex condition to the appropriate row.
- `derivativeMatrix` This square matrix approximates the first derivative on each edge. Unlike the matrices defined above, it is not a fundamental feature of QGLAB. It is used by functions that compute solutions' energy and momentum functionals of solutions, plot branches of solutions in continuation problems, and monitor conservation laws of time-dependent PDE.

The property `qg` is a MATLAB directed-graph object with additional fields necessary to define a quantum graph, consisting of a `Nodes` table and an `Edges` table. Because `qg` is a private property, viewing these tables using the syntax `G.qg.Nodes` and `G.qg.Edges` is disabled. Instead, `Nodes` and `Edges` `quantumGraph` methods have been written that return each of these tables, so we may view the tables using the syntax `G.Nodes` and `G.Edges`, as in this code listing. In addition, a method exists that returns each default table column; for example, the Robin coefficients can be returned by `G.robinCoeffs`. We examine the node data, which has three fields

```
>> disp(G.Nodes)
    robinCoeff    nodeData     y
    __________    ________    ___
        0            0        NaN
        0            0        NaN
```

The fields are:

- `robinCoeff` The Robin coefficients α_n used to define the vertex condition (1.2). If unset, it takes the default value of zero, reducing the boundary condition to the standard Neumann-Kirchoff condition. To implement a Dirichlet boundary condition (1.3), this coefficient is set to not-a-number (`NaN`).
- `nodeData` When the boundary conditions (1.2) and (1.3) are replaced with their nonhomogeneous generalizations, this field is used for the boundary data.
- `y` contains the value of ψ at the vertex, set to `NaN` on initialization.

We then examine the edges table, which has seven required fields (six for the Chebyshev discretization):

```
>> disp(G.Edges)
      EndNodes    Weight      L        nx      x        y       dx
      --------    ------    ------     --    ----    ---    ------
       1    2    1      4    4    { 6x1  double}    { 6x1  double]    1
       2    2    1 6.2832    8    {10x1  double}    {10x1  double} 0.7854
```

The fields are:

- `EndNodes` This $|\mathcal{E}| \times 2$ array the initial and final vertices of the edges, i.e., the content of the input variables `source` and `target`.
- `Weight` The weight w_j in vertex condition (1.2). Defaults to one if unset.
- `L` The array of edge lengths.
- `nx` The number of discretization points on the interior of each edge. If a scalar `nx` is passed to the `qg` command, the vector `nx` is set when the graph is built.
- `x` The ($\texttt{nx}+2$) discretization points on each edge, including ghost points for the uniform discretization and vertices in the Chebyshev discretization.
- `y` The value of ψ at the discretization points, initially set to `NaN`.
- `dx` The discretization step size of each edge. (Uniform discretization only.)

Another optional input is a separate MATLAB program that defines coordinates for plotting output. After defining the plot coordinates in the file labeled `lollipopPlotCoords.m`, the coordinates can be added to the existing graph `G` by passing its function handle to the method G.addPlotCoords(@lollipopPlotCoords) after `G` has been created, or else, can be included in the constructor with the command:

G=quantumGraph(sources,targets,L,'nxVec',nx,'PlotCoordinateFcn',@lollipopPlotCoords)

The plot coordinates are stored in fields `x1` and `x2` that are appended to both the `Edges` and `Nodes` tables. The MATLAB `plot` command has been overloaded so that `G.plot` plots the `y` coordinate over a skeleton of the graph in the `x1` and `x2` coordinates. Some graphs, such as those formed from the edges and vertices of a platonic solid, are best depicted in three space variables. Adding a third plot coordinate `x3` is possible for such cases. If this coordinate exists, then the graph is plotted in three dimensions with the `y` coordinated represented by a color scale.

QGLAB includes templates for various commonly studied graphs and a template syntax that allows the quick creation of such graphs, such as the lollipop and tetrahedron templates used in Sec. 3.3.1. The parameters used to define each template have default values that can be overridden. A gallery of graph templates is included in the documentation.

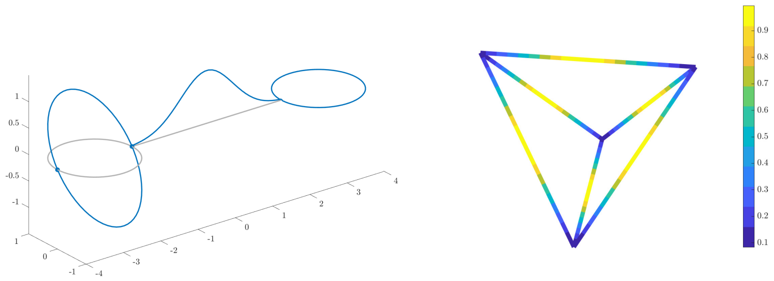
3.3. Basic operations.

MATLAB introduced *Live Scripts* in release 2016a. A live script is a rich document that includes runnable code and formatted text, entered with a simple word processor-like interface, and which integrates outputs including text and graphics, which can be exported to popular formats such as PDF, HTML, and Microsoft Word. The QGLAB package includes many examples created as live scripts and exported to HTML. Basic operations are described in the file `documentation/quantumGraphRoutines.mlx`.

3.3.1. Function evaluation and plotting.

The command to evaluate a function specified by a function handle, anonymous function, or constant value and assigns its value to the edge $\texttt{e}_j$ is `applyFunctionToEdge`. The command titled as `applyFunctionsToAllEdges` applies a cell array of functions to all the edges; for example, the following sequence of commands defines and plots a dumbbell quantum graph with the default parameters, plotted in Fig. 3.1(a):

```
G=quantumGraphFromTemplate('dumbbell');
G.applyFunctionsToAllEdges({@sin,@(x)exp(-(x-2).^2),0});
G.plot
```



(a) (b)

Fig. 3.1: (a) A function defined on the edges of a dumbbell graph. (b) A function defined on the edges of a tetrahedral graph.

QGLAB also provides some three-dimensional templates, for which the `y` values are plotted using false color, as in Fig 3.1(b), where we plot Gaussians on all edges of a regular tetrahedron using the commands:

```
Phi=solidTemplates('tetrahedron');
f=@(x)(exp((-10*(x-.5).^2)));
Phi.applyFunctionsToAllEdges({f,f,f,f,f,f});
Phi.plot;
```

Finally, on some graphs with many edges, plots in three dimensions become a confusing tangle of curves, and it is more illuminating to visualize them in false color, using an overloaded `pcolor` command, demonstrated in Sec. B.

