NUMERICAL SIMULATIONS ON NONLINEAR QUANTUM GRAPHS WITH THE GRAFIDI LIBRARY

CHRISTOPHE BESSE, ROMAIN DUBOSCQ, AND STEFAN LE COZ

Abstract. Nonlinear quantum graphs are metric graphs equipped with a nonlinear Schrödinger equation. Whereas in the last ten years they have known considerable developments on the theoretical side, their study from the numerical point of view remains in its early stages. The goal of this paper is to present the Grafidi library [18], a Python library which has been developed with the numerical simulation of nonlinear Schrödinger equations on graphs in mind. We will show how, with the help of the Grafidi library, one can implement the popular normalized gradient flow and nonlinear conjugate gradient flow methods to compute ground states of a nonlinear quantum graph. We will also simulate the dynamics of the nonlinear Schrödinger equation with a Crank-Nicolson relaxation scheme and a Strang splitting scheme. Finally, in a series of numerical experiments on various types of graphs, we will compare the outcome of our numerical calculations for ground states with the existing theoretical results, thereby illustrating the versatility and efficiency of our implementations in the framework of the Grafidi library.

1. Introduction

The nonlinear Schrödinger equation

$$iu_t + \Delta_\Omega u + f(u) = 0,$$

where $u : \mathbb{R}_t \times \Omega_x \to \mathbb{C}$ is a popular model for wave propagation in Physics. It appears in particular in the modeling of Bose-Einstein condensation and in nonlinear optics. In general, the set Ω is chosen to be either the full space $\mathbb{R}^d$ (with $d = 1$ in general in optics and $d = 1, 2$ or 3 for Bose-Einstein condensation), or a subdomain of the full space. For example, in Bose-Einstein condensation, the potential might be chosen in such a way that the condensate is confined in various shapes Ω, e.g. balls or cylinders. In some cases, the shape of Ω is very thin in one direction, for example in the case of Y-junctions (see e.g. [52]), or in the case of H-junctions (see e.g. [34]). In these cases, it is natural to perform a reduction to a one-dimensional model set on a graph approximating the underlying spatial structure (see e.g. [50]).

The study of nonlinear quantum graphs, i.e. metric graphs equipped with a nonlinear evolution equation of Schrödinger type, is therefore motivated at first by applications in Physics. An overview of various applications of nonlinear Schrödinger equations on metric graphs in physical settings is proposed by Noja in [43]. One may also refer to [29, 49] for analysis of standing waves in a physical context. The validity of the graph approximation for planar branched systems was considered by Sobirov, Babadjanov and Matrasulov [50].

The mathematical aspects of nonlinear equations set on metric graphs are also interesting on their own. Among the early studies, one finds the works of Ali Mehmeti [10], see also [11]. Dispersive effects for the Schrödinger group have been considered on star graphs [41] and the tadpole graph [42]. In the last ten years, a particular theoretical aspect has attracted considerable interest: the ground states of nonlinear quantum graphs, i.e. the minimizers on graphs of the Schrödinger energy at fixed mass constraint. The literature devoted to ground states on graphs is already too vast to give an exhaustive presentation of the many works on the topic, we refer to Section 5 for a small sample of relevant examples of the existing results. There seem to be relatively few works devoted to the numerical simulations of nonlinear quantum graphs (one may refer e.g. to [15, 37, 40] which are mostly theoretical works completed with a numerical section).

In view of the sparsity of numerical tools adapted to quantum graphs, we have developed a Python library, the Grafidi library[1], which aims at rendering the numerical simulation of nonlinear quantum graphs simple and efficient.

From a conceptual point of view, the library relies on the finite difference approximation of the Laplacian on metric graphs with vertex conditions described by matrices (see Section 2.1 for details). Inside each of

Date: March 5, 2021.
This work is partially supported by the project *PDEs on Quantum Graphs* from CIMI Labex ANR-11-LABX-0040.
[1]See `https://plmlab.math.cnrs.fr/cbesse/grafidi`

the edges of the graph, one simply uses the classical second order finite differences approximation for the second derivative in one dimension. On the other hand, for discretization points close to the vertices, the finite differences approximation would involve the value of the function at the vertex, which is not directly available. To substitute for this value, we make use of (again) finite differences approximations of the boundary conditions. As a consequence, the approximation of the Laplacian of a function close to a vertex involves values of the function on each of the edges incident to this vertex. Details are given in Section 2.

The basic functionalities of the Grafidi library are presented in Section 3. The Grafidi library has been conceived with ease of use in mind and the user should not need to deal with technicalities for most of common uses. A graph is given as a list of edges, each edge being described by the labels (e.g. A, B, etc.) of the vertices that the edge is connecting and the length of the edge. With this information, the graph-constructor of the library constructs the graph and the matrix of the Laplacian on the graph with Kirchhoff (i.e. default) conditions at the vertices and a default number of discretization points. One may obviously choose to assign other types of vertices conditions, either with one the pre-implemented type (δ, δ', Dirichlet) or even with a user defined vertex condition for advanced uses. A function on the graph is then given by the collection of functions on each of the edges. The graph and functions on the graph are easily represented with commands build in the Grafidi library.

We present in Section 4 the implementation for nonlinear quantum graphs of four numerical methods popular in the simulation of nonlinear Schrödinger equations.

The first two methods that we present concern the computation of ground states, i.e. minimizers of the energy at fixed mass. Ground states are ubiquitous in the analysis of nonlinear Schrödinger equations: they are the profiles of orbitally stable standing wave solutions and serve as building blocks for the analysis of the dynamics, in particular in the framework of the *Soliton Resolution Conjecture*. The two methods that we implement are the normalized gradient flow, which was analyzed in details in our previous work [17], and the conjugate gradient flow, which was described in [12, 22] in a general domain. The idea behind these two methods is that, since the ground states are minimizers of the energy at fixed mass, they may be obtained at the continuous level by using the so-called continuous normalized gradient flow, i.e. a gradient flow corresponding to the Schrödinger energy, projected on the sphere of constant mass.

The next two methods that we present in Section 4 concern the simulation of the nonlinear Schrödinger flow on the graph. Numerical schemes for nonlinear Schrödinger equations abound, we have selected a Crank-Nicolson relaxation scheme and a Strang splitting scheme, which have both been shown to be very efficient for the simulation of the Schrödinger flow (see [16, 54]). As for the methods to compute ground states, thanks to the Grafidi library, the implementation of the time-evolution methods is not more difficult on graphs than it is in the case of a full domain.

To illustrate and validate further the use of the Grafidi library and the numerical methods presented, we have performed a series of numerical experiments in various settings in Section 5. As the theoretical literature is mainly devoted to the analysis of ground states, we have chosen to also focus on the calculations of ground states using the normalized and conjugate gradient flows. We distinguish between four categories of graphs: compact graphs, graphs with a finite number of edges and at least one semi-infinite edge, periodic graphs and trees. For each of these types of graphs, we perform ground states calculations. The comparison of the outcomes of our experiments with the existing theoretical results reveals an excellent agreement between the two.

2. Space discretization of the Laplacian on graphs

2.1. **Preliminaries.** A *metric graph* $\mathcal{G}$ is a collection of edges $\mathcal{E}$ and vertices $\mathcal{V}$. Two vertices can be connected by more than one edge (in which case we speak of *bridge*), and an edge can connect a vertex to himself (in which case we refer to the edge as *loop*). To each edge $e \in \mathcal{E}$, we associate a length l_e and identify the edge e with the interval $[0, l_e]$ ($[0, \infty)$ if $l_e = \infty$).

A *function* ψ on the graph is a collection of maps $\psi_e : I_e \to \mathbb{R}$ for each $e \in \mathcal{E}$. It is natural to define *function spaces* on $\mathcal{G}$ as direct sums of function spaces on each edge: for $p \in [1, \infty]$ and for $k = 1, 2$, we define

$$L^p(\mathcal{G}) = \bigoplus_{e \in \mathcal{E}} L^p(I_e), \quad H^k(\mathcal{G}) = \bigoplus_{e \in \mathcal{E}} H^k(I_e).$$

We denote by $(\cdot, \cdot)$ the scalar product on $L^2(\mathcal{G})$ and by $\langle \cdot, \cdot \rangle$ the duality product on $H^1(\mathcal{G})$. As no compatibility conditions have been given on the vertices yet, a function $\psi \in H^1(\mathcal{G})$ has a priori multiple values on each of the

vertices. For a vertex $v \in \mathcal{V}$, we denote by

$$\psi(v) = (\psi_e(v))_{e\sim v} \in \mathbb{R}^{d_v}$$

the vector of the values of ψ at v, where $e \sim v$ denotes the *edges incident* to v and d_v is the *degree* of v, i.e. the number of edges incident to v. In a similar way, for $\psi \in H^2(\mathcal{G})$, we denote by

$$\psi'(v) = (\psi_e'(v))_{e\sim v} \in \mathbb{R}^{d_v}$$

the vector of the outer derivatives of ψ at the vertex v. For brevity in notation, we shall also note

$$\psi(\mathcal{V}) = (\psi(v))_{v\in\mathcal{V}}, \quad \psi'(\mathcal{V}) = (\psi'(v))_{v\in\mathcal{V}},$$

the vectors constructed by the values of ψ and ψ' at each of the vertices.

To give an example, we consider the simple 3-edges star graph $\mathcal{G}_{3,sg}$ drawn on Figure 1. The degree d_O of

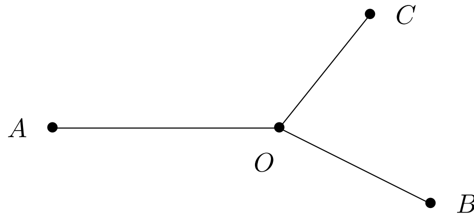


FIGURE 1. 3-edges star graph $\mathcal{G}_{3,sg}$

the vertex O is $d_O = 3$, the set of vertices is $\mathcal{V} = \{O, A, B, C\}$ and the set of edges is $\mathcal{E} = \{[OA], [OB], [OC]\}$. The vectors $\psi(O)$ and $\psi'(O)$ are given by

$$\psi(O) = \begin{pmatrix} \psi_{OA}(O) \\ \psi_{OB}(O) \\ \psi_{OC}(O) \end{pmatrix}, \qquad \psi'(O) = -\begin{pmatrix} \partial_{\mathbf{n}_{OA}}\psi_{OA}(O) \\ \partial_{\mathbf{n}_{OB}}\psi_{OB}(O) \\ \partial_{\mathbf{n}_{OC}}\psi_{OC}(O) \end{pmatrix},$$

where $\mathbf{n}_{OM} = \mathbf{OM}/\|\mathbf{OM}\|$, $M \in \{A, B, C\}$, is the inward unit vector, and

$$\partial_{\mathbf{n}_{OM}}\psi_{OM}(O) = \lim_{\substack{t\to 0\\ t>0}} \frac{\psi_{OM}(O + t\mathbf{n}_{OM}) - \psi_{OM}(O)}{t}.$$

A *quantum graph* is a metric graph $\mathcal{G}$ equipped with a *Hamiltonian* operator H, which is usually defined in the following way. The operator H is a second order unbounded operator

$$H : D(H) \subset L^2(\mathcal{G}) \to L^2(\mathcal{G}),$$

which is such that for $u \in D(H) \subset H^2(\mathcal{G})$ and for each edge $e \in \mathcal{E}$ we have

$$(Hu)_e = -u_e''. \tag{1}$$

The domain $D(H)$ of H is a subset of $H^2(\mathcal{G})$ of functions verifying specific vertex compatibility conditions, described in the following way. At a vertex $v \in \mathcal{V}$, let A_v, B_v be $d_v \times d_v$ matrices. The *compatibility conditions* for $u \in H^2(\mathcal{G})$ may then be described as

$$A_v u(v) + B_v u'(v) = 0.$$

For the full set of vertices $\mathcal{V}$, we denote by

$$A = \operatorname{diag}(A_v, v \in \mathcal{V}), \quad B = \operatorname{diag}(B_v, v \in \mathcal{V})$$

the matrices describing the compatibility conditions. The *domain* $D(H)$ of H is then given by

$$D(H) = \{u \in H^2(\mathcal{G}) : Au(\mathcal{V}) + Bu'(\mathcal{V}) = 0\}. \tag{2}$$

We will assume that A and B are such that H is self-adjoint, that is at each vertex v the $d_v \times 2d_v$ augmented matrix $(A_v|B_v)$ has maximal rank and the matrix $A_v B_v^*$ is self-adjoint. Recall that (see e.g. [14]) the boundary conditions at a vertex $v \in \mathcal{V}$ may be reformulated using three orthogonal and mutually orthogonal operators $P_{D,v}$ (D for Dirichlet), $P_{N,v}$ (N for Neumann) and $P_{R,v}$ (R for Robin) and an invertible self-adjoint operator $\Lambda_v : \mathbb{C}^{d_v} \to \mathbb{C}^{d_v}$ such that for each $u \in D(H)$ we have

$$P_{D,v}u(v) = P_{N,v}u'(v) = \Lambda_v P_{R,v}u'(v) - P_{R,v}u(v) = 0.$$

The *quadratic form* associated with H is then expressed as

$$Q(u) = \frac{1}{2}\langle Hu, u\rangle = \frac{1}{2}\sum_{e\in\mathcal{E}}\|u_e'\|_{L^2}^2 + \frac{1}{2}\sum_{v\in\mathcal{V}}(P_{R,v}u, \Lambda_v P_{R,v}u)_{\mathbb{C}^{d_v}},$$

and its *domain* is given by

$$D(Q) = H^1_D(\mathcal{G}) = \{u \in H^2(\mathcal{G}) : P_{D,v}u = 0,\ \forall v \in \mathcal{V}\}.$$

Among the many possible vertex conditions, the *Kirchhoff-Neumann condition* is the most frequently encountered. By analogy with Kirchhoff laws in electricity (preservation of charge and current), it consists at a vertex v to require:

$$u_e(v) = u_{e'}(v),\ \forall e, e' \sim v, \quad \sum_{e\sim v} u_e'(v) = 0.$$

Another popular vertex condition is the δ or *Dirac condition* of strength $\alpha_v \in \mathbb{R}$. It corresponds to continuity of the function at the vertex v, and a jump condition of size α_v on the derivatives, that is

$$u_e(v) = u_{e'}(v),\ \forall e, e' \sim v, \quad \sum_{e\sim v} u_e'(v) = \alpha_v u(v),$$

where we slightly change our notation to designate by $u(v)$ the common value of u at v. For $\alpha_v = 0$, we obviously recover the Kirchhoff-Neumann condition. If δ conditions are requested on each of the vertices of the graph, the quadratic form and its associated domain $H^1_D(\mathcal{G})$ are given by

$$Q(u) = \frac{1}{2}\sum_{e\in\mathcal{E}}\|u_e'\|_{L^2}^2 + \frac{1}{2}\sum_{v\in\mathcal{V}}\alpha_v|u(v)|^2, \quad H^1_D(\mathcal{G}) = \left\{u \in H^1(\mathcal{G}) : \forall v \in \mathcal{V}, \forall e, e' \sim v,\ u_e(v) = u_{e'}(v)\right\}.$$

2.2. **Space discretization.** We present here the space discretization of the second order unbounded operator H. We discretize each edge $e \in \mathcal{E}$ with $N_e \in \mathbb{N}^*$ interior points (when $e \in \mathcal{E}$ is semi-infinite, we choose a large but finite length and we add an artificial terminal vertex with appropriate - typically Dirichlet - boundary condition). We therefore obtain a uniform discretization $\{x_{e,k}\}_{0\leqslant k\leqslant N_e+1}$ of the edge e that can be assimilated to the interval $I_e = [0, l_e]$, *i.e.*

$$x_{e,0} := 0 < x_{e,1} < \cdots < x_{e,N_e} < x_{e,N_e+1} := l_e,$$

with $x_{e,k+1} - x_{e,k} = l_e/(N_e+1) := \delta x_e$ for $0 \leqslant k \leqslant N_e$ (see Figure 2). We denote by v_1 the vertex at $x_{e,0}$, by v_2 the one at x_{e,N_e+1} and, for any $u \in H^1_D(\mathcal{G})$, for all $e \in \mathcal{E}$ and $1 \leqslant k \leqslant N_e$,

$$u_{e,k} := u_e(x_{e,k}),$$

as well as

$$u_{e,v} := \begin{cases} u_e(x_{e,0}) & \text{if } v = v_1, \\ u_e(x_{e,N_e+1}) & \text{if } v = v_2. \end{cases}$$

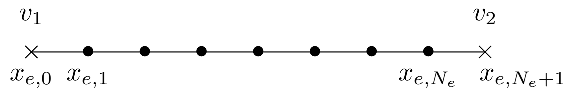


FIGURE 2. Discretization mesh of an edge $e \in \mathcal{E}$

We now assume that $N_e \geqslant 3$ and discretize the Laplacian operator on the interior of e, *i.e.* we give an approximation of $Hu(x_{e,k})$ for $1 \leqslant k \leqslant N_e$. Two cases need to be distinguished: the points closed to the boundary ($k = 1, N_e$) and the other points. We shall start with the later.

Note that we do not discretized the Laplacian on the vertices, because, as will appear in a moment, the values of the functions at the vertices are determined in terms of the values at the interior nodes with the boundary conditions.

For any $2 \leqslant k \leqslant N_e - 1$, the second order approximation of the Laplace operator by finite differences on $e \in \mathcal{E}$ is given by

$$Hu(x_{e,k}) \approx -\frac{u_{e,k-1} - 2u_{e,k} + u_{e,k+1}}{{\delta x_e}^2}.$$

For the cases $k = 1$ and $k = N_e$ corresponding to the neighboring nodes of the vertices v_1 and v_2, the approximation requires u_{e,v_1} and u_{e,v_2}. We therefore use the boundary conditions

$$A_v u(v) + B_v u'(v) = 0, \quad v \in \{v_1, v_2\},$$

in order to evaluate them. To avoid any order reduction, we use second order finite differences to approximate the outgoing derivatives. Therefore, we need the two closest neighboring nodes and for $-2 \leqslant j \leqslant 0$, we denote

$$u_{e,v_1,j} = u_e(x_{e,|j|}) \quad \text{and} \quad u_{e,v_2,j} = u_e(x_{e,N_e+j+1}).$$

The second order approximation of the outgoing derivative from e at $v \in \{v_1, v_2\}$ is given by

$$u_e'(x_{e,v}) \approx (Du_{e,v})_0 := \frac{3u_{e,v,0} - 4u_{e,v,-1} + u_{e,v,-2}}{2\delta x_e}.$$

As a matter of fact, to increase precision, we have chosen in the implementation of the Grafidi library to use third order finite differences approximations for the derivatives at the vertex. This is transparent for the user and we restrict ourselves to second order in this presentation to increase readability. We therefore have the approximation of the boundary conditions

$$A_v[u_{v,0}] + B_v[Du_{v,0}] = 0, \tag{3}$$

where $[u_{v,0}] = (u_{e,v,0})_{e\sim v}$ and $[Du_{v,0}] = ((Du_{e,v})_0)_{e\sim v}$. We define the diagonal matrix $\Lambda \in \mathbb{R}^{d_v \times d_v}$ with diagonal components by

$$\Lambda_{j,j} = \frac{1}{\delta x_{e_j}}, \quad e_j \sim v, \quad j = 1, \dots, d_v.$$

Therefore, the approximate boundary condition (3) can be rewritten as

$$\left(A_v + \frac{3}{2}B_v\Lambda\right)[u_{v,0}] = 2B_v\Lambda[u_{v,-1}] - \frac{1}{2}B_v\Lambda[u_{v,-2}], \tag{4}$$

where $[u_{v,-1}] = (u_{e,v,-1})_{e\sim v}$ and $[u_{v,-2}] = (u_{e,v,-2})_{e\sim v}$. Assuming that $A_v + \frac{3}{2}B_v\Lambda$ is invertible (which can be done without loss of generality, see [17, 19]), this is equivalent to

$$[u_{v,0}] = 2\left(A_v + \frac{3}{2}B_v\Lambda\right)^{-1} B_v\Lambda[u_{v,-1}] - \frac{1}{2}\left(A_v + \frac{3}{2}B_v\Lambda\right)^{-1} B_v\Lambda[u_{v,-2}]. \tag{5}$$

Solving the linear system (4) of size $d_v \times d_v$ allows to compute the boundary values $[u_{v,0}]$ in terms of interior nodes. Thus, the value of u_{e,v_1} (resp. u_{e,v_2}) depends linearly on the vectors $[u_{v_1,-1}]$ and $[u_{v_1,-2}]$ (resp. $[u_{v_2,-1}]$ and $[u_{v_2,-2}]$) which take values from every edge connected to the vertex v_1 (resp. v_2). It is then possible to deduce an approximation of the Laplace operator at $x_{e,1}$ and x_{e,N_e}. Indeed, from (5) there exist $(\alpha_{e,v})_{e\sim v} \in \mathbb{R}^{d_v}$, for $v \in \{v_1, v_2\}$, which depend on every discretization parameter δx_e corresponding to the edges connected to v, such that

$$Hu(x_{e,1}) \approx \frac{u_{e,2} - 2u_{e,1} + \sum\limits_{e\sim v_1} \alpha_{e,v_1}(4u_{e,v_1,-1} - u_{e,v_1,-2})}{\delta x_e{}^2},$$

and

$$Hu(x_{e,N_e}) \approx \frac{u_{e,N_e-1} - 2u_{e,N_e} + \sum\limits_{e\sim v_2} \alpha_{e,v_2}(4u_{e,v_2,-1} - u_{e,v_2,-2})}{\delta x_e{}^2}.$$

Since $(u_{e,v,j})_{-2\leqslant j\leqslant 0, v\in\{v_1,v_2\}}$ are interior mesh points from the other edges, we limit our discretization to the interior mesh points of the graph. The approximated values of u at each vertex will be computed using (5). We denote $[u] = (u_{e,k})_{1\leqslant k\leqslant N_e, e\in\mathcal{E}}$ the vector in $\mathbb{R}^N$, with $N = \sum_{e\in\mathcal{E}} N_e$, representing the values of u at each interior mesh point of each edge of $\mathcal{G}$. We introduce the matrix $[[H]] \in \mathbb{R}^{N\times N}$ corresponding to the discretization of H on the interior of each edge of the graph, which yields the approximation

$$Hu \approx [[H]]\,[u].$$

To define discretized integrals on the graph, we proceed in the following way. We use the standard trapezoidal rule on each of the edges: on an edge I_e, for a vector $[u]$ (corresponding to a discretized function u) we approximate

$$\int_{I_e} u_e(x)dx \approx \mathcal{I}_e([u]) := \delta x_e \left(\sum_{k=0}^{N_e+1} u_{e,k} - \frac{u_{e,0} + u_{e,N_e+1}}{2}\right),$$

where the terminal values $u_{e,0}$, u_{e,N_e+1} are computed with (5). The full integral is then approximated by

$$\int_{\mathcal{G}} u(x)dx \approx \sum_{e\in\mathcal{E}} \mathcal{I}_e([u]).$$

This formula defines directly the discretization of $L^p(\mathcal{G})$, that we denote $\ell^p(\mathcal{G})$.

As an example, we consider the operator H for the graph $\mathcal{G}_{3,sg}$ of Figure 1 with Dirichlet boundary conditions for the exterior vertices A, B and C and Kirchhoff-Neumann conditions for the central vertex O. We plot on Figure 3 the positions of the non zero coefficients of the corresponding matrix $[[H]]$ when the discretization is such that $N_e = 10$, for each $e \in \mathcal{E}$. The coefficients accounting for the Kirchhoff boundary condition are the ones not belonging to the tridiagonal component of the matrix.

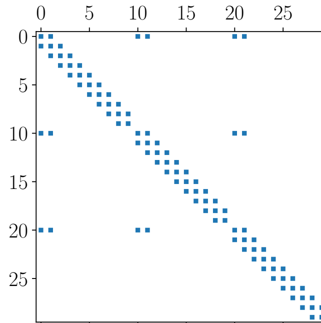


FIGURE 3. Matrix representation $[[H]]$ of H.

3. SOME ELEMENTS OF THE GRAFIDI LIBRARY

3.1. **First steps with the Grafidi library.** We introduce the Grafidi library by presenting some very basic manipulations on an example: we describe the 3-edges star graph $\mathcal{G}_{3,sg}$ drawn on Figure 1 with $\mathcal{V} = \{O, A, B, C\}$ and $\mathcal{E} = \{[OA], [OB], [OC]\}$. We assume that the length of each edge is 10. Our goal in this simple example is to draw a function u that lives on the graph $\mathcal{G}_{3,sg}$, given by

$$u(x) = \begin{cases} e^{-x^2} & \text{for } x \in [OA], \\ e^{-x^2} & \text{for } x \in [OB], \\ e^{-x^2} & \text{for } x \in [OC]. \end{cases} \tag{6}$$

The result is achieved using the code given in Listing 1. We now describe each part of this simple example. The functionalities of the Grafidi library rely on the following Python libraries: networkx, numpy and matplotlib, which we first import. The networkx library is mandatory and should be imported after starting Python. Depending on the desire to make drawings and to make linear algebra operations, it is recommended to import matplotlib and numpy. We then need to import the Grafidi library. It is made of two main classes: Graph and WFGraph, which we choose to import respectively as `GR` and `WF`.

We then begin by creating a variable `g_nx`, which is an instance of the `classes.multidigraph.MultiDiGraph` of the networkx class. This choice is motivated by the need of the description of a directed graph and the possibility of multiple edges connecting the same two nodes. Observe here that we have to choose an arbitrary orientation of the non-oriented graph for numerical purposes. We choose to describe the metric graph in the Python list `g_list`. We identify each vertex by a Python string. Each element of `g_list` corresponds to an edge connecting two vertices. The length of each edge of the graph is defined with the keyword `Length`.

Then, we define the function that we wish to plot through a dictionary where each key corresponds to an edge. The available keys can be found by the Python instruction `g.Edges.keys()`. Each key is a tuple made of three strings. The two first are the vertices labels defining the edge and the third one is an identifier that will be explained later. The values are Python lambda functions with x belonging to the interval $[0, l_e]$, where l_e is the length of the directed edge $e \in \mathcal{E}$. So, $x = 0$ corresponds to the initial vertex of e and $x = l_e$ to the last one. We construct an instance of the WFGraph class with the constructor `WFGraph` with as arguments the dictionary `fun` and the instance of the graph `g`. Since we import the class WFGraph as `WF`, the instruction may be shorten as it appears in the listing. It remains to use the `draw` method of the WFGraph class to plot the

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from Grafidi import Graph as GR
from Grafidi import WFGraph as WF

g_list=["O A {'Length':10}", "O B {'Length':10}", "O C {'Length':10}"]
g_nx = nx.parse_edgelist(g_list,create_using=nx.MultiDiGraph())

g = GR(g_nx)

fun = {}
fun[('O', 'A', '0')]=lambda x: np.exp(-x**2)
fun[('O', 'B', '0')]=lambda x: np.exp(-x**2)
fun[('O', 'C', '0')]=lambda x: np.exp(-x**2)

u = WF(fun,g)
_ = WF.draw(u)
```

LISTING 1. Simple Python example to draw a function on a 3-star graph $\mathcal{G}_{3,sg}$.

function u on $\mathcal{G}_{3,sg}$. The result is available on Figure 4. Since the **draw** function of the WFGraph class delivers outputs, we use the Python instruction `_ =` to avoid their display.

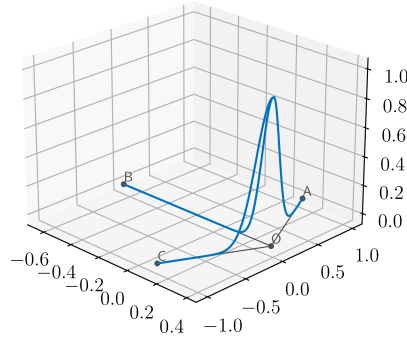


FIGURE 4. Plot of the function u on graph $\mathcal{G}_{3,sg}$

We use the networkx library to determine the geometric positions of each vertex on the plane (Oxy). More specifically, the function `networkx.drawing.layout.kamada_kawai_layout` is executed on `g_nx` within Graph class automatically to compute them. The length of each edge is however not taken into account (indeed, the networkx library is implemented for non metric graphs). To overcome this issue, we have implemented the method `Position` in Graph class. This method allows the user to define by hand the geometric positions of each vertex. Its single argument is one dictionary where the geometric position is given for each vertex. Finally, we draw the graph $\mathcal{G}_{3,sg}$ with the method `draw`. For example, the definition of the geometric positions and the representation of the graph is proposed in Listing 2.

```
NewPos={'O':[0,0],'A':[-10,0],'B':[10,0],'C':[0,10]}
GR.Position(g,NewPos)
_ = GR.draw(g)
```

LISTING 2. Definition of geometric positions of vertices

The new plot of the function u on $\mathcal{G}_{3,sg}$ and the representation of the graph are provided in Figure 5.

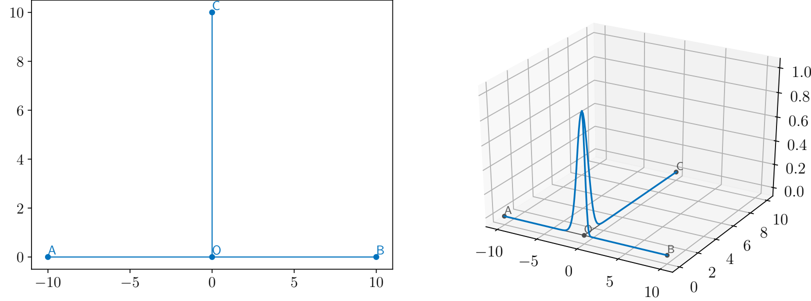


FIGURE 5. Plot of the graph $\mathcal{G}_{3,sg}$ (left) and of the function u on it (right).

3.2. **Basic elements of the Graph class.** The purpose of the Grafidi library is to provide tools to compute numerical solutions of partial differential equations involving the Laplace operator H defined by (1)-(2).Actually, the instruction `g = GR(g_nx)` in Listing 1 automatically creates the discretization matrix $[[H]]$ of the operator H following the rules defined in Section 2. By default, the standard *Kirchhoff-Neumann conditions* are considered at each vertex and $N_e = 100$ nodes are used to discretize each edge $e \in \mathcal{E}$. The total number of discretization nodes is $N = \sum_{e\in\mathcal{E}} N_e$. The matrix is stored in a sparse matrix in Compressed Sparse Column format in `-g.Lap` (actually, `g.Lap` is the approximation matrix of $-H$). If needed, the user may declare other boundary conditions at each vertex. The boundary conditions at each vertex are stored in a Python dictionary, which we call here `bc`. Each key corresponds to a vertex and the values are lists. We provide in the Grafidi library various standard boundary conditions (Kirchhoff-Neumann, Dirichlet, δ, δ'), but more general can be constructed by defining matrices A and B at each vertex as in (2). We consider again the graph $\mathcal{G}_{3,sg}$ and assume that the space discretization has to be made with 3000 interior nodes, and that boundary conditions are of homogeneous Dirichlet type at the vertices A, B and C, and of δ type with strength 1 for the vertex O. We therefore modify the instruction `g = GR(g_nx)` of Listing 1 to construct a new graph taking into account the new boundary conditions and total number of discretization points (see Listing 3). Indeed, the constructor `Graph` actually

```
bc = {'O':['Delta',1], 'A':['Dirichlet'], 'B':['Dirichlet'], 'C':['Dirichlet']}
N=3000
g = GR(g_nx,N,bc)
```

LISTING 3. Definition of boundary conditions at vertices and discretization parameter

takes three arguments: the mandatory instance of the networkx graph `g_nx`, and two optional arguments, the total number of discretization nodes `N` and the dictionary `bc` describing the boundary conditions at each vertex of graph $\mathcal{G}$.

During the creation of the graph `g`, two additional variables are also automatically created: `g.Edges` and `g.Nodes`. They allow to store informations related to the mesh of $\mathcal{G}$. We describe them on the simple two-edges star graph $\mathcal{G}_{2,sg}$ (see Figure 6). It is made of three vertices A, B, C, A being the central node, and two edges $[AB]$ and $[AC]$ with identical length L.

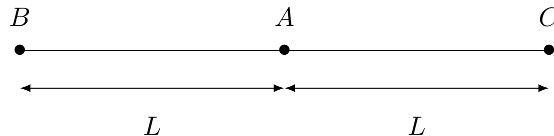


FIGURE 6. Simple two edges star graph $\mathcal{G}_{2,sg}$

We describe the mesh on the graph $\mathcal{G}_{2,sg}$. We assume that $L = 5$ and we discretize the graph with $N = 18$ interior nodes. Thus, $\delta x_i = \delta x = 1/2$ and each edge is discretized with $N_i = 9$, $i = 1, 2$, nodes. The associated mesh is drawn on Figure 7.

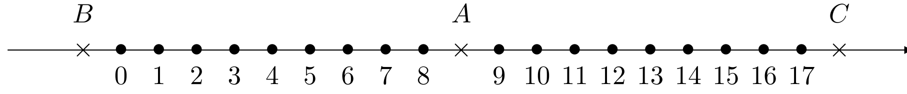


FIGURE 7. Mesh on the simple star graph $\mathcal{G}_{2,sg}$.

The discretization nodes on the edge $[AB]$ are indexed from 0 to 8 and the ones on $[AC]$ are indexed from 9 to 17. All these informations are stored in the dictionary `g.Edges`. The keys are the edges of the graph made of the vertices of each edge and a label (two vertices can be linked by many edges). For the simple two-edges graph, the dictionary is given in Listing 4 (more detailed explanations of the content of the dictionary is provided in the next section).

```
Edges = {
   ('B','A','0') : {'N':9, 'L':5, 'dx':0.5, 'Nodes':['B','A'], 'TypeC':'S', 'Indexes':[0,8]},
   ('A','C','0') : {'N':9, 'L':5, 'dx':0.5, 'Nodes':['A','C'], 'TypeC':'S', 'Indexes':[9,17]}
```

LISTING 4. The dictionary `g.Edges`

The second important variable is the dictionary `g.Nodes` that contains various important informations to build the finite differences approximation of the operator H on $\mathcal{G}$. The keys of `g.Nodes` are the identifiers for each vertex. For the simple 2-star graph, they are `'A'`, `'B'` and `'C'`. We associate to each vertex a dictionary with various keys. We describe below the most relevant keys.

- `'Degree'` is an integer containing the degree d_v of the vertex v.
- `'Boundary conditions'` is a string containing the boundary condition set on the vertex v. The current possibilities are
  - `['Dirichlet']`,
  - `['Kirchhoff']`,
  - `['Delta', val]`, where `val` is the characteristic value of the δ condition,
  - `['Delta Prime', val]`, where `val` is the characteristic value of the δ' condition,
  - `['UserDefined', [A_v,B_v]]`, where `[A_v,B_v]` are matrices used to describe the boundary condition at the vertex v.
- `'Position'` is a list $[x, y]$ representing the geometric coordinates of the vertex v.

We already met the method `draw` of Graph class. Some options are available to control figure name, color, width, markersize, textsize of the drawing of the graph (for a complete description, see Appendix). The method `draw` returns figure and axes matplotlib identifiers. This allows to have a fine control of the figure and its contained elements with matplotlib primitives.

3.3. **A first concrete example: eigenelements of the triple-bridge.** We are now able to handle more complex graphs. Since the Grafidi library relies on the MultiDiGraph - *Directed graphs with self loops and parallel edges* - class of networkx library, we can handle loops and many edges between two single vertices. The declaration of such complex graphs is easy with the Graph class, as we illustrate in the following example.

We want to represent the graph $\mathcal{G}_d$ defined on Figure 8.

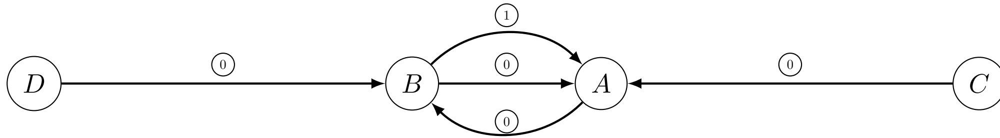


FIGURE 8. Directed graph $\mathcal{G}_d$

The vertices A and B are connected by three edges:

- one edge oriented from the vertex A to the vertex B,
- two edges oriented from the vertex B to the vertex A.

The vertices A and B are also respectively connected to the vertices C and D. We provide in Listing 5 an example of a standard declaration. The library automatically assigns an `Id` (such as the ones indicated in

```
g_list=["B A {'Length':5}", "B A {'Length':10}", "A B {'Length':10}", "C A {'Length':20}", "D B {'Length':20}"]
```

Listing 5. Declaration of complex graph

Figure 8) and a type "segment" `'S'` or "curve" `'C'` to each edge. This operation is transparent for the user. If only one edge connects two vertices, the `Id` is set to `'0'` and the chosen type is `'S'`. On the contrary, the algorithm chooses between `'S'` and `'C'` and the `Id` is incrementally increased starting from `'0'` when multiple edges connect the same two vertices. If a selected edge is of type `'C'`, it will be represented as curved line $\overset{\frown}{AB}$ (actually an half-ellipsis of length `Length`) going from A to B counterclockwise (as a consequence, the edge will be "up" or "down" depending on the position of the vertices, see Figure 9).

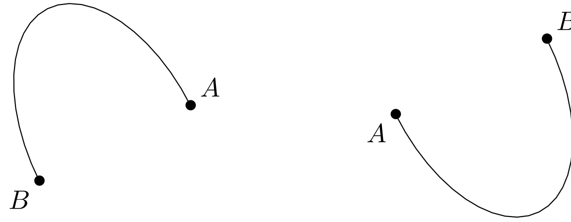


Figure 9. The two configurations "up" and "down" of the oriented curved edge $\overset{\frown}{AB}$

The user can also explicitly provide the edge type and `Id` informations as in Listing 6.