3.3.2. Getting data on and off the graph. The discretized Poisson problem (2.14) and the discretized eigenvalue problem (2.25), and other discretized problems described below, are all posed in terms of unknown column vectors. By contrast, this data is stored edge by edge in the `quantumGraph` object in `G.qg.Edges.y`. The command `graph2column` creates such a column vector from the data in graph `G`, while the command `column2graph` loads the data from a column vector onto the edges of the graph including the vertices (Chebyshev) or ghost points (Uniform). Under the uniform discretization, the command also interpolates the data to the vertices. The `column2graph` command is also called by the command `G.plot(y)` to plot the contents of the vector `y` over the skeleton defined by the graph.

3.3.3. Other important overloaded functions. Many methods from MATLAB's directed graphs class have been overloaded so that, for example, a call to `G.numnodes` returns the number of nodes and `G.numedges` the number of edges. An overloaded `spy` command (along with additional formatting) was used to visualize the Laplacian matrix in Figs. 2.2 and Fig. 2.4. Overloaded versions of the `norm` and `dot` commands are used frequently throughout the package. Sec. B gives a complete listing of the functions.

4. Conclusions. QGLAB is a robust and versatile MATLAB package for the automated computational solution to spectral accuracy of linear and nonlinear problems on Quantum Graphs. It may be used to quickly build graph models, analyze their spectrum, compute nonlinear bifurcations, and solve evolutionary equations. Algorithms for constructing graphs, solving problems on them, and manipulating and visualizing solutions are implemented at a high level, hiding most details of the implementation from the end user and allowing them to focus on the mathematical problem and not the numerical and algorithmic details.

Linear and nonlinear PDEs on quantum graphs remain a vibrant area of analysis in spectral geometry in which the interaction of geometry, topology, and symmetry gives rise to diverse mathematical questions [7, 16, 44] with many open problems left to explore. They arise in models for condensed matter physics [14, 77], they model dynamics that occur in thin or fiber domains such as carbon nanotubes [40, 55, 56], they can serve as models for network analysis [66], and they can be used to model continuum operators on manifolds [51].

Many previously studied problems on combinatorial graphs have analogies on metric graphs that remain open and where the spectrum of behaviors is likely to be much richer. For example, the spectral optimization of combinatorial graphs has been studied in [69], and others have examined how the symmetries of discrete Laplacians can lead to interesting spectral features such as Dirac points and flat bands [57, 62]. The study of time-dependent evolution equations on quantum graphs remains in its infancy [37, 52]. QGLAB, which allows the quick setup and numerical analysis of such problems, is an ideal tool for exploring these problems.

Supplementary Material

This appendix contains two sections. The first, Section A, 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 A.1 and evolutionary PDE problems in Section A.2. All the examples are included as live scripts (MATLAB `.mlx` files) in the directory `source/examples`. The second part, Sec. B, contains a complete listing of user-callable function definitions and explicit instructions for their use.

Appendix A. Extended examples.

A.1. Stationary problems.

A.1.1. Eigenproblems. QGLAB overloads MATLAB's `eigs` function, which computes a finite number of eigenvalues and eigenvectors of the discretized Laplacian defined by the generalized eigenvalue problem (2.25) defined for either the uniform or Chebyshev discretization. The eigenfunctions returned are normalized to have unit L^2 norm defined over the graph and, if single-signed, to be positive. Basic eigenfunction calculations are described in detail in the live script `starEigenfunctionsDemo.mlx`, which considers a star-shaped graph with three finite edges of lengths $\left\{\frac{3}{2}, 1, 1\right\}$ connected to a central vertex, with Kirchhoff conditions at the central vertex and the end of the longer edge $\mathtt{e}_1$ and Dirichlet condition at the remaining two vertices $\mathtt{v}_3$ and $\mathtt{v}_4$:

```
LVec=[1.5 1 1]; nX = 40; rC = [0 0 nan nan];
G = quantumGraphFromTemplate('star','LVec',LVec,'nX',nX,'robinCoeff',rC);
[V,lambda]=eigs(Phi,4); % Compute 4 eigenvalues
```

The eigenfunctions are stored as columns of the array `V` and are plotted in Fig. 2.5, using, for example, the command G.plot(V(:,1)). The eigenvalues are consistent with the symbolic secular determinant

$$\Sigma(k) = \frac{16}{3}\sin\frac{k}{2}\left(1-\sin^2\frac{k}{2}\right)\left(1-9\sin^2\frac{k}{2}+12\sin^4\frac{k}{2}\right),$$

computed using the command G.secularDet, which is plotted in Fig. A.1, along with the computed values of $k=\sqrt{-\lambda}$. In particular, the "third" eigenvalue $\lambda = -\pi^2 \approx -0.9865$ has multiplicity two, as this plot shows. In general, a numerical eigenvalue solver will return two eigenvalues very closely spaced rather than a double eigenvalue. The graph Γ in this example is symmetric under the interchange of the edges $\mathtt{e}_2$ and $\mathtt{e}_3$, and thus its Laplacian operator is too. The multiplicity-one eigenfunctions respect this symmetry, but the multiplicity-two eigenfunctions returned by `eigs` do not. The live script contains code that takes appropriate linear combinations of the two computed eigenvectors to produce eigenvectors that are odd and even with respect to this symmetry. These are shown in Fig 2.5 above. In that figure, we plot the ground state eigenfunction and the first three excited states of a Y-shaped quantum graph with Dirichlet conditions at the ends of the two shorter edges. The two rightmost images show eigenfunctions that are, respectively, even and odd, with respect to the interchange of edges $\mathtt{e}_2$ and $\mathtt{e}_3$. The well-known Mathworks membrane logo is the ground state of the Laplace operator on an L-shaped region with Dirichlet boundary conditions. We have adopted an equivalent logo for QGLAB, shown in Fig A.2.

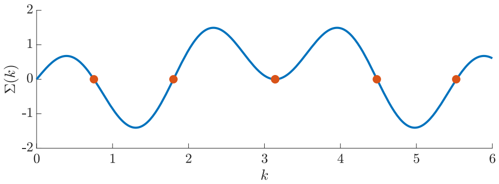


Fig. A.1: The secular determinant $\Sigma(k)$, of the Y-shaped graph discussed in the text, along with the computed values $k_j = \sqrt{-\lambda_j}$, which sit right on top of the zeros. Also, note that the second excited state has multiplicity two.