```
g_list=["B A {'Length': 5,'Line':'S','Id':'0'}",\
        "B A {'Length':10,'Line':'C','Id':'1'}",\
        "A B {'Length':10,'Line':'C','Id':'0'}",\
        "C A {'Length':20,'Line':'S','Id':'0'}",\
        "D B {'Length':20,'Line':'S','Id':'0'}"]
```

Listing 6. User defined description of the graph $\mathcal{G}_d$.

The plot with the Grafidi library of the graph $\mathcal{G}_d$ with positions adjusted is presented on Figure 10.

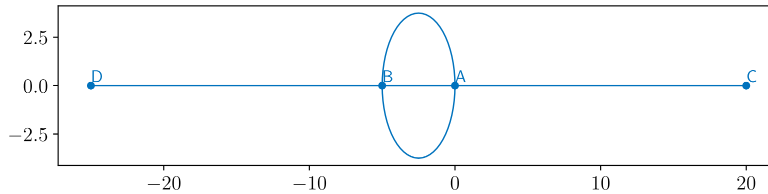


Figure 10. Plot of the graph $\mathcal{G}_d$ with Grafidi library.

As an illustration, we now present the computations of some eigenelements of the operator H on the graph $\mathcal{G}_d$ with Kirchhoff boundary conditions at the vertices A and B and Dirichlet ones at the vertices C and D. Since the approximation matrix of ∂_{xx} is automatically generated and stored in `g.Lap`, we can compute the eigenelements of $[[H]] = -$`g.Lap`. We present in Listing 7 the easiest way to compute the first four eigenvalues/eigenvectors and to draw the eigenvectors on $\mathcal{G}_d$. It is understood that all libraries appearing in Listing 1 are already imported.

Listing 7 works as follows. To compute the eigenelements of $[[H]]$, we use the function `linalg.eigs` of the library `scipy.sparse`. We transform each eigenfunction (stored in the matrix `EigVecs`) as an instance of the WFGraph class by the instruction `EigVec = WF(np.real(EigVecs[:,k]),g)`, where `g` is the graph instance

```
import scipy.sparse as scs

g_list=["B A {'Length':5}", "B A {'Length':10}", "A B {'Length':10}", "C A {'Length':20}", "D B {'Length':20}"]
g_nx = nx.parse_edgelist(g_list,create_using=nx.MultiDiGraph())
g = GR(g_nx)

bc = {'A':['Kirchhoff'], 'B':['Kirchhoff'], 'C':['Dirichlet'], 'D':['Dirichlet']}
N=3000
g = GR(g_nx,N,bc)
NewPos={'A':[0,0],'B':[-5,0],'C':[20,0],'D':[-25,0]}
GR.Position(g,NewPos)

[EigVals, EigVecs] = scs.linalg.eigs(-g.Lap,k=4,sigma=0)
Fig=plt.figure(figsize=[9,6])
for k in range(EigVals.size):
    ax=Fig.add_subplot(2,2,k+1,projection='3d')
    EigVec = WF(np.real(EigVecs[:,k]),g)
    EigVec = EigVec/WF.norm(EigVec,2)
    _=WF.draw(EigVec,AxId=ax)
    ax.set_title(r'$\lambda_{}=$'.format(k)+f'{np.real(EigVals[k]):f}')
```

LISTING 7. Computation of some eigenelements of $[[H]]$ on $\mathcal{G}_d$

of Graph class representing $\mathcal{G}_d$. Next, we normalize the eigenfunction. One notices that the L^2 norm of an instance of WFGraph can be simply computed with the instruction `WF.norm(EigVec,2)`. We are also able to divide a WFGraph entity by a scalar (`EigVec/WF.norm(EigVec,2)`). Each eigenvector is finally plotted with the command `WF.draw(EigVec,AxId=ax)`. The option `AxId` allows to plot the eigenvector on the matplotlib axes `ax`. The fours eigenvectors with their associated eigenvalues λ_j are represented in Figure 11.

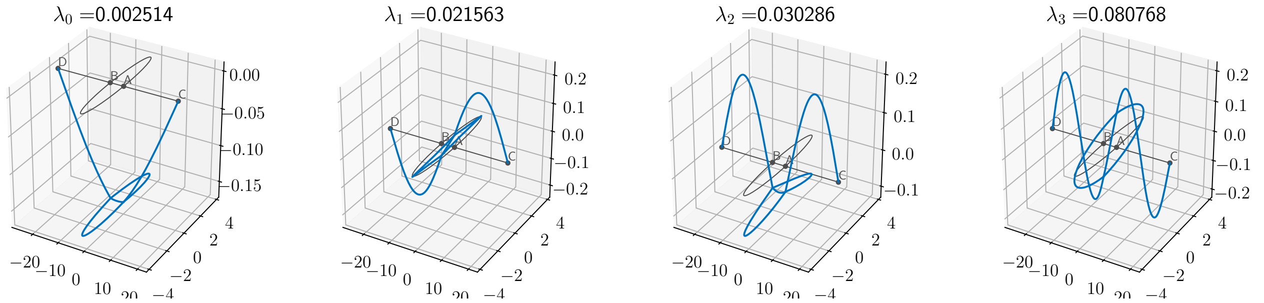


FIGURE 11. The first four eigenvectors of $[[H]]$ on graph $\mathcal{G}_d$.

4. Numerical methods for stationary and time dependent Schrödinger equations

In this section, we discuss the implementation with the Grafidi library of various methods to compute grounds states or dynamical solutions of time-dependent Schrödinger equations on nonlinear quantum graphs.

4.1. **Computation of ground states on quantum graphs.** We begin with the computation of ground states. For a given second order differential operator H on a quantum graph $\mathcal{G}$, a ground state is a minimizer of the Schrödinger energy E at fixed mass M, where

$$E(u) = \frac{1}{2}\langle Hu, u\rangle - \frac{1}{2}\int_{\mathcal{G}} G(|u|^2)dx, \quad G' = g, \quad M(u) = \|u\|^2_{L^2(\mathcal{G})},$$

where g is the nonlinearity. In the following, we consider the case of a power-type nonlinearity

$$g(u) = |u|^{p-1}u, \quad p > 1.$$

To compute ground states, the most common methods are gradient methods. Here, we will cover two popular gradient methods: the Continuous Normalized Gradient Flow (CNGF), which we have analyzed in the context

of quantum graphs in [17], and a nonlinear (preconditionned) conjugate gradient flow (see [12, 22]), which we implement in the particular context of graphs without further theoretical analysis.

4.1.1. *The continuous normalized gradient flow.* We start with the CNGF method. We fix $\delta t > 0$ a certain gradient step and $m > 0$ the mass of the ground state. Let $\rho = \sqrt{m}$ be the L^2-norm of the ground state. The method is divided into two steps: first a semi-implicit gradient descent step then a projection on the constraint manifold (here the L^2-sphere of radius ρ). In practice, we construct a sequence $\{u^n\}_{n\geqslant 0}$ (which will converge to the ground state) given by

$$\begin{cases} u_*^{n+1} = u^n - \delta t \left(H u_*^{n+1} - |u^n|^{p-1} u_*^{n+1} \right), \\ u^{n+1} = \rho u_*^{n+1} / \|u_*^{n+1}\|_{L^2(\mathcal{G})}, \end{cases}$$

where the initial data $u^0 \in L^2(\mathcal{G})$ is chosen such that $\|u^0\|_{L^2(\mathcal{G})} = \rho$. The implementation is described in Algorithm 1, where we have chosen a stopping criterion corresponding to the stagnation of the sequence of vectors $[u^n]$ in the ℓ^2-norm. The gradient step requires to solve a linear system whose matrix is

$$[[M_n]] = [[\mathrm{Id}]] + \delta t \left([[H]] - [[|u^n|^{p-1}]] \right).$$

Here, the matrix $[[|u^n|^{p-1}]]$ is a diagonal matrix constructed from the vector $[|u^n|^{p-1}]$.

Algorithm 1 CNGF algorithm

Require: $[u_0] \in \ell^2(\mathcal{G})$ with $\|[u_0]\|_{\ell^2} = \rho$, $\varepsilon > 0$, `Stop_Crit` = `True`, $n = 0$ and `Iter_max` = 1000
  while `Stop_Crit` **and** $n \leqslant$ `Iter_max` **do**
    Solve $\left([[\mathrm{Id}]] + \delta t[[H]] - \delta t[[|u^n|^{p-1}]]\right)[u_*^{n+1}] = [u^n]$
    $[u^{n+1}] \leftarrow \rho [u_*^{n+1}] / \|[u_*^{n+1}]\|_{\ell^2}$
    `Stop_Crit` $\leftarrow \|[u^{n+1}] - [u^n]\|_{\ell^2} / \|[u^n]\|_{\ell^2} > \varepsilon$
    $n \leftarrow n + 1$
  end while

We now proceed to translate Algorithm 1 (with $p = 3$) into a Python script using the Grafidi library. First, we need to construct a quantum graph. We choose to use the same graph as in Listing 7. Our code can be seen in Listing 8 (we avoid repetition in the listings, and consider that Listing 7 is executed prior to Listing 8).

A few comments are in order. The initial data u^0 is set as a function that is quadratic on the edges connecting A and B, increasing from D to B as $\exp(-0.01(x-20)^2)$ (where 20 is the length of $[DB]$) and increasing from C to A as $\exp(-0.01(x-20)^2)$ (where 20 is the length of $[CA]$). An instance u of WFGraph on `g` which corresponds to u^0 is constructed accordingly. The variable $\rho = 1$ corresponding to the L^2-norm is set and the variable u is normalized by using the function `norm` of WFGraph. A function `E` is defined that corresponds to the energy and we can see that we have used the `Lap` function of WFGraph to apply the operator $[[H]]$ to u as well as the function `dot` to compute the scalar product. We set the variables $\delta t = 10^{-1}$ and $\varepsilon = 10^{-8}$. The part of the matrix $[[M_n]]$ that is independent of n is built in the variable `M_1` which is the sum of $[[\mathrm{Id}]]$ (given by the variable `g.Id` from Graph class) and $-\delta t[[H]]$ (where $-[[H]]$ is given by the variable `g.Lap` from Graph class) and we also note that the matrix is sparse. When entering the loop (with at most 1000 iterations), we make a copy of u, then construct the matrix $[[M_n]]$ by adding the diagonal matrix from the nonlinearity (given through the function `GR.Diag` from Graph). The linear system whose matrix is $[[M_n]]$ and right-hand-side $[[u^n]]$ is solved thanks to the function `Solve` from WFGraph. Then, the variable u is normalized, the evolution of the energy is printed, the stopping criterion is computed through the boolean variable `Stop_Crit` and, finally, we verify if the stopping criterion is attained (in which case we exit the loop and draw u).

In the end, we obtain a ground state depicted in Figure 12 which is computed in 665 iterations.

4.1.2. *The nonlinear conjugate gradient flow.* We now turn to the more sophisticated nonlinear conjugate gradient method. It is an extension of the conjugate gradient method that is used to solve linear systems. Here, we choose to use in the context of quantum graphs the method described for full spaces in [12, Algorithm 2], which uses a preconditionner providing robustness. The method consists in the construction of a sequence $\{u^n\}_{n\geqslant 0}$

```
fun = {}
fun[('D', 'B', '0')]=lambda x: np.exp(-10e-2*(x-20)**2)
fun[('C', 'A', '0')]=lambda x: np.exp(-10e-2*(x-20)**2)
fun[('A', 'B', '0')]=lambda x: 1-(x-10)*x/50
fun[('B', 'A', '0')]=lambda x: 1+(x-5)*x/20
fun[('B', 'A', '1')]=lambda x: 1+(x-10)*x/30

u   = WF(fun,g)
rho = 1
u   = rho*u/WF.norm(u,2)

def E(u):
    return -0.5*WF.Lap(u).dot(u) - 0.25*WF.norm(u,4)**4
En0 = E(u)

delta_t = 10e-1
Epsilon = 10e-8
M_1 = g.Id - delta_t*g.Lap
for n in range(1000):
    u_old = u
    M = M_1 - delta_t*GR.Diag(g,abs(u)**2)
    u = WF.Solve(M,u)
    u = rho*WF.abs(u)/WF.norm(u,2)
    En = E(u)
    print(f"Energy evolution: {En-En0 : 12.8e}",end='\r')
    En0 = En
    Stop_crit = WF.norm(u-u_old,2)/WF.norm(u_old,2)<Epsilon
    if Stop_crit:
        break

_=WF.draw(u)
print()
```

LISTING 8. Computation of a ground state using the CNGF method.

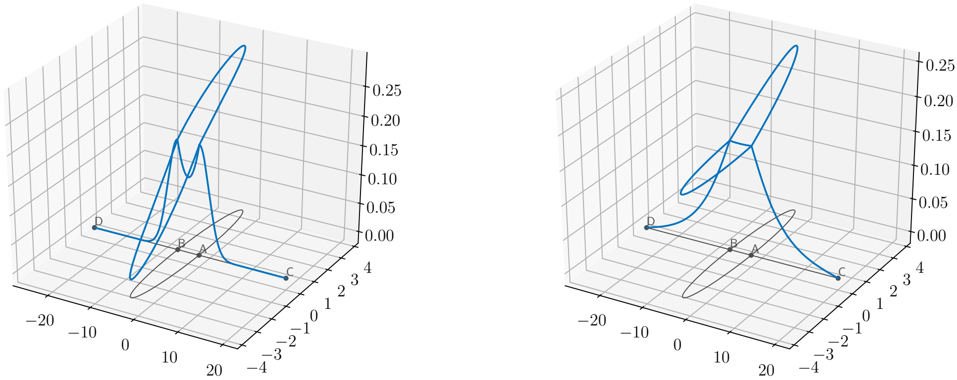


FIGURE 12. Initial data (left) and ground state (right) obtained with the code from Listing 8.

(converging to the ground state), which is recursively defined by

$$\begin{cases} r^n = \mathcal{P}_{T,u^n}(Hu^n - |u^n|^{p-1}u^n) \\ \beta^n = \max\left(0, \langle r^n - r^{n-1}, Pr^n\rangle / \langle r^{n-1}, Pr^{n-1}\rangle\right) \\ d^n = -Pr^n + \beta_n p^{n-1} \\ p^n = \mathcal{P}_{T,u^n} d^n \\ \theta^n = \min_{\theta\in[-\pi,\pi]} E(\cos(\theta)u^n + \sin(\theta)\mathcal{P}_S p^n) \\ u^{n+1} = \cos(\theta^n)u^n + \sin(\theta^n)\mathcal{P}_S p^n \end{cases}$$

where $\mathcal{P}_{T,u}$ is the orthogonal projection on the tangent manifold of the sphere $\mathbb{S}_\rho = \{u \in L^2(\mathcal{G}) : \|u\|_{L^2} = \rho\}$ at u given by

$$\mathcal{P}_{T,u} v = v - \frac{\langle v, u\rangle}{\|u\|_{L^2}^2} u,$$

$\mathcal{P}_S$ is the orthogonal projection on $\mathbb{S}_\rho$ given by

$$\mathcal{P}_S v = \frac{\rho v}{\|v\|_{L^2}},$$

and $P = (1+H)^{-1}$ is the preconditionner. The implementation of the method is described in Algorithm 2, where we have added a preliminary gradient descent step to initialize the iterative procedure.

Algorithm 2 Nonlinear Conjugate Gradient algorithm

Require: $[u^{-1}] \in \ell^2(\mathcal{G})$ with $\|[u^{-1}]\|_{\ell^2} = \rho$, $\varepsilon > 0$, `Stop_Crit` = `True`, $n = 0$ and `Iter_max` $= 500$
  $\lambda^{-1} \leftarrow \langle [[H]][u^{-1}] - [[|u^{-1}|^{p-1}]][u^{-1}], [u^{-1}]\rangle / \|[u^{-1}]\|_{\ell^2}$
  $[r^{-1}] \leftarrow [[H]][u^{-1}] - [[|u^{-1}|^{p-1}]][u^{-1}] - \lambda^{-1}[u^{-1}]$
  Solve $(\alpha[[\mathrm{Id}]] + [[H]])[v^{-1}] = [r^{-1}]$
  $[p^{-1}] \leftarrow [v^{-1}] - \langle [v^{-1}], [u^{-1}]\rangle / \|[u^{-1}]\|_{\ell^2}[u^{-1}]$
  $[\ell^{-1}] \leftarrow \rho[p^{-1}]/\|[p^{-1}]\|_{\ell^2}$
  Minimize $f(\theta^{-1}) = E(\cos(\theta^{-1})[u^{-1}] + \sin(\theta^{-1})[\ell^{-1}]), \quad \theta^{-1} \in [-\pi, \pi]$
  $[u^0] \leftarrow \cos(\theta^{-1})[u^{-1}] + \sin(\theta^{-1})[\ell^{-1}]$
  while `Stop_Crit` **and** $n \leqslant$ `Iter_max` **do**
    $\lambda^n \leftarrow \langle [[H]][u^n] - [[|u^n|^{p-1}]][u^n], [u^n]\rangle / \|[u^n]\|_{\ell^2}$
    $[r^n] \leftarrow [[H]][u^n] - [[|u^n|^{p-1}]][u^n] - \lambda^n[u^n]$
    Solve $(\alpha[[\mathrm{Id}]] + [[H]])[v^n] = [r^n]$
    $\beta^n \leftarrow \max\left(0, \langle [r^n] - [r^{n-1}], [v^n]\rangle / \langle [r^{n-1}], [v^{n-1}]\rangle\right)$
    $[d^n] \leftarrow -[v^n] + \beta^n[p^{n-1}]$
    $[p^n] \leftarrow [d^n] - \langle [d^n], [u^n]\rangle / \|[u^n]\|_{\ell^2}[u^n]$
    $[\ell^n] \leftarrow \rho[p^n]/\|[p^n]\|_{\ell^2}$
    Minimize $f(\theta^n) = E(\cos(\theta^n)[u^n] + \sin(\theta^n)[\ell^n]), \quad \theta^n \in [-\pi, \pi]$
    $[u^{n+1}] \leftarrow \cos(\theta^n)[u^n] + \sin(\theta^n)[\ell^n]$
    `Stop_Crit` $\leftarrow \|[u^{n+1}] - [u^n]\|_{\ell^2}/\|[u^n]\|_{\ell^2} > \varepsilon$
    $n \leftarrow n+1$
  end while

The corresponding code in Python, with the help of the Grafidi library, is given in Listings 9. We use the same quantum graph as in Section 3.1, with in particular a δ condition with parameter 1 at O. The initial function will be the function u defined in (6), normalized to verify the mass constraint. Listings 1-3 are assumed to have been executed prior to Listings 9.