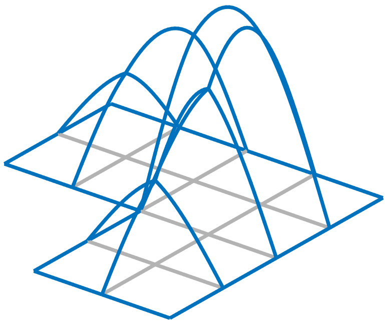


Fig. A.2: The QGLAB logo, the ground state eigenfunction of an L-shaped quantum graph with Dirichlet conditions at the exterior vertices.

A.1.2. Poisson problems. The method `solvePoisson` solves the Poisson problem (2.10). The live script `poissonExample.mlx` demonstrates this solver on a quantum graph composed of three vertices and four edges, two of which are loops, as shown in Fig A.3. At the two internal vertices, it satisfies an inhomogeneous Robin-Kirchhoff condition, while at the one pendant vertex, it satisfies an inhomogeneous Dirichlet condition:

$$\sum_{\mathtt{e}_m \in \mathcal{V}_1} \psi'_m(\mathtt{v}_1) + \Psi(\mathtt{v}_1) = 1; \; \sum_{\mathtt{e}_m \in \mathcal{V}_2} \psi'_m(\mathtt{v}_1) + \Psi(\mathtt{v}_2) = 2; \; \Psi(\mathtt{v}_3) = 3.$$

The right hand sides are given by $f = \{\cos x, x, \sin 2x, 1\}$ on the four edges. The following lines summarize the live script, where the variables `f` and `phi` are the Poisson data on the edges and vertices, respectively, and the exact solution can be read off from the listing:

```
s=[1 1 2 2]; t=[1 2 2 3]; L=[2*pi 4 2*pi 1]; robinCoeff=[1 1 nan];
    nx = 10;
phi = (1:3)'; % The nonhomogeneous term for the vertex condition
G=quantumGraph(s,t,L,'RobinCoeff',robinCoeff,'nxVec',nx,'nodeData'
    ,phi);
f = G.applyFunctionsToAllEdges({@(x)cos(x);@(x)x;@(x)sin(2*x);1});
numericalSolution = G.solvePoisson('edgeData',f);
psiExact={@(x)(19/2-cos(x)); @(x)(51-45*x+x.^3)/6;...
          @(x)(-130-3*sin(2*x))/12; @(x)(-65+80*x+3*x.^2)/6};
exactSolution = G.applyFunctionsToAllEdges(psiExact);
error=G.norm(exactSolution-numericalSolution);
```

This returns a value of error=0.0089, which decreases by approximately a factor of four to error=0.0022 when we set nx=20, providing empirical evidence of second-order convergence, as expected. A second live script `poissonExampleChebyshev.mlx` changes the third line to set the discretization to `Chebyshev` and performs two discretizations, with `n=16` and `n=32` points per edge. In this case, the errors in the computed solution are 2.07×10^{-8} and 5.93×10^{-11}, a reduction of about 349 times, which seems consistent with spectral convergence.

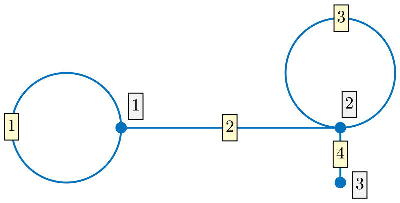


Fig. A.3: The graph used in the Poisson equation example of Sec. A.1.2.

A.1.3. Nonlinear standing waves and bifurcation diagrams.

Computing individual solutions. We begin with an example computing a single solution to the stationary cubic NLS (1.9) on a dumbbell graph:

```
45 Phi = quantumGraphFromTemplate('dumbbell');
46 fcns = getNLSFunctionsGraph(Phi);
47 Lambda = -1;
48 f = @(z)fcns.f(z,Lambda);
49 M = @(z)fcns.fLinMatrix(z,Lambda);
50 y0 = Phi.applyFunctionsToAllEdges({0,@(x)sech((x-2)),0});
51 y = solveNewton(y0,f,M); Phi.plot(y)
```

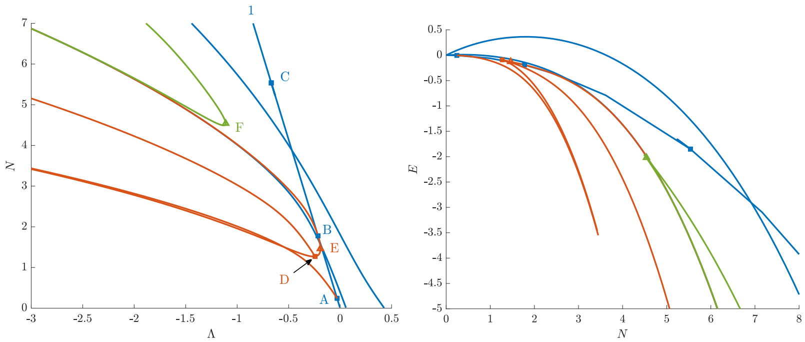
The function getNLSFunctionsGraph defines the discretized version of the nonlinear functional and several of its partial derivatives and assigns them to a structure array called fcns. By default, this uses the function $f(z) = 2z^3$ from Eq. (1.9). Still, the user may provide a symbolic function of one 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. (1.9) as a function of both Ψ and Λ, but in this first example, we fix $\Lambda = -1$ and consider only Ψ as unknown. In lines 4 and 5, *anonymous functions* are used to instruct MATLAB to consider them as functions of Ψ alone. We search for a unimodal solution to Eq. (1.9) with $\Lambda = 1$ centered on the central edge of a dumbbell graph, so we prepare an initial guess in line 6 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 [28–30, 79]. The capabilities of QGLAB are much more modest but allow for the simple setup and solution to basic continuation and bifurcation problems on quantum graphs.

An extended example on 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. A.4, reproducing a figure from [47], which contains far more details and graphs of several of the solutions at various points on the bifurcation diagram.



(a) (b)

Fig. A.4: (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.

This figure comprises nine separately-computed curves, each representing dozens of solutions to Eq. (1.9). 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

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

It is straightforward to show that if λ is an eigenvalue of the operator $-\triangle$, then branch **1** has a bifurcation point at $\Lambda = -\lambda/2$ [47, 60]. 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**,

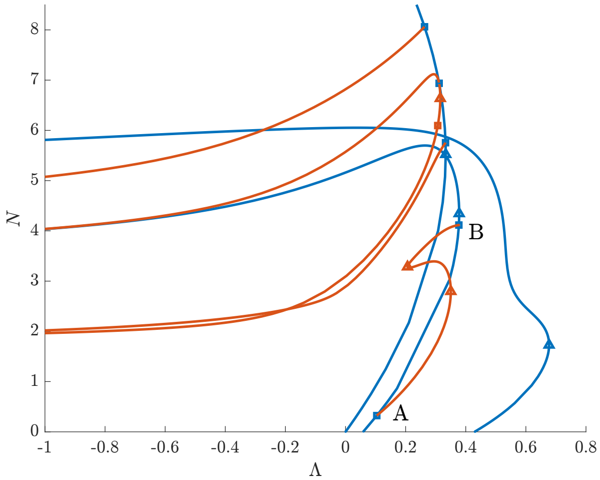


Fig. A.5: A partial bifurcation diagram of the stationary NLS equation on a dumbbell quantum graph with a cubic-quintic nonlinearity.

QGLAB has detected bifurcation points on branch **1**, and we have chosen to 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 (1.10), as shown in the right image of Fig. A.4.

The nonlinear term in stationary NLS (1.9) 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. A.5, 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 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 Appendix B.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. [20]. 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. (A.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 [43].

The double-zero eigenvalue at this bifurcation has two orthogonal eigenfunctions plotted in Fig. A.6. 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 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. A.4, we confirm the statement of Ref. [20] 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

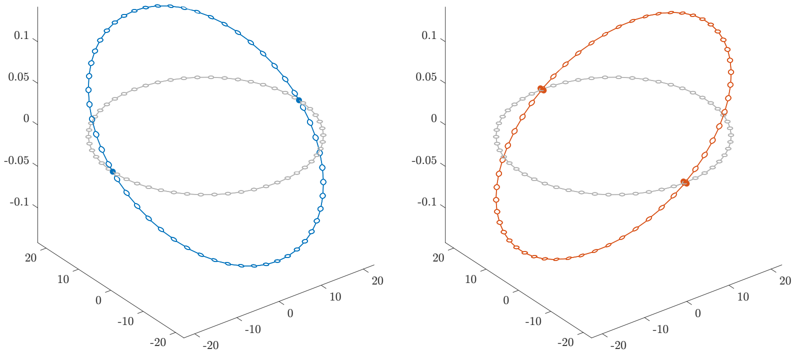


Fig. A.6: The eigenfunctions corresponding to the smallest nonzero eigenvalue on the necklace quantum described in the text. The left eigenfunction has two nodes on "strings" and four local extrema on "pearls", while the right eigenfunction has four nodes on "pearls" and two local extrema on "strings."

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.

A.2. Evolutionary PDE.

A.2.1. Simple methods. Before describing the use of the solvers described in Sec. 2.4, we use QGLAB to construct some basic solvers of the type seen in an elementary class on numerical PDE to demonstrate the simplicity of setting up and solving initial value problems of the heat and wave equations.

The Crank-Nicholson method for the heat equation. A common method to solve the heat equation

$$\frac{\partial u}{\partial t} = \triangle u \tag{A.2}$$

is the Crank-Nicholson method, which iterates

$$\left(\mathbf{I} - \frac{h}{2}\mathbf{L}\right)\mathbf{u}_{n+1} = \left(\mathbf{I} + \frac{h}{2}\mathbf{L}\right)\mathbf{u}_n.$$

Where $\mathbf{u}_n$ is the discretized solution at time $t_n = nh$ and $\mathbf{L}$ is the discretized Laplacian matrix. In QGLAB, this is evaluated on the interior grid to give

$$\left(\mathbf{P}_{\text{int}} - \frac{h}{2}\mathbf{L}_{\text{int}}\right)\boldsymbol{\psi}_{n+1} = \left(\mathbf{P}_{\text{int}} + \frac{h}{2}\mathbf{L}_{\text{int}}\right)\boldsymbol{\psi}_n.$$

Combining this with homogeneous vertex conditions yields

$$\left(\mathbf{P}_{\text{VC}} - \frac{h}{2}\mathbf{L}_{\text{VC}}\right)\boldsymbol{\psi}_{n+1} = \left(\mathbf{P}_0 + \frac{h}{2}\mathbf{L}_0\right)\boldsymbol{\psi}_n.$$

We solve this problem on the dumbbell graph in the live script `heatOnDumbbell`. After removing the code for plotting and calculating the conserved total heat, the code reads

```
Phi=quantumGraphFromTemplate('dumbbell');
y=Phi.applyFunctionsToAllEdges({@(x)(2-2*cos(x-pi/3)),1,@cos
    });
dt=0.01; tFinal=10; nStep=tFinal/dt;
Lint=Phi.wideLaplacianMatrix;
Pint=Phi.interpolationMatrix;
Lplus=Phi.extendWithZeros(Pint+dt/2*Lint);
Lminus=Phi.extendWithVC(Pint-dt/2*Lint);
for k=1:nStep
    y = Lminus\(Lplus*y);
end
```

This solution's initial and final states are shown in Fig. A.7. The total heat is conserved to twelve digits by this calculation.

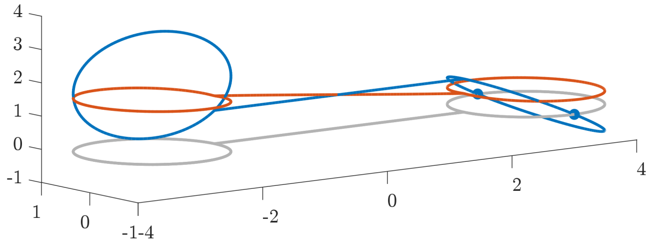


Fig. A.7: The initial (blue) and final (red) states of the heat equation on a dumbbell graph computed using the Crank-Nicholson code in the text.