```
import scipy.optimize as sco

rho = 2
Epsilon = 10e-8

u = rho*u/WF.norm(u,2)

def E(u):
    return -0.5*WF.Lap(u).dot(u) - 0.25*WF.norm(u,4)**4
def P_S(u):
    return rho*u/WF.norm(u,2)
def P_T(u,v):
    return v - v.dot(u)/(WF.norm(u,2)**2)*u
def GradE(u):
    return -WF.Lap(u)-WF.abs(u)**2*u
def Pr(u):
    return WF.Solve(0.5*g.Id-g.Lap,u)
def E_proj(theta,u,v):
    return E(np.cos(theta)*u+np.sin(theta)*v)
```

```
def argmin_E(u,v):
    theta = sco.fminbound(E_proj,-np.pi,np.pi,(u,v),xtol = 1e-14,maxfun = 1000)
    return np.cos(theta)*u+np.sin(theta)*v
En = E(u)

rm1 = P_T(u,-GradE(u))
vm1 = Pr(rm1)
pnm1 = P_T(u,Pr(rm1))
lm1 = P_S(pnm1)
u = argmin_E(u,lm1)

for n in range(500):
    r = P_T(u,-GradE(u))
    v = Pr(r)
    beta = max(0,(r-rm1).dot(v)/rm1.dot(vm1))
    rm1 = r
    vm1 = v
    d = -v + beta*pnm1
    p = P_T(u,d)
    pm1 = p
    l = P_S(p)
    um1 = u
    u = argmin_E(u,l)
    En0 = En
    En = E(u)
    print(f"Energy evolution: {En-En0 : 12.8e}",end='\r')
    Stop_crit = WF.norm(u-um1,2)/WF.norm(um1,2)<Epsilon
    if Stop_crit:
        break
_=WF.draw(u)
print()
```

LISTING 9. Computation of a ground state using the nonlinear conjugate gradient method.

As for Listing 8, a few comments are in order. The initial data u^0 is set as a function that is decreasing from O to A, O to B and O to C as $\exp(-x^2)$. A function `P_S` is defined that corresponds to $\mathcal{P}_S$, another one `P_T` corresponds to $\mathcal{P}_{T,u}$ and another one `GradE` corresponds to the gradient of the energy. Furthermore, a function `Pr` computes the application of the preconditionner $[[P]]$ to an instance of WFGraph and returns the result as an instance of WFGraph. The function `E_proj` computes the energy $E(\cos(\theta)u + \sin(\theta)v)$ with variables u, v as instance of WFGraph and θ a scalar. The function `argmin_E` is defined to return $w = \cos(\theta)u + \sin(\theta)v$ where θ is the solution of the minimum of $f(\theta) = E(\cos(\theta)u + \sin(\theta)v)$ and, moreover, it uses the function `fminbound` from Scipy (the maximum of iterations is fixed to 500 and the tolerance to 10^{-8}).

In the end, we obtain a ground state depicted in Figure 13 which is computed in 9 iterations.

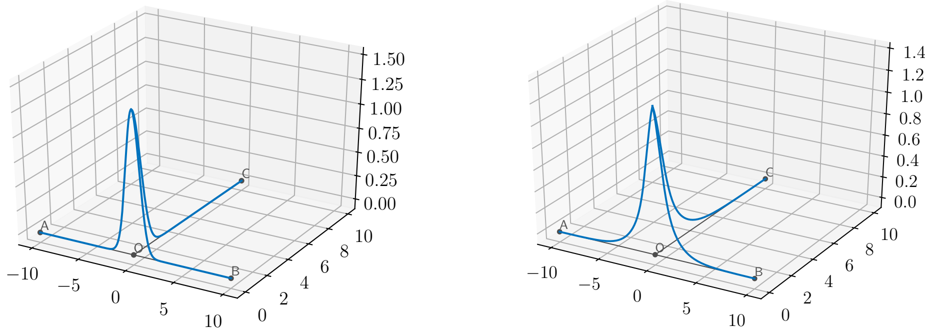


FIGURE 13. Initial data (left) and ground state (right) obtained with the code from Listing 9.
On peut garder la donnée initiale qui est différente de la figure 5 car elle a été renormalisée.

4.2. **Simulation of solutions of time-dependent nonlinear Schrödinger equations on graphs.** In this section, we discuss the dynamical simulations of nonlinear Schrödinger equations on quantum graphs. To be more specific, we wish to simulate the solution on $\mathbb{R}^+ \times \mathcal{G}$ of the following time-dependent equation

$$\begin{cases} i\partial_t \psi = H\psi - |\psi|^2\psi, \\ \psi(t=0) = \psi_0 \in L^2(\mathcal{G}). \end{cases} \tag{7}$$

We have chosen to present our results for the cubic power nonlinearity, but extension to other types of nonlinearity is straightforward.

4.2.1. *The Crank-Nicolson relaxation scheme.* One popular scheme to discretize (7) in time is the Crank-Nicolson scheme [23] which is of second order. Since the equation is nonlinear, the main drawback of the Crank-Nicolson scheme is the need to use a fixed-point method at each time step, which can be quite costly. To avoid this issue, we use the relaxation scheme proposed in [16] which is semi-implicit and of second order. Let $\delta t > 0$ be the time step. The relaxation scheme applied to (7) is given by

$$\begin{cases} \dfrac{\phi^{n+\frac{1}{2}} + \phi^{n-\frac{1}{2}}}{2} = -|\psi^n|^2 \\ i\left(\dfrac{\psi^{n+1} - \psi^n}{\delta t}\right) = H\left(\dfrac{\psi^{n+1} + \psi^n}{2}\right) + \phi^{n+\frac{1}{2}}\left(\dfrac{\psi^{n+1} + \psi^n}{2}\right), \quad \forall n \geqslant 0, \\ \phi^{-\frac{1}{2}} = -|\psi^0|^2 \quad \text{and} \quad \psi^0 = \psi_0 \in L^2(\mathcal{G}), \end{cases} \tag{8}$$

where ψ^n is an approximation of the solution ψ of (7) at time $n\delta t$. By introducing the intermediate variable $\psi^{n+\frac{1}{2}} = (\psi^{n+1} + \psi^n)/2$, we deduce Algorithm 3.

Algorithm 3 Relaxation scheme

Require: $[\psi^0] \in \ell^2(\mathcal{G})$, $\delta t > 0$, $T > 0$ and $N = \lceil T/\delta t \rceil$
  $[\phi^{-\frac{1}{2}}] = -|[\psi^0]|^2$
  for $n = 1, \dots, N$ **do**
    $[\phi^{n+\frac{1}{2}}] = -2|[\psi^n]|^2 - [\phi^{n-\frac{1}{2}}]$
    Solve $\left([[\mathrm{Id}]] + i\delta t/2[[H]] + i\delta t/2[[\phi^{n+\frac{1}{2}}]]\right)[\psi^{n+\frac{1}{2}}] = [\psi^n]$
    $[\psi^{n+1}] \leftarrow 2[\psi^{n+\frac{1}{2}}] - [\psi^n]$
  end for

We now wish to perform a simulation on the tadpole graph depicted in Figure 14 with the following lengths: $|AB| = 6$ and $|BC| = |CB| = \pi$. Observe here that we have to introduce an auxiliary vertex C with Kirchhoff condition, and the loop is constructed as two half-loop edges connecting B and C. This is of no consequence for the behavior of wave functions on the loop, as it was observed in [14, Remark 1.3.3] that a vertex with Kirchhoff conditions with only two incident edges can always be removed.

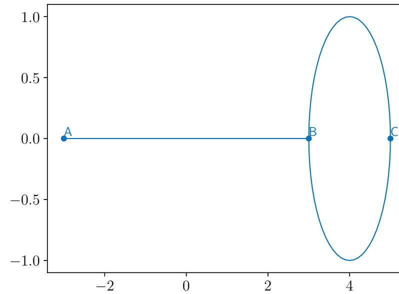


FIGURE 14. Graph for the simulation with the relaxation scheme.

The boundary conditions that we take for the operator H are Dirichlet at A and Kirchhoff at B and C. The initial data is taken as a bright soliton in the middle of the segment $[AB]$ with an initial velocity, *i.e.*

$$\psi^0(x) = \begin{cases} \dfrac{m}{2\sqrt{2}} \operatorname{sech}\left(\dfrac{m(x-3)}{4}\right) e^{icx}, & \text{for } x \in [AB], \\ 0, & \text{for } x \in [BC] \cup [CB], \end{cases}$$

with $m = 20$ and $c = 3$. Our simulation ends at time $T = 1$ with a step time of $\delta t = 10^{-3}$. We observe that the numerical scheme (8) involves complex valued functions ψ^{n+1}, ψ^n and $\phi^{n+1/2}$. We therefore have to explicitly declare `WFGraph` instances with complex type. To this aim, we set `WF(fun,g,Dtype='complex')`. The argument `Dtype` is by default set to `'float'`. The choice `Dtype='complex'` allows to use `numpy.complex128` arrays for linear algebra operations. This leads us to Listing 10 where we implemented the relaxation scheme with the Grafidi library. This listing gives us the opportunity to discuss the outputs of `draw` function of WFGraph and their usage. The return of `draw` is a three components tuple `K,fig,ax`:

- `fig` is the matplotlib figure identifier where the plots are made,
- `ax` is the matplotlib axes included in `fig`,
- `K` collection of elements actually drawn in `ax`.

When we call `draw` with `K` as second argument, it automatically updates the collection of elements in `K` into the figure `fig` without completely redrawing it, which is more efficient. In order to apply this modification, we need to use both `fig.canvas.draw()` and `plt.pause(0.01)`.

```
g_list = ["A B {'Length': 6}", "B C {'Length':3.14159}", "C B {'Length':3.14159}"]
g_nx = nx.parse_edgelist(g_list,create_using=nx.MultiDiGraph())
bc = {'A':['Dirichlet'], 'B':['Kirchhoff'], 'C':['Kirchhoff']}
N=3000
g = GR(g_nx,N,bc)
NewPos={'A':[-3,0],'B':[3,0],'C':[5,0]}
GR.Position(g,NewPos)

m = 20
c = 3
fun = {}
fun[('A', 'B', '0')]=lambda x: m/2/np.sqrt(2)/np.cosh(m*(x-3)/4)*np.exp(1j*c*x)
psi = WF(fun,g,Dtype='complex')

K,fig,ax=WF.draw(WF.abs(psi))

T = 1
delta_t = 1e-3
phi = -WF.abs(psi)**2
M_1 = g.Id - 1j*delta_t/2*g.Lap
for n in range(int(T/delta_t)+1):
    phi = -2*WF.abs(psi)**2 - phi
    M = M_1 + 1j*delta_t/2*GR.Diag(g,phi)
    varphi = WF.Solve(M,psi)
    psi = 2*varphi - psi
    if n%100==0:
        _=WF.draw(WF.abs(psi),K)
        fig.canvas.draw()
        plt.pause(0.01)

_=WF.draw(WF.abs(psi),K)
```

LISTING 10. Simulation of a soliton traveling in a quantum graph with a relaxation scheme.

The result of the simulation can be seen in Figure 15 where the absolute value of ψ at different times is given.

4.2.2. *The Strang splitting scheme.* Another popular approach for the simulation of nonlinear Schrödinger evolution is the so-called splitting method [54]. As is well-known, the idea behind splitting methods is to "split" the full evolution equation into several (simpler) dynamical equations which are solved successively at each time step. In the case of (7), we split the equation into a linear part and a nonlinear part. The equation corresponding to the linear part is

$$i\partial_t \psi = H\psi, \tag{9}$$

and the equation associated to the nonlinear part is

$$i\partial_t \psi = -|\psi|^2\psi.$$

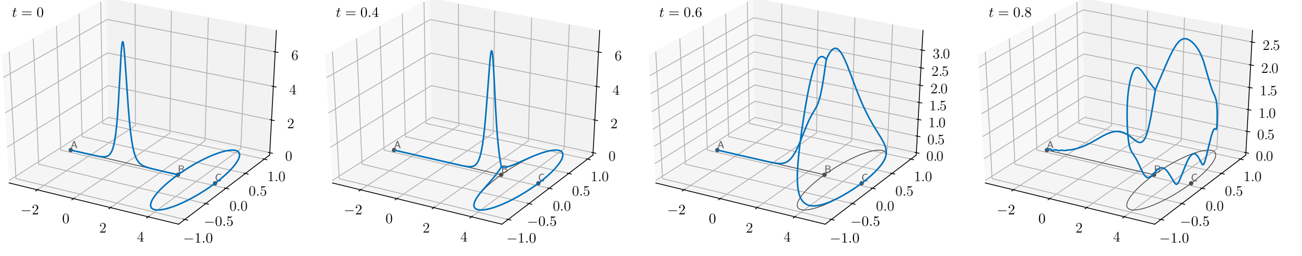


FIGURE 15. Simulation of (7) at times $T = 0, 0.4, 0.6, 0.8$ with the relaxation scheme.

This is motivated by the fact that the solution for the nonlinear part can be obtained explicitly. We use a Strang splitting scheme of second order [51]. For a given time step $\delta t > 0$, we obtain the following method, for any $n \geqslant 0$,

$$\begin{cases} \psi^{n+\frac{1}{3}} = e^{i\delta t/2|\psi^n|^2}\psi^n, \\ \frac{\psi^{n+\frac{2}{3}}-\psi^{n+\frac{1}{3}}}{\delta t} = H\left(\frac{\psi^{n+\frac{2}{3}}+\psi^{n+\frac{1}{3}}}{2}\right), \\ \psi^{n+1} = e^{i\delta t/2|\psi^{n+\frac{2}{3}}|^2}\psi^{n+\frac{2}{3}}, \end{cases}$$

where we have used a Crank-Nicolson scheme to discretize in time Equation (9). Through the introduction of an intermediate variable $\psi^{n+\frac{1}{2}} = (\psi^{n+\frac{2}{3}} + \psi^{n+\frac{1}{3}})/2$, we deduce Algorithm 4.

Algorithm 4 Strang splitting scheme

Require: $[\psi^0] \in \ell^2(\mathcal{G})$, $\delta t > 0$, $T > 0$ and $N = \lceil T/\delta t \rceil$
  for $n = 1, \dots, N$ **do**
    $[\psi^{n+\frac{1}{3}}] \leftarrow \mathtt{exp}(i\delta t/2|[\psi^n]|^2)[\psi^n]$
    Solve $([[\mathrm{Id}]] + i\delta t/2[[H]])\,[\psi^{n+\frac{1}{2}}] = [\psi^{n+\frac{1}{3}}]$
    $[\psi^{n+\frac{2}{3}}] \leftarrow 2[\psi^{n+\frac{1}{2}}] - [\psi^{n+\frac{1}{3}}]$
    $[\psi^{n+1}] \leftarrow \mathtt{exp}(i\delta t/2|[\psi^{n+\frac{2}{3}}]|^2)[\psi^{n+\frac{2}{3}}]$
  end for

We now wish to perform a simulation on the graph depicted in Figure 16 with the following lengths: $|AB| = 6$, $|BC| = |BD| = 10.61$ and $|CE| = |CF| = |DG| = |DH| = 9.96$.

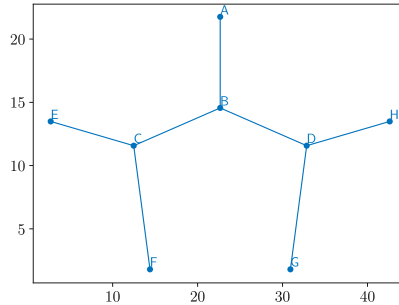


FIGURE 16. Graph of the simulation for the splitting scheme.

The boundary conditions that we would like for the operator H are Kirchhoff at B, C and D and Dirichlet for all the others. The initial data is a bright soliton in the middle of segment $[AB]$ with an initial velocity $c = 3$ and $m = 15$. Our simulation ends at time $T = 2$ with a step time of $\delta t = 10^{-3}$. This leads us to Listing 11 where we implemented the Strang splitting scheme with the Grafidi library.

The result of the simulation can be seen in Figure 17 where the absolute value of ψ is plotted at different times. The ripples are expected to appear, as when reaching a vertex the solution will split between waves going through the vertex and a reflected wave, which will itself interact with the rest of the incident wave.

```
g_list=["A B {'Length':7.20}", "B C {'Length':10.61}", "B D {'Length':10.61}",\
    "C E {'Length':9.96}", "C F {'Length':9.96}", "D G {'Length':9.96}", \
    "D H {'Length':9.96}"]
g_nx = nx.parse_edgelist(g_list,create_using=nx.MultiDiGraph())
bc = {'A':['Dirichlet'],'B':['Kirchhoff'],'C':['Kirchhoff'],\
    'D':['Kirchhoff'],'E':['Dirichlet'],'F':['Dirichlet'],\
    'G':['Dirichlet'],'H':['Dirichlet']}
N=3000
g = GR(g_nx,N,bc)

NewPos = {
    'A': [22.656, 21.756], 'B': [22.656, 14.556], 'C': [12.473, 11.573],\
    'D': [32.838, 11.573], 'E': [2.7, 13.49],     'F': [14.39, 1.8],\
    'G': [30.922, 1.8],    'H': [42.612, 13.49]}
GR.Position(g,NewPos)

m = 15
c = 3
x0 = 7.2/2
fun = {}
fun[('A', 'B', '0')]=lambda x: m/2/np.sqrt(2)/np.cosh(m*(x-x0)/4)*np.exp(1j*c*x)
psi = WF(fun,g,Dtype='complex')

K,fig,ax=WF.draw(WF.abs(psi))

T = 2
delta_t = 1e-3
M = g.Id - 1j*delta_t*g.Lap/2

for n in range(int(T/delta_t)):
    psi = psi*WF.exp(1j*delta_t/2*WF.abs(psi)**2)
    varphi = WF.Solve(M,psi)
    psi = 2*varphi - psi
    psi = psi*WF.exp(1j*delta_t/2*WF.abs(psi)**2)
    if n%100==0:
        _=WF.draw(WF.abs(psi),K)
        fig.canvas.draw()
        plt.pause(0.01)