The leapfrog method for nonlinear Klein-Gordon equations. The time-stepping methods described in Sec. 2.4 are only set up to solve ODE of the form (2.26) and cannot handle equations of higher order in time. The following example, contained in the live script `sineGordonOnTetra.mlx`, illustrates as a second-order example the sine-Gordon equation on the tetrahedron quantum graph, considered previously in [37], which is formed from a wave equation provided with a sinusoidal potential,

$$\Psi_{tt} - \triangle\Psi + \sin\Psi = 0. \tag{A.3}$$

As with other problems, we first discretize the equation in time only with a time step h, so that Ψ_n is the solution at discrete time $t = nh$. Applying second-order centered differences in time gives an iteration of the form

$$\Psi_{n+1} = \Psi_n + (\Psi_n - \Psi_{n-1}) + h^2 \left(\triangle \Psi_n - \sin \Psi_n\right). \tag{A.4}$$

Applying the discretization in space gives

$$\mathbf{P}_{\text{int}} \boldsymbol{\psi}_{n+1} = \mathbf{P}_{\text{int}} \left(\boldsymbol{\psi}_n + (\boldsymbol{\psi}_n - \boldsymbol{\psi}_{n-1}) - h^2 \sin \boldsymbol{\psi}_n\right) + h^2 \mathbf{L}_{\text{int}} \boldsymbol{\psi}_n \tag{A.5}$$

and enforcing the vertex conditions gives

$$\mathbf{P}_{\text{VC}} \boldsymbol{\psi}_{n+1} = \mathbf{P}_0 \left(\boldsymbol{\psi}_n + (\boldsymbol{\psi}_n - \boldsymbol{\psi}_{n-1}) - h^2 \sin \boldsymbol{\psi}_n\right) + h^2 \mathbf{L}_0 \boldsymbol{\psi}_n.$$

The initial conditions $\boldsymbol{\psi}|_{t=0} = \boldsymbol{\psi}_0$ and $\frac{\partial}{\partial t} \boldsymbol{\psi}|_{t=0} = \boldsymbol{\phi}_0$ are given. Because the leapfrog scheme is a multistep method, it requires an approximation to $\boldsymbol{\psi}|_{t=h}$ that is accurate to $O(h^2)$ and satisfies the vertex conditions. This solves

$$\mathbf{P}_{\text{VC}} \boldsymbol{\psi}_1 = \mathbf{P}_0 \left(\boldsymbol{\psi}_0 + h \boldsymbol{\phi}_0 - \frac{h^2}{2} \sin \boldsymbol{\psi}_0\right) + \frac{h^2}{2} \mathbf{L}_0 \boldsymbol{\psi}_0.$$

The line of the live script that executes the time-stepper in Eq. (A.5) is

```
u2 = PVC\(P0*(u1 + (u1-u0) - dt^2*sin(u1)) + dt^2*L0*u1);
```

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 [37], 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. A.8, 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 reflected after encountering vertices, while in the second case, the faster solitons can pass through the vertices.

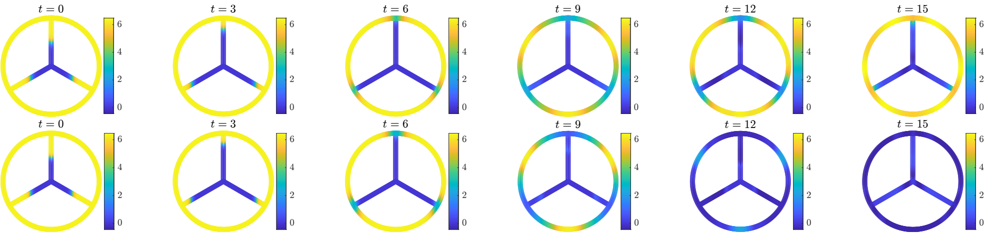


Fig. A.8: Evolution of sine-Gordon solitons propagating along the edges of a tetrahedron. (Top) the vertices reflect solitons with $c = 0.9$ while (Bottom) those with $c = 0.95$ are transmitted.

A.2.2. The DAE formulation using MATLAB's ODE solvers. Kairzhan et al. consider the cubic NLS equation posed on a "star graph" consisting of three half-lines joined at a single vertex [52]. The evolution conserves mass, i.e., the squared L^2 norm (1.4), and the energy (1.10), but in general, does not conserve momentum. If, however, the parameters w_m are appropriately chosen in the weighted Kirchhoff-Robin vertex condition (1.2) to form a so-called "balanced" star graph and the initial condition is chosen to lie in a particular invariant subspace, then the dynamics on this graph reduce to the dynamics of the NLS equation on the whole line, and the momentum is conserved, as are all other conserved quantities.

We simulate the collision of a soliton propagating on edge $\mathbf{e}_1$ toward the vertex results in Fig. A.9 (an expanded version of Fig. 2.8 above) using the DAE solver `qgde23t`, which is recommended for moderately stiff problems. The first simulation is computed on a "balanced" graph whose vertex condition is defined by the weight vector $\mathbf{w} = (2, 1, 1)$ in the first line of the following script.

```
Phi = quantumGraphFromTemplate('star','LVec',8*pi,'weight',[2
    1 1]);
init1 = @(x)  exp(1i*x).*sech(x-4*pi);
init2 = @(x) exp(-1i*x).*sech(x+4*pi);
u0 = Phi.applyFunctionsToAllEdges({init1,init2,init2});
mu = -1i;
F =@(t,z) -2i * z.^2 .* conj(z);
[t,u] = Phi.qgde23t(mu,F,0:.5:11,u0,'AbsTol',1e-6,'RelTol',1e
    -4);
```

In the second simulation, the weight vector is changed to $\mathbf{w} = (1, 1, 1)$. In the first simulation, the soliton splits into two, each new soliton propagating along the edge with its original amplitude and velocity, while in the second simulation, much of the soliton's energy is reflected and propagates backward along the incoming edge. Both simulations conserve the energy and mass to about four digits, while only the first simulation conserves momentum.

For stiff systems such as the KPP equation, a heat equation with quadratic nonlinearity,

$$u_t = \mu \triangle u + u(1-u),$$

QGLAB provides the implicit solver `qgde15s`. This system arises, for example, as a model of species spread in an ecosystem. We consider the solution of this equation with $\mu = 1$ on a honeycomb graph in the demonstration `KPPonHoneycomb` and in Fig. 2.9, which may be thought of as showing the spread of a species along a road network.

A.2.3. Fixed-step Runge-Kutta solvers. The DAE solver that uses routines from the MATLAB ODE suite can be replaced by the fixed-step fully implicit Runge-Kutta solver `qgdeN34DIRK` or the implicit-explicit RK solver `qgdeSDIRK443`. Without documenting a full convergence or timing study, we may report that, on the first author's M1 Mac laptop, the simulation reported in Fig. A.9 took about 3 seconds of run time. Choosing the time step small enough to achieve the same accuracy in the conserved quantities, the fully implicit method took about 1500 seconds, while the implicit-explicit method took only 1.5 seconds—the clear winner in this contest.