_=WF.draw(WF.abs(psi),K)
```

LISTING 11. Simulation of a soliton traveling in a tree-shaped quantum graph with a splitting scheme.

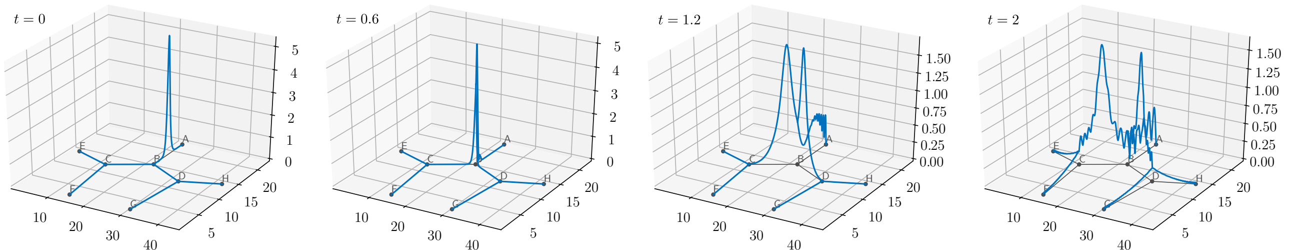


FIGURE 17. Simulation of (7) at times $t = 0, 0.6, 1.2, 2$ with a splitting scheme.

5. GROUND STATES: NUMERICAL EXPERIMENTS AND THEORETICAL VALIDATION

In this section, we present various numerical computations of ground states. In many cases, explicit exact solutions are available. We show the efficiency of the CNGF scheme for all these cases. Even though the CNGF method was built for a general nonlinearity, we focus in this section on the computations of the ground states

of the focusing cubic nonlinear Schrödinger (NLS) equation on a graph $\mathcal{G}$, that reads

$$i\psi_t = H\psi - \lambda|\psi|^{p-1}\psi, \quad x \in \mathcal{G}, \tag{10}$$

with $\lambda > 0$.

Unless otherwise specified, we assume that $\lambda = 1$ and $p = 3$.

In what follows, we discuss only the power case nonlinearity and we focus on the results involving the obtention of ground states by minimization of the energy under a fixed mass constraint. It is in general not an easy task to prove that the standing wave profiles obtained by other techniques (e.g. bifurcation) are (or are not) minimizers at fixed mass (even locally).

Recall that for the classical nonlinear Schrödinger equation, the equation is said to be L^2-subcritical if (in one dimension) $1 < p < 5$. This is also the range of exponents for which standing waves are stables, and for which they can be obtained as minimizers of the energy at fixed mass. Metric graphs being based on one dimensional structures (segments and half-lines), the interesting range of exponents for the nonlinearity is $1 < p \leqslant 5$, with the expectation of additional difficulties in the analysis at the critical case $p = 5$. The global dimension of the graph might induce further restriction on the set of possible exponents, e.g. for the 2-d grid $\mathbb{Z}^2$, which is locally 1-d but globally 2-d (we will comment on that later on).

For the cubic nonlinear Schrödinger equation on a finite (bounded or unbounded) graph, and at sufficiently large mass, Berkolaiko, Marzuola and Pelinovsky [15] established the following results. For any edge of the graph, there exists a bound state located on the graph, i.e. it is positive, achieves its maximum on the edge, and the mass of the bound state is concentrated on the edge up to an exponentially small error (see [15, Theorem 1.1] for a precise statement). Moreover, comparing the energies of these bound states, the authors are able to find the one with the smallest energy at fixed mass. Note, however, that the bound state with the smallest energy has not been proven yet to be the ground state. Heuristic arguments in favor of this hypothesis are given in [15, Section 4.4]. The results of [15] have to be put in perspective with the results established by Adami, Serra and Tilli [9] for generic sub-critical power type nonlinearities. Indeed, by very elegant purely variational techniques, Adami, Serra and Tilli [9] established for non-compact graphs the existence of positive bound states achieving their maximum on any chosen finite edge. These bound states are obtained by purely variational techniques: it is proved that they are global minimizers of the energy among the class of functions with fixed mass, *and the additional constraint that the functions should achieve their maximum on the given edge*. It turns out unexpectedly that the minimizer so obtained lies in fact in the interior of the constraint, hence it may also be characterized as a *local* (but obviously not necessarily global) minimizer of the energy at fixed mass. In the same vein, the existence of local minimizers of the energy for fixed mass has also been established by Pierotti, Soave and Verzini [48] in cases where no ground state exists. As the estimates [15, (4.6) and (4.7)] indicate, a pendant edge is clearly preferable to a non-pendant one. However, for non-pendant edges, the differences between energies are quite small.

From the preceding discussion, we infer that extra-care is required when performing numerical experiments, as the outcome of the algorithm may very be only a local minimizer and not a global one.

We have divided this section into four parts, depending on the kind of graphs considered: compact graphs, graphs with finitely many edges, one of which is semi-infinite, periodic graphs and,finally, trees. If the vertices conditions are not specified, it means that Kirchhoff conditions are assumed.

5.1. **Compact graphs.** Compact graphs are made of a finite number of edges, all of which are of finite length. On compact graphs, the existence of minimizers in the subcritical case $1 < p < 5$ is granted by Gagliardo-Nirenberg inequality and the compactness of the injection of $H^1(\mathcal{G})$ into $L^p(\mathcal{G})$ for $1 \leqslant p \leqslant \infty$. Hence the main question becomes to identify (or, in the absence of suitable candidates, to describe) the minimizer. Several works have been recently devoted to general compact graphs : [15, 20, 24, 26, 38]. For the simplest compact graphs like the line segment or the ring, the minimizer is (usually) known and this offers us good test cases for our algorithm. Results applying to general compact graphs are not always easy to test numerically (e.g. in [24], Dovetta proved for any compact graph, for any $1 < p < 5$ and for any mass the existence of a sequence of bound state whose energy goes to infinity, but capturing this sequence at the numerical level would require the development of new specific tools). However, it was established in [20] that constant solutions on compact graphs are the ground state (for sub-critical nonlinearities) for sufficiently small mass, a feature which is easy to observe numerically.

The simplest of compact graphs are the segment (two vertices connected by an edge) and the ring (one vertex and an edge connecting the vertex to himself). As the ring case was considered in detail from a variational point of view in [31], we chose to conduct experiments in this case and compare the numerical outcomes with the

theoretical results of [31]. Beside the elementary cases of the segment and the ring, many compact graphs are of interest. We will present some experiments performed in the case of the dumbbell graph, for which several recent solid theoretical studies exist (see e.g. [30, 40]).

5.1.1. *The ring.* From a numerical point of view, we obtain a ring (i.e. a one loop graph) by gluing together two half circles with Kirchhoff conditions at the vertices (as already explained in Section 4.2, it is innocuous for the functions on the graphs). Considering the loop graph with an edge of length T is equivalent to work on the line $\mathbb{R}$ with T-periodic functions, i.e to work in the functional setting:

$$H^1_{\text{loc}} \cap P_T, \quad P_T = \{f \in L^2_{\text{loc}}(\mathbb{R}) : \; f(x+T) = f(x), \; \forall x \in \mathbb{R}\}.$$

Minimizers in $H^1_{\text{loc}} \cap P_T$ of the Schrödinger energy

$$E_{\text{ring}}(\psi) = \frac{1}{2}\int_0^T |\psi'(x)|^2 \, dx - \frac{1}{4}\int_0^T |\psi(x)|^4 \, dx \tag{11}$$

at fixed mass m were described explicitly in [31] in terms of Jacobi elliptic functions. Recall that the function dn is the Jacobi elliptic function defined by

$$\text{dn}(x;k) = \sqrt{1 - k^2 \sin^2(\phi)}, \quad k \in (0,1), \tag{12}$$

where ϕ is defined through the inverse of the incomplete elliptic integral of the first kind

$$x = F(\phi, k) = \int_0^\phi \frac{d\theta}{\sqrt{1 - k^2 \sin^2(\theta)}}.$$

The snoidal and cnoidal functions are given by

$$\text{sn}(x;k) = \sin(\phi), \qquad \text{cn}(x;k) = \cos(\phi). \tag{13}$$

Recall also that the complete elliptic integrals of first and second kind are given by $K(k) = F(\pi/2;k)$ and $E(k) = E(\pi/2;k)$, where

$$E(\phi;k) = \int_0^\phi \sqrt{1 - k^2 \sin^2(\theta)} \, d\theta.$$

The solutions of the minimization problem (11) are given as follows.

(1) For all $0 < m < \frac{2\pi^2}{\lambda T}$, the unique minimizer (up to a phase shift) is the constant function

$$\psi_{\text{ring}} = \sqrt{\frac{m}{T}}.$$

(2) For all $\frac{2\pi^2}{\lambda T} < m < \infty$, the unique minimizer (up to phase shift and translation) is the rescaled dnoidal function

$$\text{dn}_{\alpha,\beta,k} = \frac{1}{\alpha}\,\text{dn}\left(\frac{\cdot}{\beta}; k\right)$$

where the parameters α, β and k are uniquely determined.

(3) If $\lambda = 2$, given $k \in (0,1)$, $T = 2K(k)$, and $m = 2E(k)$, the unique minimizer (up to phase shift and translation) is

$$\text{dn} = \text{dn}(\cdot, k).$$

We place ourselves in the case of item (3) and compute the ground state on the one loop graph with perimeter 2π and $\lambda = 2$. The parameter k is therefore such that $k^2 = 0.9691073732421548$ and we fix the mass to $2E(k)$. We discretize each half circle with $N_e = 1000$ grid nodes. The gradient step is $\delta t = 10^{-2}$. Our experiment gives a remarkable agreement between the theoretical minimizer and the numerically computed minimizer, as shown in Figures 18 and 19.

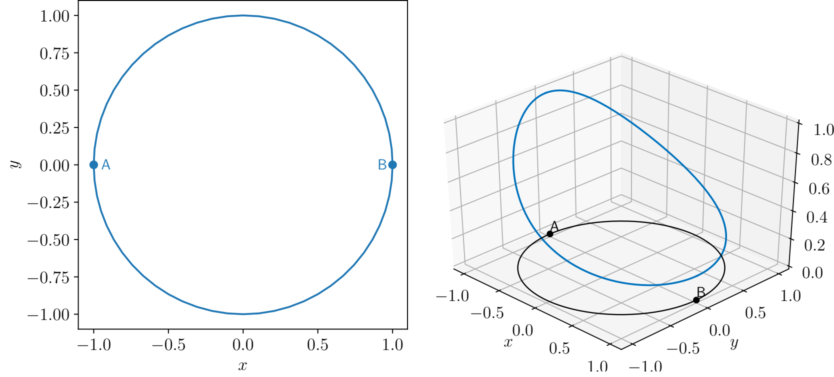


FIGURE 18. Ring of radius 1 (left) and the numerical ground state (right) when $\lambda = 2$.

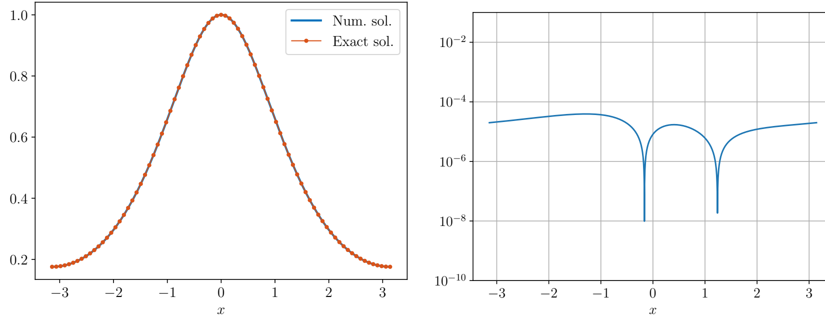


FIGURE 19. Comparison between exact and numerical solution for the ring (where the $\log 10$ of the difference is depicted on the right).

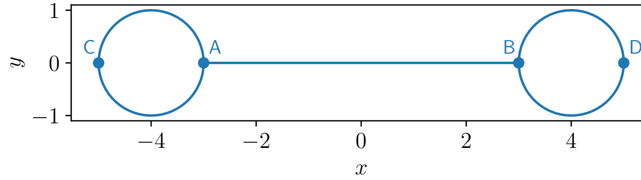


FIGURE 20. The dumbbell graph

5.1.2. *The dumbbell.* The dumbbell graph is a structure made of two rings attached to a central line segment subject to Kirchhoff conditions at the junctions (see Figure 20). Each ring can be assimilated as a loop as in the previous section and is therefore numerically made of two glued half circles. The central line segment has a length $2L$ and the perimeter of each loop is 2π. We set $\lambda = 2$, $L = 3$ and consider the the minimizers of the energy

$$E_{\text{dumbbell}}(\psi) = \frac{1}{2}\int_{\mathcal{G}} |\psi'(x)|^2 - \frac{1}{2}|\psi(x)|^4\,dx$$

with fixed mass M_{dumbbell}. According to [40], there exist m^* and m^{**} (explicitly known) such that $0 < m^* < m^{**} < \infty$ and the following behavior for standing wave profiles on the dumbbell graph holds. For $0 < M_{\text{dumbbell}} < m^*$, the ground state is given by the constant solution $\psi(x) = p$, where p is implicitly given by

$$M_{\text{dumbbell}} = 2(L + 2\pi)p^2.$$

This constant solution undertakes a symmetry breaking bifurcation at m^* and a symmetry preserving bifurcation at m^{**}, which result in the appearance of new positive non-constant solutions. The asymmetric standing wave is a ground state for $M_{\text{dumbbell}} \gtrsim m^*$, and the symmetric standing wave is not a ground state for $M_{\text{dumbbell}} \gtrsim m^{**}$. In our case, the values for m^* and m^{**} are

$$m^* = 0.18646428284896863, \qquad m^{**} = 1.2334076715778846.$$

Observe that the three profiles described above are expected to be local minimizers of the energy at fixed mass, hence we should be able to find them with our numerical algorithm, provided the initial data is suitably chosen. We have found that the three following initial data were leading to the various desired behaviors (in the following, ν is a normalization constant adjusted in such a way that the mass constraint is verified):

- the constant initial data : $\psi_1 \equiv \nu$,
- a gaussian centered on the left loop : $\psi_2(x)_{|CA} = \nu e^{-10x^2}$ and 0 elsewhere,
- a gaussian centered at $x = 2$ on the central edge : $\psi_3(x)_{|AB} = \nu e^{-10(x-2)^2}$ and 0 elsewhere.

We will run the normalized gradient flow for each of these initial data for the three following masses:

$$0 < m_1 = 0.10 < m^*, \quad m^* < m_2 = 0.75 < m^{**}, \quad m^{**} < m_3 = 1.50.$$

The parameters of the algorithm are set as follows. The total number of discretization nodes is $N = 1000$ and $\delta t = 10^{-2}$. The stopping criterion is set to 10^{-8}, and the maximal number of iteration is set to 50000 (which is large enough so that it is never reached in our experiences). The results are in perfect agreement with the theoretical results, as shown in Figure 21. In particular, one can see that for large mass $m_3 > m^{**}$, it is indeed possible to recover the three bound states described theoretically, and comparison of the energies shows that the asymmetric bound state centered on a loop is indeed the ground state. For the smaller mass m_2, the algorithm selects the constant or the asymmetric state, and comparison of the energy shows that the later is indeed the ground state. And for m_1, the algorithm converges in each case towards the constant function. Very small differences in the final energies (after the eighth digit in the m_1 case) may be noted, which are due to our stopping criterion set at 10^{-8}.

Compacity Compactnessfor graphs may be violated in several ways: with a semi-infinite edge, or with an infinite number of edges (which may be arranged e.g. periodically or in tree form). We discuss these cases in the next sections.

5.2. **Graphs with a semi-infinite edge.** In this section, we consider graphs having a finite number of edges, one of which is of semi-infinite length. A typical example for this kind of graph is the N-star graph, consisting of a vertex to which N semi-infinite edges are attached. We will discuss this example in Section 5.2.2. Before that, we will recall in Section 5.2.1 some of the results obtained by Adami and co. concerning a topological obstruction leading to non existence of ground states on nonlinear quantum graphs. Another example, the tadpole graph, will be discussed in Section 5.2.3.

5.2.1. *The topological obstruction.* The existence of ground states with prescribed mass for the focusing nonlinear Schrödinger equation (10) on non-compact finite graphs $\mathcal{G}$ equipped with Kirchhoff conditions at the vertices is linked to the topology of the graph. Actually, a topological hypothesis (H) can prevent a graph from having ground states for every value of the mass (see [8] for a review). For the sake of clarity, we recall that a *trail* in a graph is a path made of adjacent edges, in which every edge is run through exactly once. In a trail vertices can be run through more than once. The assumption (H) has many formulations (again, see [8]) but we give here only one.

Assumption 5.1 (Assumption (H))**.** Every $x \in \mathcal{G}$ lies in a trail that contains two half-lines.

If a finite non-compact graph with Kirchhoff conditions at the vertices verifies Assumption 5.1, then no ground state exists, unless the graph is isomorphic to a tower of bubbles (see Figure 22). Examples of graphs verifying Assumption 5.1 abound, some are drawn on Figure 22. Fortunately, graphs not satisfying Assumption 5.1 and for which ground states exist also abound, some are shown on Figure 23.

5.2.2. *Star graphs.* Star-graphs provide typical examples for nonlinear quantum graphs, as they are non-trivial graphs but retain many features of the well-studied half-line. As star-graphs with Kirchhoff condition at the vertex verify Assumption 5.1 and therefore do not possess a ground state, one usually studies star graphs with other vertex conditions such as δ or δ' conditions.

In this section, we are interested in the computation of ground state solutions for a general N-edges star-graph $\mathcal{G}$ with a central vertex denoted by A with a δ vertex condition at A. Each edge will be numbered with a label $i = 1, \dots, N$ (see Figure 24) and will be identified when necessary with the right half-line $[0, \infty)$. The unknown ψ is the collection of the functions ψ_i living on every edge: $\psi = (\psi_1, \cdots, \psi_N)^T$. The total mass is defined by $M_N(\psi) = \sum_{i=1}^N \int_{\mathbb{R}^+} |\psi_i(x_i)|^2 \, dx_i$.

(a) $m = 0.10$, $u_0 = \psi_1$, $E = -2.6930411461.10^{-4}$

(b) $m = 0.10$, $u_0 = \psi_2$, $E = -2.6930411103.10^{-4}$

(c) $m = 0.10$, $u_0 = \psi_3$, $E = -2.6930411193.10^{-4}$

(d) $m = 0.75$, $u_0 = \psi_1$, $E = -1.5148356447.10^{-2}$

(e) $m = 0.75$, $u_0 = \psi_2$, $E = -2.7205037742.10^{-2}$

(f) $m = 0.75$, $u_0 = \psi_3$, $E = -2.7205037743.10^{-2}$

(g) $m = 1.50$, $u_0 = \psi_1$, $E = -6.0593425789.10^{-2}$

(h) $m = 1.50$, $u_0 = \psi_2$, $E = -1.5097807829.10^{-1}$

(i) $m = 1.50$, $u_0 = \psi_3$, $E = -1.2925753851.10^{-1}$

FIGURE 21. Outcomes of the CNGF Algorithm 1 on the dumbbell graph for three remarkable values of the mass (one for each row) and three possible initial data (one for each column). In each subcaption, the quantity E given is the final energy.

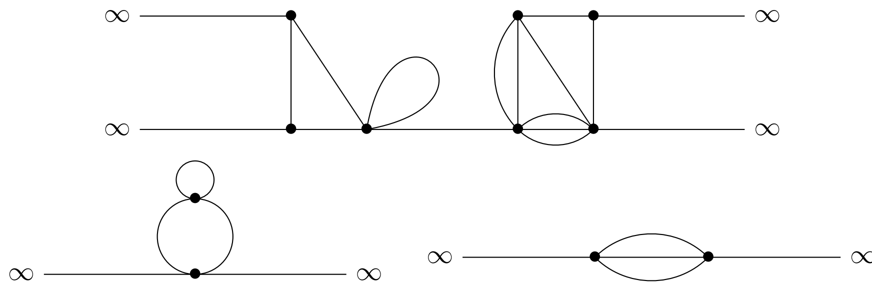


FIGURE 22. Graphs satisfying Assumption (H) : a generic graph (top), a tower of bubble (bottom left), a triple bridge (bottom right)

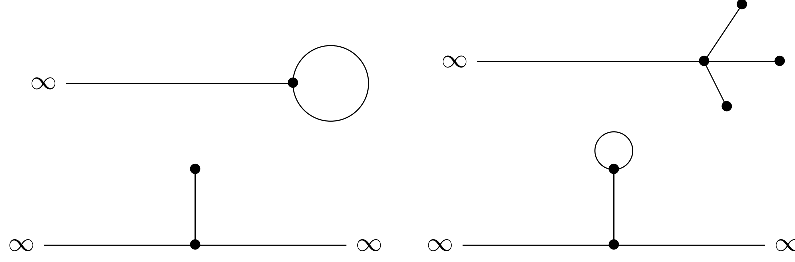


FIGURE 23. Graphs not satisfying Assumption (H) : a tadpole (top left), a 3-fork (top right), a line with a pendant (bottom left), a sign-post (bottom right)

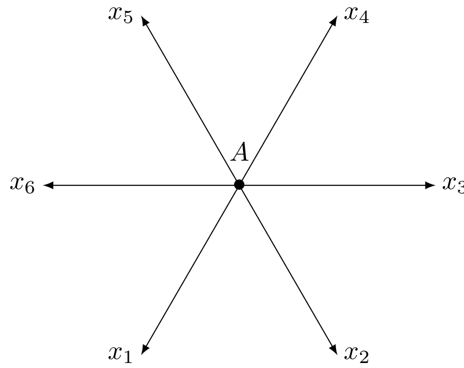


FIGURE 24. Star-graph with $N = 6$ edges

The δ boundary conditions at A are the generalization for $N > 2$ of the δ potential on the line (i.e. the 2-star graph, see e.g. [36, 39] for studies of ground in this case):

$$\psi_j(0) = \psi_k(0) =: \psi(0), \quad 1 \leqslant j, k \leqslant N, \qquad \sum_{j=1}^{N} \psi_j'(0) = \alpha\psi(0).$$

Ground states exist only for attractive δ potential, therefore we assume that

$$\alpha < 0.$$

We set $\lambda = 1$. The energy is given by