Appendix B. Function Listing and Detailed Instructions.

QGLAB is implemented as a MATLAB *Project*. After starting MATLAB, the user should open

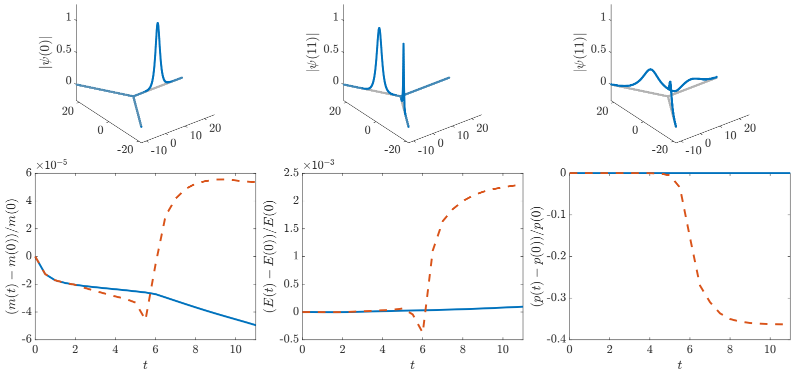


Fig. A.9: Collision of an NLS soliton with the vertex of a star graph. **Top:** The absolute value of the solution ψ. (Left) Initial time. (Center) Final time, balanced graph. (Right) Final time, unbalanced graph. **Bottom:** Relative changes in the computed conserved quantities (Left) Mass. (Center) Energy. (Right) Momentum. The solid blue line shows the balanced graph, and the dashed red line shows the unbalanced graph. While both the energy and mass are well conserved by the evolution in both problems, only the balanced graph conserves the momentum.

the folder titled `Quantum-Graphs`, whose subfolder structure is shown in Fig. B.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).

B.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(j)` 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 $1 \leq m \leq |\mathcal{V}|$ 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.

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

Fig. B.1: Directory structure of QGLAB

- `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. (1.2). Use the value `NaN` to indicate the Dirichlet boundary condition (1.3). If scalar, apply the same value at all vertices. Default: 0.
- `Weight` The vector of weights w_m in Eq. (1.2). 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.

B.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. (2.11) 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. (2.12), 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. (2.11), (2.12) 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. (2.13) 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.

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

B.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(Phi.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.`

B.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.

- `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 is used to create initial guesses for the nonlinear standing wave solvers.
- `addEdgeField` and `addNodeField`

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

- `integral` Computes the weighted integral $\int_\Gamma \Psi \, \mathrm{d}x = \sum_{m=1}^{|\mathcal{E}|} \int_{\mathsf{e}_m} \psi_m(x) \, \mathrm{d}x$.
- `norm` Uses `integral` to compute the L^p norm (1.4).
- `dot` Uses `integral` to compute the L^2 inner product (1.5).
- `energyNLS` Uses `integral` to compute the NLS energy (1.10).
- `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 a function defined over the edges of a randomly generated Delaunay triangulation, shown in Fig. B.2.

```
Phi=delaunaySquare('n',8);
f=@(x1,x2)sin(2*pi*x1).*sin(2*pi*x2);
Phi.applyGraphicalFunction(f);
Phi.plot; figure; Phi.pcolor
```

- `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.

B.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

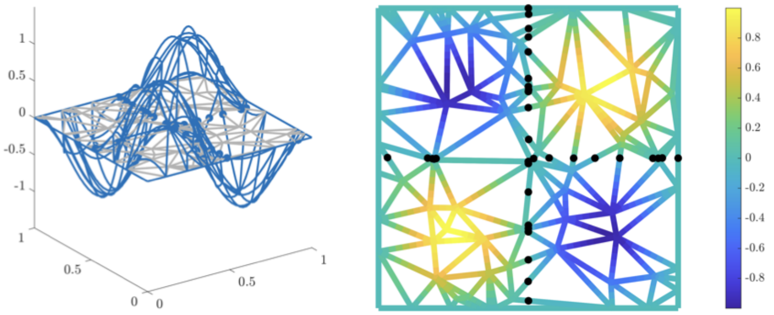


Fig. B.2: Visualization of a function defined on a random graph using (left) `plot` and (right) `pcolor`, where zeros are indicated with black dots.