$$E_{N,\delta}(\psi) = \sum_{i=1}^{N} \left[\frac{1}{2} \int_{\mathbb{R}^+} |\psi_i'(x_i)|^2 \, dx_i - \frac{1}{4} \int_{\mathbb{R}^+} |\psi_i(x_i)|^4 \, dx_i \right] + \frac{\alpha}{2} |\psi_1(0)|^2.$$

Let $m^* = 4|\alpha|/N$. It was proved in [2] that there exists a ground state minimizing $E_{N,\delta}$ when $M_N = m$ if $m < m^*$ (there is no constraint if $N = 2$). The ground state is explicitly given in [1] and [2] as follows. Let ω be implicitly given by

$$m = 2N\sqrt{\omega} - 2\alpha.$$

Let $\bar{x}$ be defined by

$$\bar{x} = \frac{1}{\sqrt{\omega}} \text{arctanh} \left(\frac{|\alpha|}{N\sqrt{\omega}} \right).$$

Then, the energy reaches its minimum when $\psi = \psi_{\delta,\omega}$ (up to a phase factor) where each component of $\psi_{\delta,\omega}$ is given by

$$\psi_{\delta,\omega,i}(x_i) = \frac{\sqrt{2\omega}}{\cosh(\sqrt{\omega}(x_i + \bar{x}))}, \quad 1 \leqslant i \leqslant N,$$

with $\omega \in (\alpha^2/N^2, +\infty)$. The mass of $\psi_{\delta,\omega}$ is indeed

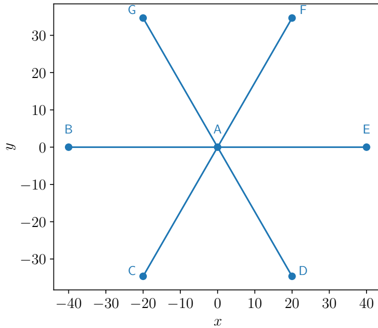
$$M_N(\psi_{\delta,\omega}) = 2N\sqrt{\omega} - 2\alpha = m,$$

and its energy is given by

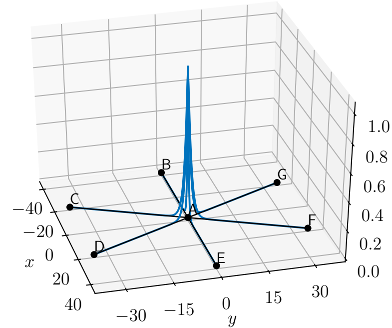
$$E_{N,\delta}(\psi_{\delta,\omega}) = -\frac{N}{3}\omega^{3/2} - \frac{\alpha^3}{3N^2}.$$

In order to compute numerically the ground state, each edge of the approximated graph (see Figure 25 (a)) is of length 40 and discretized with $N_e = 800$ nodes. We add homogeneous Dirichlet boundary conditions at the terminal end of each edge. The gradient step is $\delta t = 10^{-2}$ and we perform 3000 iterations. Each component of the initial data ψ_0 is a Gaussian $\psi_{0,i} = \rho_i e^{-10x_i^2}$ and ρ_i is computed in such a way that the mass of ψ_0 is m. We set $\alpha = -4$ and $\omega = 1$. The outcome is plotted on Figure 25 (b).

We plot on Figure 26 the comparison between the exact solution and the numerical one on an edge (left) and the modulus of the difference in log scale (right), thereby showing the very good agreement of our numerical computations with the theory.



(a) The approximated star-graph



(b) The numerical ground state when $\alpha = -4$ and $\omega = 1$

FIGURE 25. Star graph with 6 edges and δ-condition

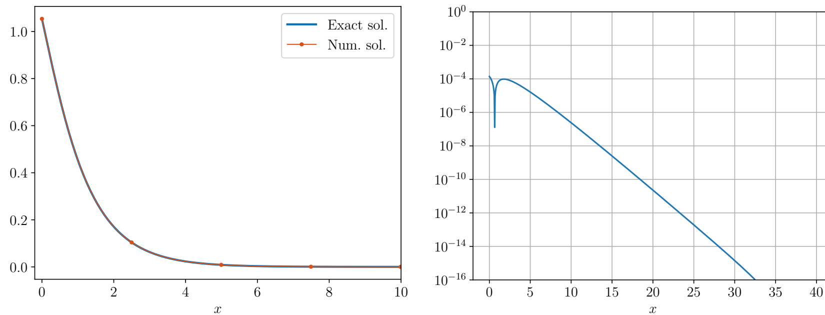


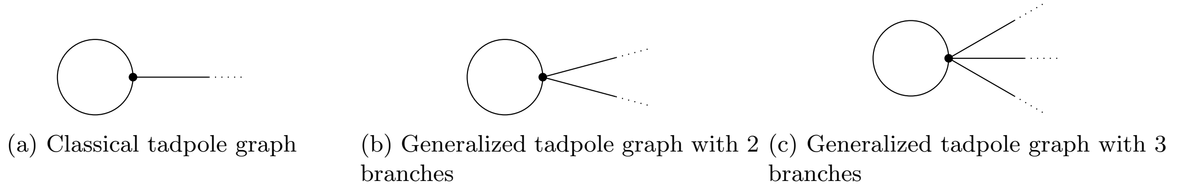
FIGURE 26. Comparison between exact and numerical solutions for δ-condition on a star-graph with 6 edges (where the log 10 of the difference is depicted on the right).

5.2.3. *The tadpole.* The classical tadpole graph consists of one loop with a half line attached to it and was considered in the subcritical case $1 < p < 5$ in [7, 21, 44]. The existence of a ground state for any given mass was established in [7, p 214], and the loop-centered bound state is the good candidate for the ground state. A classification of standing waves was performed in the cubic case $p = 3$ by Cacciapuoti, Finco and Noja [21], and was later extended to the whole subcritical range $1 < p < 5$ by Noja, Pelinovsky, and Shaikhova [44], with some orbital stability results.

The generalized tadpole graph consists of one loop with K half-lines attached at the same vertex (see e.g. Figure 27) and was treated in [15]. When $K = 2$, it is a particular case of the tower of bubbles on the line, with one bubble, and the ground state is known to be the soliton of the real line, folded on the bubble (see [6, Example 2.4]). For $K \geqslant 3$, there is no ground state (as Assumption 5.1 is verified).

Noja-Pelinovski [45] recently analyzed in details the standing waves on the tadpole graph for the critical quintic nonlinearity, with an alternative variational technique (minimization of the H^1 norm at fixed L^6-norm).

In particular, they established the existence of a branch of standing waves for which three regimes exist, depending of the frequency ω of the wave. There exist $0 < \omega_0 < \omega_1$ such that standing waves are ground states if $0 < \omega < \omega_0$, local minimizers of the energy at fixed mass if $\omega_0 < \omega < \omega_1$, and saddle points for the energy at fixed mass if $\omega > \omega_1$.



(a) Classical tadpole graph (b) Generalized tadpole graph with 2 branches (c) Generalized tadpole graph with 3 branches

FIGURE 27. Examples of tadpole graphs

In this section, we present the computation of the ground state to the NLS equation (10) with $\lambda = 1$ on a classical tadpole graph. The graph $\mathcal{G}$ is made of a ring of perimeter $2L$ and a semi-infinite line (tail) originated from a vertex with Kirchhoff condition. It is conjectured in [21] that the ground state exists and is made of a dnoidal-type function on the ring and a sech-type function on the tail. Its explicit formula on the ring is

$$\psi_{\text{ring}}(x) = \sqrt{\frac{2\omega}{2-k_*^2}}\,\text{dn}\left(\sqrt{\frac{\omega}{2-k_*^2}}x; k^*\right), \quad 0 < k_* < 1,$$

where dn is given by (12) and $k_* \in (0,1)$ is the solution of

$$\frac{3k^4}{1-k^2}\text{cn}^2\left(\frac{L\sqrt{\omega}}{\sqrt{2-k^2}}; k\right)\left[1 - \text{cn}^2\left(\frac{L\sqrt{\omega}}{\sqrt{2-k^2}}; k\right)\right] = 1,$$

where cn denotes the cnoidal function defined in (13). The solution on the tail is

$$\psi_{\text{tail}}(x) = \frac{\sqrt{2\omega}}{\cosh(\sqrt{\omega}(x-b))},$$

where b is determined by the negative solution of

$$\frac{1}{\cosh^2(\sqrt{\omega}b)} = \frac{\psi_{\text{ring}}^2(L)}{2\omega}.$$

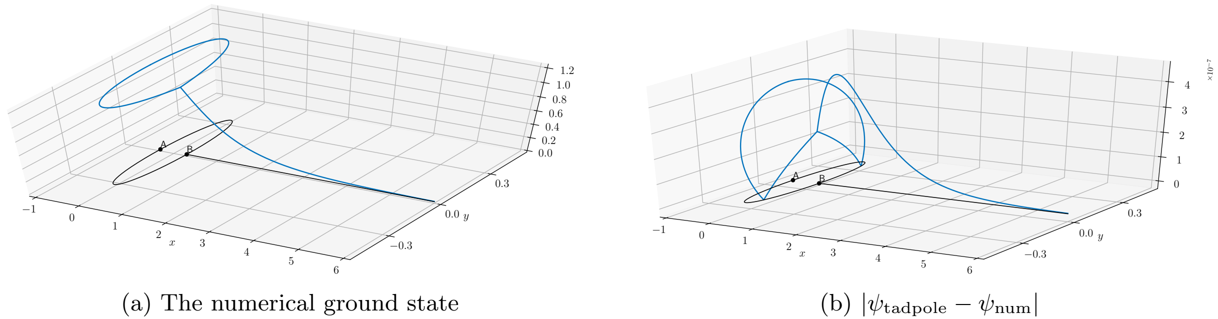
We take a ring of radius $1/\pi$, so that $L = 1$, and we approximate the tail by a segment of length 30. We add homogeneous Dirichlet boundary conditions at the terminal vertex. We take $\omega = 1$. With these quantities, the couple (k_*, b) is given by

$$k_* = .81664827149276692790, \quad b = .89507479534736339894.$$

The mass of the ground state $\psi_{\text{tadpole}} = (\psi_{\text{ring}}, \psi_{\text{tail}})$ is

$$M(\psi_{\text{tadpole}}) = 3.1727382562292.$$

The numerical solution is plotted in Figure 28 (a). We also plot the difference in absolute value between ψ_{tadpole} and the numerical solution on Figure 28 (b). The maximum value of the error is $4.45 \cdot 10^{-7}$.



(a) The numerical ground state (b) $|\psi_{\text{tadpole}} - \psi_{\text{num}}|$

FIGURE 28. On the tadpole graph

5.3. **Periodic graphs.** Periodic graphs are graphs with an infinite number of (usually finite length) edges, for which an elementary structure, the periodicity cell, is repeated in one or more directions.

In the case of 1-d periodic graphs (i.e. graphs for which the periodicity cell is copied in only one direction), Dovetta [25] proved that the situation is similar to the one of the real line: for $1 < p < 5$, there exists a ground state for every mass. The critical case $p = 5$ is a bit more complicated. On one hand, if the graph satisfies the equivalent of the topological Assumption (H) adapted to the periodic setting (Assumption (H_{per})), Dovetta [25] proved the non-existence of ground states. On the other hand, for graphs violating this topological assumption (see for example Figure 29), there may exist a whole interval of mass for which a ground state exists.

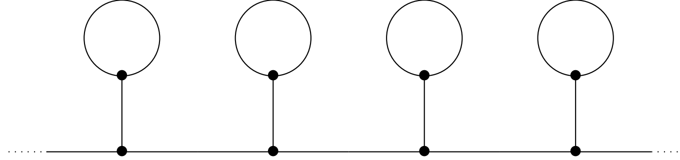


FIGURE 29. A periodic graph not satisfying Assumption (H_{per})

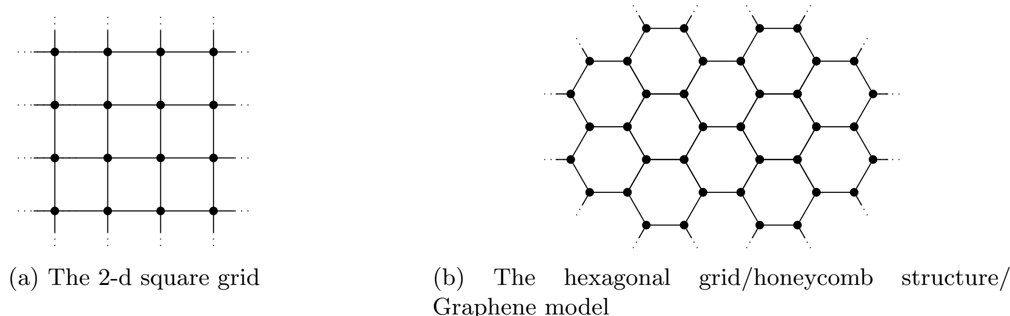
In a somewhat different framework (including in particular periodic potentials in the problem), Pankov [46] proved, under a spectral assumption on the underlying quantum graph, the existence of localized and periodic standing wave profile solutions. These profiles are obtained by minimizing the action (which in our case corresponds to $E + \omega M$ for a fixed ω) on the corresponding Nehari manifold, but, as usual, it is unclear how and in which case these profiles could also be minimizers of the energy at fixed L^2 norm (recall that in the case of the real line minimizers are obtained on the Nehari manifold for any $1 < p < \infty$, whereas on the mass constraint they exist only if $1 < p < 5$).

That graphs periodic along only one direction essentially mimic the behavior of the real line is somewhat expected. However, if the periodicity occur in more than one direction, a new dimensionality of the problem may appear (which was also absent for non-compact graphs with a finite number of edges). At the microscopic level, periodic graphs remain clearly 1-d structures. But at the macroscopic level, periodic graphs may be seen as higher dimensional structures, for examples the 2-d grid (see Figure 30 (a)) or the honeycomb hexagonal grid (see Figure 30 (b)) are clearly 2-d structures at the macroscopic level. This dimensional transition is reflected in the range of critical exponents and masses. Non-compact graphs with a finite number of edges share the same critical exponent (from a nonlinear Schrödinger point of view) as the line, i.e. the graphs are subcritical for power nonlinearities with exponents $1 < p < 5$, and minimization of the energy under a fixed mass constraint is possible only if $1 < p \leqslant 5$. On the other hand, it was revealed in [4, 5] that a *dimensional crossover* with a continuum of critical exponents occurs for the 2-d grid and the hexagonal grid. More precisely, the following has been established in [4, 5]. If $1 < p < 3$, then there exists a ground state for any possible value of the mass. If $3 \leqslant p < 5$, then there exists a critical value m_c of the mass such that ground states exist if and only if $m \geqslant m_c$ (unless $p = 3$, in which case the case $m = m_c$ is open). If $p \geqslant 5$, then a ground state never exists, no matter the value of the mass. Recall that 3 (resp. 5) is the critical exponent for the nonlinear Schrödinger equation on $\mathbb{R}^2$ (resp. on $\mathbb{R}$). Similar results have been obtained for the 3-d grid by Adami and Dovetta [3].

In what follows, we present some numerical experiments realized in two model cases: the necklace and the hexagonal grid.

5.3.1. *The necklace.* The necklace graph is a periodic graph consisting of a series of loops alternating with single edges (see Figure 31) and is probably one of the simplest non-trivial periodic graphs. The validity of the NLS approximation for periodic quantum graphs of necklace type was established by Gilg, Pelinovsky and Schneider [28]. Moreover, Pelinovsky and Schneider [47] showed the existence, at fixed sufficiently small frequency ω, of two symmetric positive exponentially decaying bound states, one located at the center of the single edge and the other equally distributed with respect to the centers of each half-loop. It is conjectured in [47] that the state located on the single edge should be the ground state at small mass. On the other hand, for large masses, it was experimentally observed in [15] that their estimates on edge localized bound states could also be applied in the case of the necklace graph. The conclusion of this observation is that at large mass the ground state should be centered on the loop if the length of the internal edge is smaller that the length of the half-loop, and vice versa.

We have performed numerical calculations of the ground states on a necklace consisting of loops of total length π (i.e. each branch of the loop is of length $\pi/2$) and connecting edges of length 1. The length of the necklace is



(a) The 2-d square grid

(b) The hexagonal grid/honeycomb structure/ Graphene model

FIGURE 30. Doubly periodic metric graphs

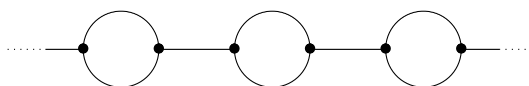
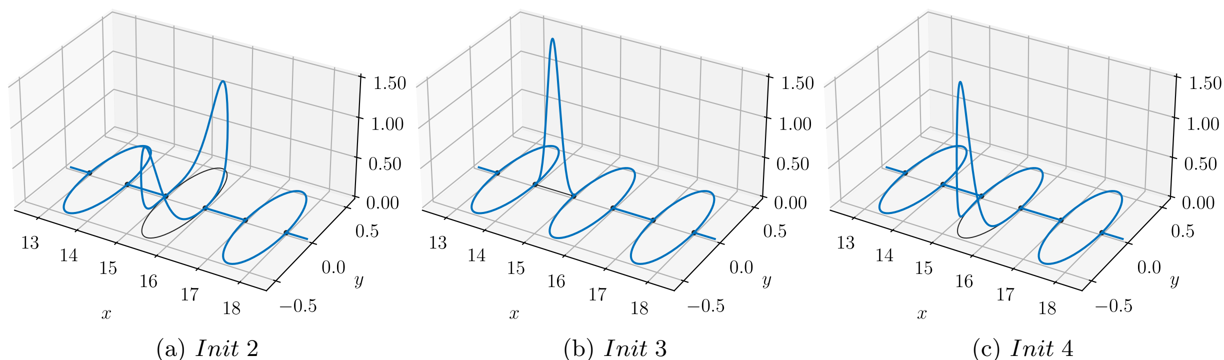


FIGURE 31. The necklace graph, a periodic graph with alternating loop and single edge

chosen to be large, but obviously necessarily finite. In practice, the length needs to be adapted depending on the mass m on which we are minimizing the Schrödinger energy. Indeed, it is expected (and appears to be so in practice) that the ground state will be decaying as $e^{-m|x|}$ from some central point on the graph (here, $|x|$ is referring to the (graph) distance with respect to this point). Therefore, the smaller the mass is, the larger the length of the necklace needs to be in order to fully capture the tail of the ground state. The conditions at the vertices are Kirchhoff conditions, apart from the end points where we have chosen to set Dirichlet conditions.

We have chosen to perform a collection of experiments for masses varying from very small to very large and with three different types of initial data, all positioned on the periodicity cell at the middle of the necklace: two gaussians concentred and centered on each of the branches of the loop (referred to as *Init* 2, see Figure 32 (a)), a gaussian concentred and centered on the single connecting edge (referred to as *Init* 3, see Figure 32 (b)), and a gaussian concentred and centered on a branch of the circle (referred to as *Init* 4, see Figure 32 (c)).



(a) *Init* 2 (b) *Init* 3 (c) *Init* 4

FIGURE 32. The three initial data constructed with Gaussians of the form Ce^{-10x^2}, centered on edges, truncated at the end points, and with C adjusted to satisfy the mass constraint.

We first present a global picture (see Figure 33) of the ground states for L^2 norms ranging from $\rho = 0.1$ to $\rho = 15$ (recall that $\rho = \sqrt{m}$). Since we expected the energy to be of order m^3, we have presented the mass-energy with $\rho^6 = m^3$ log-scale on the horizontal axis. Our expectation is confirmed by the representation which is indeed a straight line, with a slight shift around $\rho = 2.5$ corresponding to a bifurcation (on which we will comment after). We observe that for small masses, the ground state is scattered across many periodicity cells. As the mass increases, the ground state becomes more and more concentrated on a loop, first symmetrically on both branches of the loop, then on only one branch of the loop.

Figure 33 was devoted to the ground state. In fact, we may perform a more detailed analysis and obtain other branches of local minimizers of the energy at fixed mass. Indeed, provided the parameters of our algorithm are

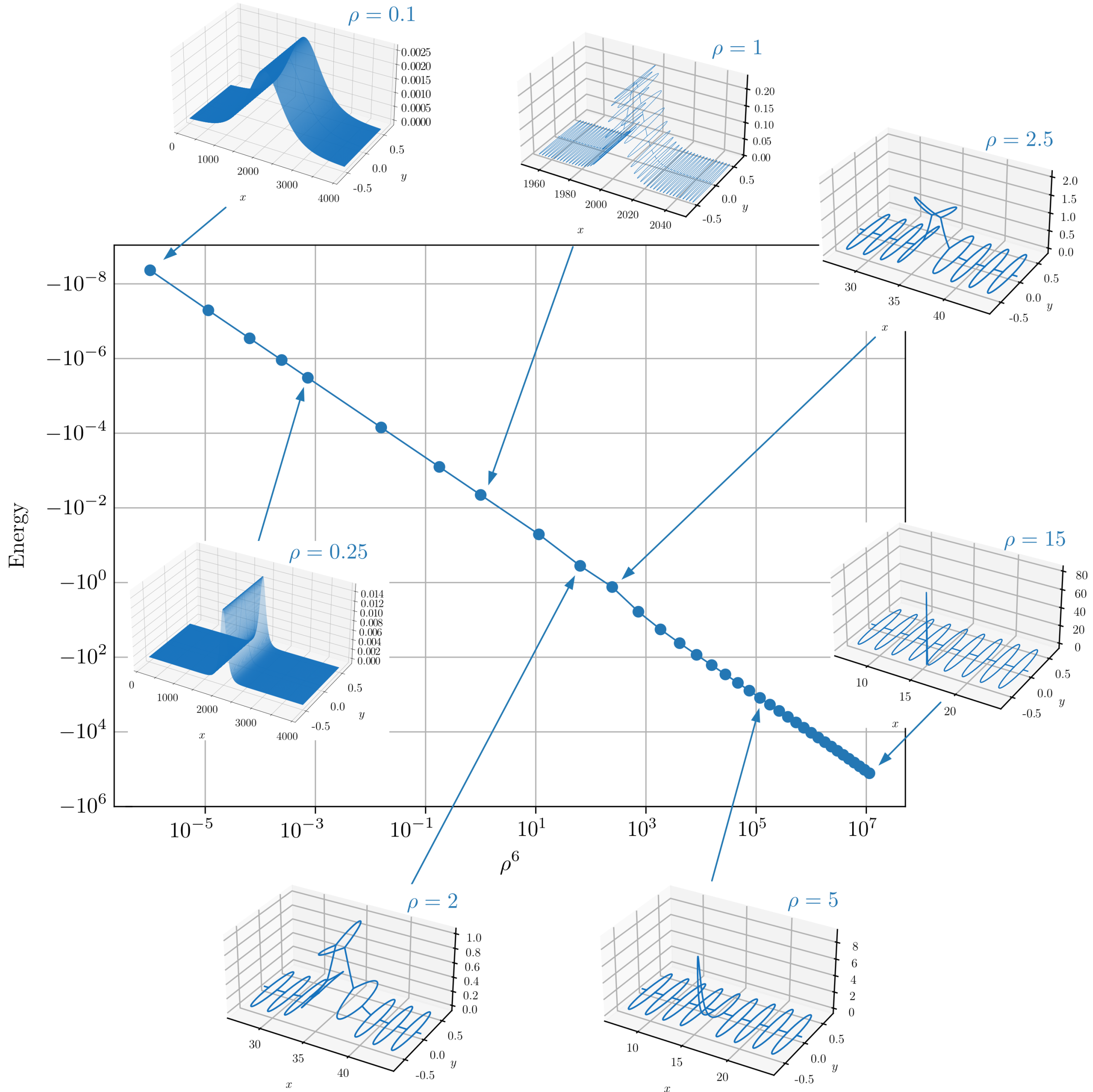


FIGURE 33. Mass-Energy diagram with sample representations for the ground states on a necklace graph

suitably chosen, starting from each of the initial data *Init* j, $j = 1, 2, 3$, we should have convergence towards the closest local minimizer of the energy at fixed mass. The obtained minimizer should itself enjoy similar features as the initial data (e.g. the place of centering). We present the outcome of our simulations in Figure 34. Each initial data gives rise to a branch of local minimizers. For small mass, the branches corresponding to *Init* 2 and *Init* 4 coincide and correspond to the ground state, which is centered on a loop and symmetric with respect to both sides of the loop. At $\rho \simeq 2.5$, we observe a bifurcation and the branches corresponding to *Init* 2 and *Init* 4 separate, as the *Init* 4 branch bifurcates with smaller energy and is formed of ground states peaked on one side of a loop, whereas the *Init* 4 branch continues the branch of symmetric states on a loop (which are not anymore ground states). The *Init* 3 branch is formed all along of states centered on a single edge. It is never

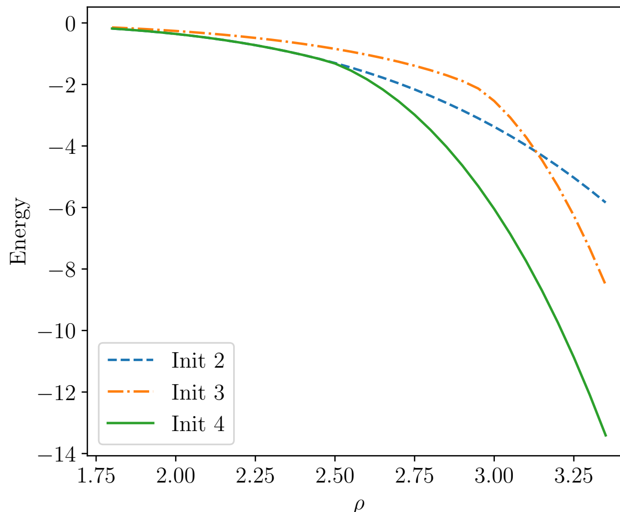


FIGURE 34. Mass-Energy diagram for local minimizers computed with *Init* 2, *Init* 2, and *Init* 4

a ground state branch, but it is meeting the *Init* 4 branch at small and large mass, up to a point where they become indistinguishable numerically (for large mass, outside of Figure 34, at $\rho \simeq 10$).

5.3.2. *The honeycomb.* We now turn to the honeycomb grid. This is a graph which is built recursively using a hexagonal tessellation. As the necklace graph, it is a very simple periodic graph and we can see that it is two-dimensional on a large scale. In [4], the existence of minimizers for the NLS energy functional is proved for $1 < p < 3$, for any mass. Here, we perform some numerical simulations in the case $p = 2$. To be more specific, we use the gradient methods to compute the ground state of the NLS energy functional under a specified mass. As noted in [4, 5], for low masses, we expect the ground state to display a 2d structure due to the spreading on the graph. For large masses, on the contrary, the ground state should be more localized on the graph and we expect a 1d structure. The goal of this numerical investigation is to describe the transition from the 2d regime to the 1d regime by varying the mass of the ground state from 1 to 16.

The graph is set such that each edge has a length of 1. We have obtained the Mass-Energy diagram which is depicted in Figure 35. To begin with, we note that there is a linear relation between the energy and ρ^4. We can see that, for low masses, the ground state looks like a 2d ground state in the Euclidean case. Furthermore, we remark that it is centered on a node (that is, its maximum is located on a node) and symmetric. As the mass grows larger, the ground state is more concentrated. Then, between a mass of 11.4 and 11.5, we observe a structural transition: the minimizer becomes centered on an edge (still symmetric). For larger masses, it keeps concentrating (slowly) on a single edge and, thus, it displays a 1d regime.

5.4. **Metric Trees.** Metric trees are tree-type graphs endowed with a metric structure. In this section, we are interested in the case of binary trees, i.e. trees for which each vertex (except for the root, if any) has degree 3 and all the edges share the same length. Dispersion of the Schrödinger group on trees (with δ conditions at the vertices) was investigated by Banica and Ignat [13]. Existence of ground states on metric trees (with Kirchhoff conditions at the vertices) has been considered by Dovetta, Serra and Tilli [27] in the case of binary trees, either rooted or non-rooted. Let $\mathcal{G}$ be a rooted or non-rooted binary tree with Kirchhoff vertices conditions and (following the notation of [27]), define the minimum of the Schrödinger energy at fixed mass m by