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*pi 4*pi 2*pi])
```

is shown in Fig. B.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 [1, 5, 6]. The underlying symmetry of the construction here is crucial to the analysis.

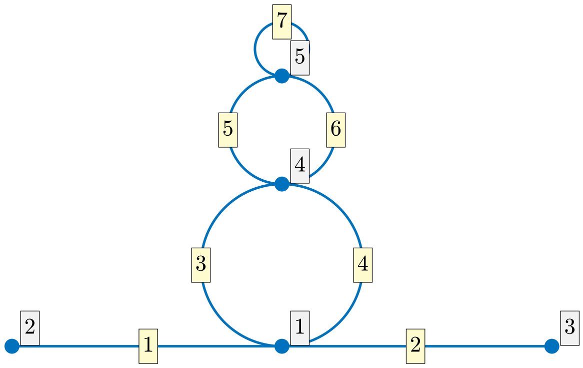


Fig. B.3: The default `bubbleTower` quantum graph, with five vertices and seven edges.

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`.

B.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.

B.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 (1.9), as well as several derivatives of this function, and the antiderivative of the nonlinearity, used in computing the energy.
- `continuerSet` This function is used to set 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 value: 4°.
  - `minNormDelta` The minimum step length below which the continuation solver does not attempt to refine the branch further. Default value: 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 value: 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 value: 4.
  - `LambdaThresh` Threshold for the frequency Λ, so the continuation routine terminates when this value is crossed. Default value: -1.
  - `maxPoints` The maximum number of points to compute on a given branch. Default value: 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 to 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 solution based on an initial guess that places a "bump" somewhere on the graph defined from its plotting coordinates, a 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 branch 1.
- `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 text file containing one-word 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`.

B.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.

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

[2] R. Adami, C. Cacciapuoti, D. Finco, and D. Noja, *Fast solitons on star graphs*, Rev. Math. Phys., 23 (2011), pp. 409–451.

[3] R. Adami, C. Cacciapuoti, D. Finco, and D. Noja, *Stationary states of NLS on star graphs*, Europhys. Lett., 100 (2012), p. 10003.

[4] R. Adami, E. Serra, and P. Tilli, *Threshold phenomena and existence results for NLS ground states on metric graphs*, J. Funct. Anal., 271 (2016), pp. 201–223.

[5] 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.

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

[7] L. Alon, R. Band, and G. Berkolaiko, *Nodal statistics on quantum graphs*, Commun. Math. Phys., 362 (2018), pp. 909–948.

[8] M. Arioli and M. Benzi, *A finite element method for quantum graphs*, IMA J. Numer. Anal., 38 (2018), pp. 1119–1163.

[9] U. M. Ascher, S. J. Ruuth, and R. J. Spiteri, *Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations*, Appl. Numer. Math., 25 (1997), pp. 151–167.

[10] J. L. Aurentz and L. N. Trefethen, *Block operators and spectral discretizations*, SIAM Review, 59 (2017), pp. 423–446.

[11] R. Band, G. Berkolaiko, H. Raz, and U. Smilansky, *The number of nodal domains on quantum graphs as a stability index of graph partitions*, Commun. Math. Phys., 311 (2012), pp. 815–838.

[12] T. Beck, I. Bors, G. Conte, G. Cox, and J. L. Marzuola, *Limiting eigenfunctions of Sturm–Liouville operators subject to a spectral flow*, Ann. Math. Que., (2020), pp. 1–21.

[13] S. Becker, F. Gregorio, and D. Mugnolo, *Schrödinger and polyharmonic operators on infinite graphs: Parabolic well-posedness and p-independence of spectra*, J. Math. Anal. Appl., 495 (2021), p. 124748.

[14] S. Becker and M. Zworski, *Magnetic oscillations in a model of graphene*, Commun. Math. Phys., 367 (2019), pp. 941–989.

[15] G. Berkolaiko, *An elementary introduction to quantum graphs*, in Geometric and Computational Spectral Theory, vol. 700 of Contemp. Math., Amer. Math. Soc., Providence, RI, 2017, pp. 41–72.

[16] G. Berkolaiko, J. Kennedy, P. Kurasov, and D. Mugnolo, *Surgery principles for the spectral analysis of quantum graphs*, Trans. Am. Math. Soc., 372 (2019), pp. 5153–5197.

[17] G. Berkolaiko and P. Kuchment, *Introduction to Quantum Graphs*, vol. 186 of Mathematical Surveys and Monographs, American Mathematical Soc., Providence, RI, 2013.

[18] G. Berkolaiko, J. L. Marzuola, and D. E. Pelinovsky, *Edge-localized states on quantum graphs in the limit of large mass*, Ann. Henri Poincaré C, 38 (2020), pp. 1295–1335.

[19] J.-P. Berrut and L. N. Trefethen, *Barycentric Lagrange interpolation*, SIAM Review, 46 (2004), pp. 501–517.

[20] 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.

[21] W. Borrelli, R. Carlone, and L. Tentarelli, *Nonlinear Dirac equation on graphs with localized nonlinearities: bound states and nonrelativistic limit*, SIAM J. Math. Anal., 51 (2019), pp. 1046–1081.

[22] W. Borrelli, R. Carlone, and L. Tentarelli, *An overview on the standing waves of nonlinear Schrödinger and Dirac equations on metric graphs with localized nonlinearity*, Symmetry, 11 (2019), p. 169.

[23] D. Borthwick, E. M. Harrell II, and K. Jones, *The heat kernel on the diagonal for a compact metric graph*, in Ann. Henri Poincaré, Springer, 2022, pp. 1–20.

[24] C. Cacciapuoti, S. Dovetta, and E. Serra, *Variational and stability properties of constant solutions to the NLS equation on compact metric graphs*, Milan J. Math., 86 (2018), pp. 305–327.

[25] C. Cacciapuoti, D. Finco, and D. Noja, *Ground state and orbital stability for the NLS equation on a general starlike graph with potentials*, Nonlinearity, 30 (2017), p. 3271.

[26] G. Conte, *Numerical Analysis of Linear and Nonlinear Schrödinger Equations on Quantum Graphs*, PhD thesis, University of North Carolina, 2022.

[27] C. De Coster, S. Dovetta, D. Galant, and E. Serra, *On the notion of ground state for nonlinear Schrödinger equations on metric graphs*, arXiv preprint arXiv:2301.08001, (2023).

[28] 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.

[29] 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.

[30] 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.

[31] S. Dovetta, *Existence of infinitely many stationary solutions of the L^2-subcritical and critical NLSE on compact metric graphs*, J. Differ. Equations, 264 (2018), pp. 4806–4821.

[32] S. Dovetta, *Mass-constrained ground states of the stationary NLSE on periodic metric graphs*, NoDEA, 26 (2019), pp. 1–30.

[33] S. Dovetta and L. Tentarelli, *Ground states of the L^2-critical NLS equation with localized nonlinearity on a tadpole graph*, in Discrete and Continuous Models in the Theory of Networks, Springer, 2020, pp. 113–125.

[34] T. A. Driscoll and N. Hale, *Rectangular spectral collocation*, IMA J. Numer. Anal., 36 (2016), pp. 108–132.

[35] T. A. Driscoll, N. Hale, and L. N. Trefethen, *Chebfun Guide*, Pafnuty Publications, 2014.

[36] Y. Du, B. Lou, R. Peng, and M. Zhou, *The Fisher-KPP equation over simple graphs: varied persistence states in river networks*, J. Math. Biol., 80 (2020), pp. 1559–1616.

[37] 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.

[38] L. Edsberg, *Introduction to Computation and Modeling for Differential Equations*, Wiley, 2016.

[39] P. Exner, J. P. Keating, P. Kuchment, A. Teplyaev, and T. Sunada,

Analysis on Graphs and Its Applications: Isaac Newton Institute for Mathematical Sciences, Cambridge, UK, Jan. 8-Jun. 29, 2007, vol. 77 of Proc. Symposia Pure Math., American Mathematical Soc., 2008.

[40] P. Exner and O. Post, *Convergence of spectra of graph-like thin manifolds*, J. Geom. Phys., 54 (2005), pp. 77–115.

[41] P. Exner and O. Post, *Approximation of quantum graph vertex couplings by scaled Schrödinger operators on thin branched manifolds*, J. Phys. A: Math. Theor., 42 (2009), p. 415305.

[42] P. Exner and O. Post, *A general approximation of quantum graph vertex couplings by scaled Schrödinger operators on thin branched manifolds*, Commun. Math. Phys., 322 (2013), pp. 207–227.

[43] 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.

[44] S. Gnutzmann and U. Smilansky, *Quantum graphs: Applications to quantum chaos and universal spectral statistics*, Adv. Phys., 55 (2006), pp. 527–625.

[45] S. Gnutzmann and D. Waltner, *Stationary waves on nonlinear quantum graphs: General framework and canonical perturbation theory*, Phys. Rev. E, 93 (2016), p. 032204.

[46] N. Goloshchapova, *A nonlinear Klein–Gordon equation on a star graph*, Math. Nachr., 294 (2021), pp. 1742–1764.

[47] 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.

[48] W. J. F. Govaerts, *Numerical Methods for Bifurcations of Dynamical Equilibria*, SIAM, 2000.

[49] D. Grieser, *Spectra of graph neighborhoods and scattering*, Proc. London Math. Soc., 97 (2008), pp. 718–752.

[50] E. Harrell, *Spectral theory on combinatorial and quantum graphs*, in Théorie spectrale des graphes et des variétés, Kairouan, Tunisia, C. Anné and N. Torki-Hamza, eds., CIMPA, 2016.

[51] S. Holden and G. Vasil, *A continuum limit for the Laplace operator on metric graphs*, arXiv preprint arXiv:2301.07086, (2023).

[52] A. Kairzhan, D. E. Pelinovsky, and R. H. Goodman, *Drift of spectrally stable shifted states on star graphs*, SIAM J. Appl. Dyn. Syst., 18 (2019), pp. 1723–1755.

[53] H. B. Keller, *Numerical solution of bifurcation and nonlinear eigenvalue problems*, in Applications of Bifurcation Theory, P. Rabinowitz, ed., vol. 84 of Publ. Math. Res. Center Univ. Wisconsion, Academic Press, 1977.

[54] T. Kottos and U. Smilansky, *Quantum chaos on graphs*, Phys. Rev. Lett., 79 (1997), p. 4794.

[55] P. Kuchment, *Graph models for waves in thin structures*, Waves in Random Media, 12 (2002), pp. R1–R24.

[56] P. Kuchment and O. Post, *On the spectra of carbon nano-structures*, Comm. Math. Phys., 275 (2007), pp. 805–826.

[57] L.-K. Lim, J.-N. Fuchs, F. Piéchon, and G. Montambaux, *Dirac points emerging from flat bands in Lieb-Kagome lattices*, Phys. Rev. B., 101 (2020), p. 045131.

[58] D. Maier, W. Reichel, and G. Schneider, *Breather solutions for a semilinear Klein-Gordon equation on a periodic metric graph*, J. Math. Anal. Appl., 528 (2023), p. 127520.

[59] G. Malenova, *Spectra of quantum graphs*, master's thesis, Czech Technical University in Prague, 2013.

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

[61] Mathworks, *Choose an ODE solver*. https://www.mathworks.com/help/matlab/math/choose-an-ode-solver.html. accessed 9/15/2022.

[62] L. Morales-Inostroza and R. A. Vicencio, *Simple method to construct flat-band lattices*, Phys. Rev. A, 94 (2016), p. 043831.

[63] D. Mugnolo, *Semigroup methods for evolution equations on networks*, Understanding Complex Systems, Springer Cham, 2014.

[64] D. Mugnolo, D. Noja, and C. Seifert, *Airy-type evolution equations on star graphs*, Analysis & PDE, 11 (2018), pp. 1625–1652.

[65] A. H. Nayfeh and B. Balachandran, *Applied nonlinear dynamics: Analytical, computational and experimental methods*, Wiley-VCH, 1995.

[66] S. Nicaise, *Some results on spectral theory over networks, applied to nerve impulse transmission*, in Polynômes Orthogonaux et Applications: Proc. Laguerre Sympos, Bar-le-Duc, Oct 15–18, 1984, Springer, 1985, pp. 532–541.

[67] D. Noja, *Nonlinear Schrödinger equation on graphs: recent results and open problems*, Phil. Trans. R. Soc. A, 372 (2014), p. 20130002.

[68] D. Noja, D. Pelinovsky, and G. Shaikhova, *Bifurcations and stability of standing waves in the nonlinear Schrödinger equation on the tadpole graph*, Nonlinearity, 28 (2015), p. 2343.

[69] B. Osting and J. Marzuola, *Spectrally optimized pointset configurations*, Constructive Approximation, 46 (2017), pp. 1–35.

[70] J. A. Pava and M. Cavalcante, *Linear instability criterion for the Korteweg–de Vries equation on metric star graphs*, Nonlinearity, 34 (2021), p. 3373.

[71] O. Post, *Spectral convergence of quasi-one-dimensional spaces*, Ann. Henri Poincaré, 7 (2006), pp. 933–973.

[72] O. Post, *Spectral analysis on graph-like spaces*, vol. 2039 of LNM, Springer Science & Business Media, 2012.

[73] K. K. Sabirov, D. Babajanov, D. U. Matrasulov, and P. G. Kevrekidis, *Dynamics of dirac solitons in networks*, J. Phys. A: Math. Theor., 51 (2018), p. 435203.

[74] E. Serra and L. Tentarelli, *Bound states of the NLS equation on metric graphs with localized nonlinearities*, J. Differ. Equations, 260 (2016), p. 5627.

[75] E. Serra and L. Tentarelli, *On the lack of bound states for certain NLS equations on metric graphs*, Nonlinear Anal. Theory Methods Appl., 145 (2016), pp. 68–82.

[76] L. F. Shampine and M. W. Reichelt, *The MATLAB ODE suite*, SIAM J. Sci. Comput., 18 (1997), pp. 1–22.

[77] S. P. Shipman, *Reducible Fermi surfaces for non-symmetric bilayer quantum-graph operators*, J. Spectral Theory, 10 (2019), pp. 33–72.

[78] Z. Sobirov, D. Babajanov, D. Matrasulov, K. Nakamura, and H. Uecker, *Sine-Gordon solitons in networks: Scattering and transmission at vertices*, Europhys. Lett., 115 (2016), p. 50002.

[79] 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.

[80] K. Xu and N. Hale, *Explicit construction of rectangular differentiation matrices*, IMA J. Numer. Anal., 36 (2016), pp. 618–632.

Acknowledgments. The authors thank Greg Berkolaiko, Dmitry Pelinovsky, David Shirokoff, and Nick Trefethen for their helpful conversations. RG credits a semester-long funded visit to the IMA at the University of Minnesota in 2017 for giving him the time and inspiration to begin working on quantum graphs.