$$\mathcal{L}_{\mathcal{G}}(m) = \min\{E(u) : u \in H^1_D(\mathcal{G}),\ M(u) = m\}.$$

It was proved in [27] that there exists a critical mass $m^*_{\mathcal{G}} \geqslant 0$ such that

$$\begin{cases} \mathcal{L}_{\mathcal{G}}(m) = \frac{1}{2}\lambda_1 m, & \text{and there is no ground state,} & \text{if } \mu \in (0, m^*_{\mathcal{G}}), \\ \mathcal{L}_{\mathcal{G}}(m) < \frac{1}{2}\lambda_1 m, & \text{and a ground state exists,} & \text{if } m > m^*_{\mathcal{G}}, \end{cases}$$

where λ_1 is the optimal constant for the Poincaré inequality on the graph. The nonlinearity considered in [27] is any mass-subcritical power nonlinearity, i.e. $|u|^{p-1}u$ with $1 < p < 5$. If $3 < p < 5$ or if $1 < p < 5$ and minimization is done in the class of radially symmetric functions, the authors of [27] proved that $m^*_{\mathcal{G}} > 0$. The

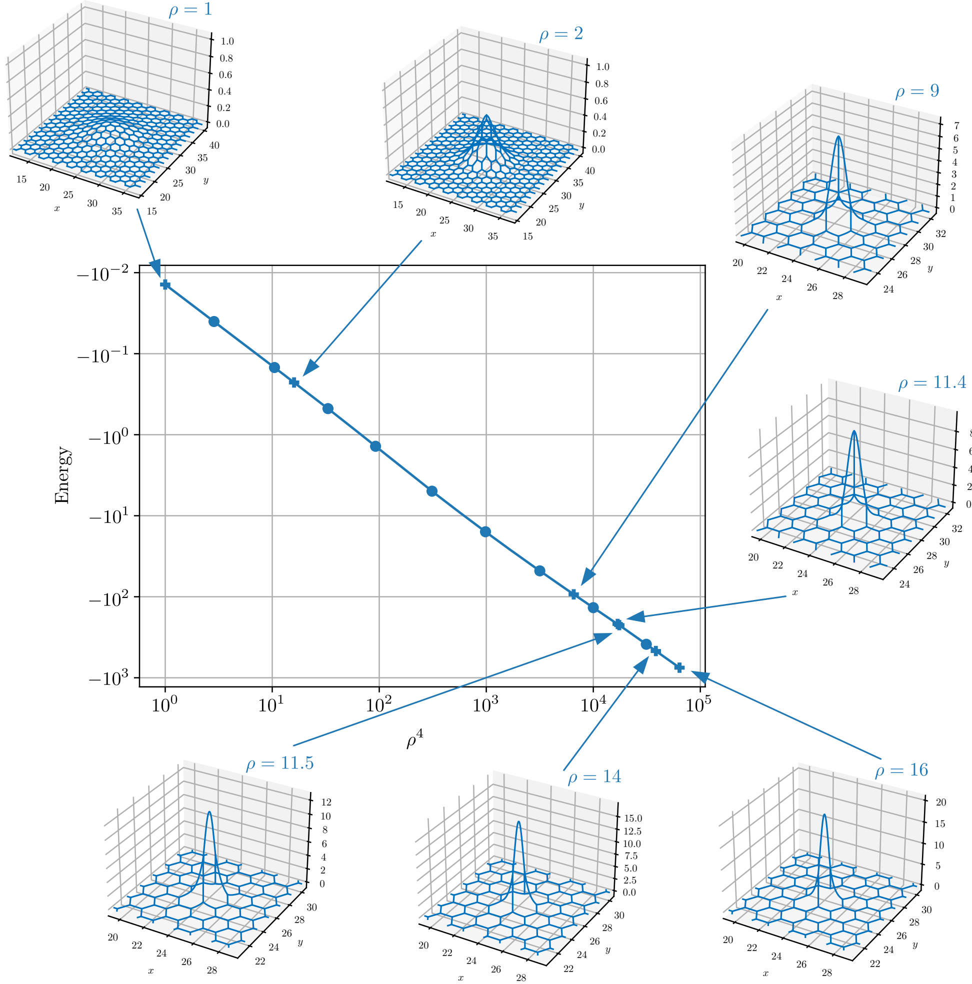


FIGURE 35. Mass-Energy diagram for the minimizers of NLS energy functional on a honeycomb grid.

case $1 < p < 3$ is open if no symmetry assumption is made, but the authors conjecture that $m^*_{\mathcal{G}}$ is also positive in this case. Moreover, they conjecture that minimizers should be radial even when no symmetry assumption is made on the class of function in which minimization is done. This is confirmed by experiments that we conducted on a binary tree of depth 6 with each branch of approximate length 10 (we have arranged the vertices in such a way that they are on concentric circles). We give a sample result of our experiments in Figure 36.

APPENDIX A. FEATURES OF THE GRAFIDI LIBRARY

The Grafidi library relies on the following Python libraries: Matplotlib [35], Networkx [32], Numpy [33], Scipy [53].

A.1. Methods from the Graph class.

A.1.1. *__init__.* The Graph class constructor builds an instance of a graph which is based on the graphs from the NetworkX library (Network Analysis in Python).

g = Graph(g_nx,Np,user_bc)

Parameters:

g_nx: an instance of a NetworkX graph that must have for each edge at least the attribute `'Length'` with value a positive scalar.

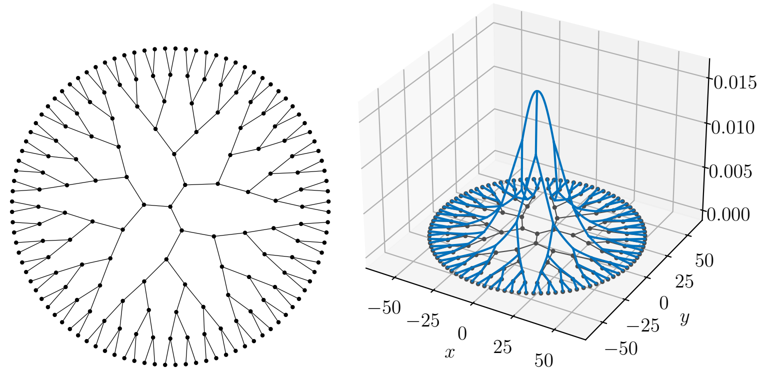


FIGURE 36. Metric tree (left) and ground state with mass constraint $\rho = 0.1$ (right) for $|u|u$ nonlinearity.

Np: (optional) an integer corresponding to the total number of discretization points on the graph. By default, the number of discretization points is set to 100 on each edge.

user_bc: (optional) a dictionary whose keys are the identifiers of vertices used to describe edges in **g_nx** and whose values must be of the form: `['Dirichlet']` for Dirichlet boundary condition, `['Kirchhoff']` for Kirchhoff-Neumann boundary condition, `['Delta',val]` for a δ boundary condition with a strength equal to `val` (which must be a scalar), `['Delta_Prime',val]` for a δ' boundary condition with a strength equal to `val` (which must be a scalar) or `['UserDefined',[A_v,B_v]]` for a user-defined boundary condition with matrices A_v and B_v which must be 2-dimensional `numpy.array` instances. For a full description of all boundary conditions see [17] Not the best ref ?. By default, the boundary conditions for all vertices are Kirchhoff-Neumann boundary conditions.

Return:

g: an instance of the Graph class which contains the finite-differences discretization of the Laplace operator as well as the identity matrix corresponding to the identity operator on the graph. We can access these matrices with `g.Lap` and `g.Id` which are 2-dimensional `scipy.sparse` instances. The sparse format of these instances is `csc` (Compressed Sparse Columns).

A.1.2. *Position.* A method that enables the user to set the position (on the x, y-plane) of every vertex on the graph. This is only useful when drawing a graph or a wave-function on the graph.

Position(g,dict_nodes)

Parameters:

g: an instance of the Graph class whose edges' position will be set.

dict_nodes: a dictionary whose keys are the identifiers of the nodes used in **g** and whose values must be of the form `[posx,posy]` where `posx` and `posy` must be scalars corresponding to the desired x and y coordinates associated to the key node.

A.1.3. *draw.* A method to plot the graph in the Matplotlib figure named `'QGraph'`. Each vertex is represented as a dot and its associated label is displayed.

draw(g,AxId,Color,Text,TextSize,LineWidth,MarkerSize,FigName)

Parameters:

g: an instance of the Graph class.

AxId: (optional) an `Axes` instance of the Matplotlib library. Allow to draw the graph **g** in an already existing axes.

Color: (optional) by default, the color of the graph is blue. It allows to specify an alternative color. The user must follow the standard naming color of Matplotlib library.

Text: (optional) Logical variable. This option allows to control the display of the vertices labels. By default, `Text=True`. To avoid the display of labels, set `Text=False`.

TextSize: (optional) a float variable. This allows to control the text size to display vertices labels. By default, the text size parameter is set to 12.

LineWidth: (optional) a float variable. This allows to control the width of the curve representing an edge. By default, the value is set to 1.

MarkerSize: (optional) a float variable. This allows to control the size of the marker representing the vertices of the graph. The default value is 20.

FigName: (optional) a string variable. By default, the name of the figure is `'QGraph'`. The user can change the name of the figure.

Return:

fig: the figure Matplotlib instance containing the axes `ax`.

ax: the axes Matplotlib instance containing the plot of `g`.

A.1.4. *Diag.* A method constructing a diagonal matrix with respect to the discretization points on the graph. The diagonal is explicitly prescribed.

M = Diag(g,diag_vect)

Parameters:

g: an instance of the Graph class.

diag_vect: either an instance of WFGraph or a 1-dimensional `numpy.array` corresponding to the desired diagonal.

Return:

M: a matrix whose diagonal corresponds to `diag_vect`. It is a 2-dimensional `scipy.sparse` instance. The sparse format of this instance is `csc` (Compressed Sparse Columns).

A.2. **Methods from the WFGraph class.**

A.2.1. *__init__.* The WFGraph class constructor builds a discrete function that is described on a discretized graph (given by an instance of the Graph class).

psi = WFGraph(initWF,g,Dtype)

Parameters:

initWF: either a dictionary whose keys are the identifiers of edges of `g` and whose values are `lambda` functions with a single argument (say `x`) describing the desired function in an analytical way on the corresponding edge or a 1-dimensional `numpy.array` instance which corresponds to the discretized function on the discretization points of `g`. Note that, in the first case, the variable `x` will take values between 0 and the length of the edge (starting at the node corresponding to the first coordinate of the edge's identifier).

g: (optional) an instance of `Graph` on which the function is described. If it has already been set in a previous instance of `WFGraph`, it does not need to be prescribed again.

Dtype: (optional) a string set by default to `'float'`. The default data type for `numpy.arrays` is `np.float64`. It is possible to switch to complex arrays by setting `Dtype = 'complex'`.

Return:

psi: an instance of the WFGraph class which contains `vect`, a 1-dimensional `numpy.array` associated to the discretization points of the graph `g`.

A.2.2. *norm.* This method enables to compute the ℓ^p-norm of a discrete function on a graph. It is computed with a trapezoidal rule on each vertex of the graph.

a = norm(psi,p)

Parameters:

psi: an instance of the WFGraph class whose norm is computed on its associated graph.

p: a scalar value that corresponds to the exponent of the ℓ^p space.

Return:

a: a scalar value that is the ℓ^p-norm of **psi** on its graph.

A.2.3. *dot.* A method that computes the ℓ^2 (hermitian) inner product between two discrete functions on a graph.

a = dot(psi,phi)

Parameters:

psi: an instance of the WFGraph class.
phi: an instance of the WFGraph class.

Return:

a: a (complex) scalar value that corresponds to the inner product between **psi** and **phi** on their associated graph.

A.2.4. *draw.* A method to plot an instance `f` of the WFGraph class and the graph `g` (instance of the Graph class) in the Matplotlib figure named `'Wave function on the graph'`. Each vertex of `g` is represented as a dot and its associated label is displayed.

draw(f,data_plot,fig_name,Text,AxId,ColorWF,ColorG,TextSize,LineWidth,MarkerSize,... LineWidthG,AlphaG,xlim,ylim)

Parameters:

f: an instance of the WFGraph class.
data_plot: (optional) a list of elements (matplotlib primitives) representing the plot of `f` already existing in the figure. This variable allows to efficiently update the figure containing the plot of the wave function `f` without redrawing all the scene.
fig_name: (optional) a string variable. By default, the name of the figure is `'Wave function on the graph'`. The user can change the name of the figure.
Text: (optional) Logical variable. This option allows to control the display of the labels of the vertices of `g`. By default, `Text=True`. To avoid the display of labels, set `Text=False`.
AxId: (optional) an `Axes` instance of the Matplotlib library. Allow to draw `f` and `g` in an already existing axes.
ColorWF: (optional) by default, the color of `f` is blue. It allows to specify an alternative color. The user must follow the standard naming color of Matplotlib library.
ColorG: (optional) by default, the color of `g` is dark gray. It allows to specify an alternative color. The user must follow the standard naming color of Matplotlib library.
TextSize: (optional) a float variable. This allows to control the text size to display labels of vertices of `g`. By default, the text size parameter is set to 10.
LineWidth: (optional) a float variable. This allows to control the width of the curve representing `f`. By default, the value is set to 1.5.
MarkerSize: (optional) a float variable. This allows to control the size of the marker representing the vertices of `g`. The default value is 10.
LineWidthG: (optional) a float variable. This allows to control the width of the curves representing the edges of `g`. By default, the value is set to 0.8.
AlphaG: (optional) a float variable belonging to $[0, 1]$. It allows to adjust the transparency (alpha property) of the graph `g` (both the edges, markers and labels). By defaults, the value is set to 1. If the user chooses `AlphaG=0`, the graph `g` is completely transparent and does not appear.
xlim: (optional) a two-components list instance allowing to adjust the x-axis view limits.
ylim: (optional) a two-components list instance allowing to adjust the y-axis view limits.

Return:

K: a list of elements (matplotlib primitives) representing the plot of `f` in `ax`.
fig: the figure Matplotlib instance the axes `ax`.
ax: the axes Matplotlib instance containing the plot of `f` and `g`.

A.2.5. *Arithmetic operations:* `+`, `-`, `*` *and* `/`. The basic arithmetic operations can be applied to two instances of `WFGraph`. As a matter of fact, these operations are carried pointwise on the `vect` associated to each instance. The output is an instance of `WFGraph` with the resulting `vect` associated.

A.2.6. *Mathematical functions:* `abs`*,* `Real`*,* `Imag`*,* `**`*,* `exp`*,* `cos`*,* `sin` *and* `log`*.* Some basic mathematical functions can be applied to an instance of `WFGraph`. It turns out that the function is applied pointwise on the `vect` associated to the instance. The output is an instance of `WFGraph` with the resulting `vect` associated.

A.2.7. *Lap.* This method applies the (finite-differences) discretization of the Laplace operator to a discrete function on a graph.

phi = Lap(psi)

Parameters:

psi: an instance of the WFGraph class on which the discrete Laplace operator is applied (specifically, on its associated `vect`).

Return:

phi: an instance of the WFGraph class.

A.2.8. *Solve.* A method that solves a linear system where the matrix is understood as a discrete operator and the right-hand-side is understood as a discrete function on a graph.

phi = Solve(M,psi)

Parameters:

M: a 2-dimensional `scipy.sparse` instance which is associated to a discrete operator on the graph of **psi** and that we formally want to inverse.
psi: an instance of the WFGraph class which correspond to the right-hand-side of the linear system.

Return:

phi: an instance of the WFGraph class.

References

[1] R. Adami, C. Cacciapuoti, D. Finco, and D. Noja. Stationary states of NLS on star graphs. *EPL (Europhysics Letters)*, 100(1):10003, 2012.
[2] R. Adami, C. Cacciapuoti, D. Finco, and D. Noja. Constrained energy minimization and orbital stability for the NLS equation on a star graph. *Ann. Inst. H. Poincaré Anal. Non Linéaire*, 31(6):1289–1310, 2014.
[3] R. Adami and S. Dovetta. One-dimensional versions of three-dimensional system: ground states for the NLS on the spatial grid. *Rend. Mat. Appl. (7)*, 39(2):181–194, 2018.
[4] R. Adami, S. Dovetta, and A. Ruighi. Quantum graphs and dimensional crossover: the honeycomb. *Commun. Appl. Ind. Math.*, 10(1):109–122, 2019.
[5] R. Adami, S. Dovetta, E. Serra, and P. Tilli. Dimensional crossover with a continuum of critical exponents for NLS on doubly periodic metric graphs. *Anal. PDE*, 12(6):1597–1612, 2019.
[6] R. Adami, E. Serra, and P. Tilli. NLS ground states on graphs. *Calc. Var. Partial Differential Equations*, 54(1):743–761, 2015.
[7] R. Adami, E. Serra, and P. Tilli. Threshold phenomena and existence results for NLS ground states on metric graphs. *J. Funct. Anal.*, 271(1):201–223, 2016.
[8] R. Adami, E. Serra, and P. Tilli. Nonlinear dynamics on branched structures and networks. *Riv. Math. Univ. Parma (N.S.)*, 8(1):109–159, 2017.
[9] R. Adami, E. Serra, and P. Tilli. Multiple positive bound states for the subcritical NLS equation on metric graphs. *Calc. Var. Partial Differential Equations*, 58(1):Art. 5, 16, 2019.
[10] F. Ali Mehmeti. *Nonlinear waves in networks*, volume 80 of *Mathematical Research*. Akademie-Verlag, Berlin, 1994.
[11] F. Ali Mehmeti, J. von Below, and S. Nicaise, editors. *Partial differential equations on multistructures*, volume 219 of *Lecture Notes in Pure and Applied Mathematics*. Marcel Dekker, Inc., New York, 2001.
[12] X. Antoine, A. Levitt, and Q. Tang. Efficient spectral computation of the stationary states of rotating Bose–Einstein condensates by preconditioned nonlinear conjugate gradient methods. *Journal of Computational Physics*, 343:92–109, 2017.
[13] V. Banica and L. I. Ignat. Dispersion for the Schrödinger equation on the line with multiple Dirac delta potentials and on delta trees. *Anal. PDE*, 7(4):903–927, 2014.
[14] G. Berkolaiko and P. Kuchment. *Introduction to quantum graphs*, volume 186 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, 2013.
[15] G. Berkolaiko, J. L. Marzuola, and D. E. Pelinovsky. Edge-localized states on quantum graphs in the limit of large mass, 2019.
[16] C. Besse. A relaxation scheme for the nonlinear Schrödinger equation. *SIAM Journal on Numerical Analysis*, 42(3):934–952, 2004.
[17] C. Besse, R. Duboscq, and S. Le Coz. Gradient Flow Approach to the Calculation of Ground States on Nonlinear Quantum Graphs. arXiv:2006.04404, June 2020.
[18] C. Besse, R. Duboscq, and S. Le Coz. Grafidi. *PLMlab repository*, `https://plmlab.math.cnrs.fr/cbesse/grafidi`, 2021.
[19] K. Bhandari, F. Boyer, and V. Hernández-Santamaría. Boundary null-controllability of 1-D coupled parabolic systems with Kirchhoff-type condition. working paper or preprint, hal-02748405, July 2020.

[20] 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(2):305–327, 2018.

[21] C. Cacciapuoti, D. Finco, and D. Noja. Topology-induced bifurcations for the nonlinear Schrödinger equation on the tadpole graph. *Phys. Rev. E (3)*, 91(1):013206, 8, 2015.

[22] I. Danaila and B. Protas. Computation of ground states of the Gross–Pitaevskii functional via Riemannian optimization. *SIAM Journal on Scientific Computing*, 39(6):B1102–B1129, 2017.

[23] M. Delfour, M. Fortin, and G. Payr. Finite-difference solutions of a non-linear schrödinger equation. *Journal of computational physics*, 44(2):277–288, 1981.

[24] S. Dovetta. Existence of infinitely many stationary solutions of the L^2-subcritical and critical NLSE on compact metric graphs. *J. Differential Equations*, 264(7):4806–4821, 2018.

[25] S. Dovetta. Mass-constrained ground states of the stationary NLSE on periodic metric graphs. *NoDEA Nonlinear Differential Equations Appl.*, 26(5):Paper No. 30, 30, 2019.

[26] S. Dovetta, M. Ghimenti, A. M. Micheletti, and A. Pistoia. Peaked and low action solutions of nls equations on graphs with terminal edges. *SIAM Journal on Mathematical Analysis*, 52(3):2874–2894, 2020.

[27] S. Dovetta, E. Serra, and P. Tilli. Nls ground states on metric trees: existence results and open questions. *Journal of the London Mathematical Society*, n/a(n/a), 2020.

[28] S. Gilg, D. Pelinovsky, and G. Schneider. Validity of the NLS approximation for periodic quantum graphs. *NoDEA Nonlinear Differential Equations Appl.*, 23(6):Art. 63, 30, 2016.

[29] S. Gnutzmann and D. Waltner. Stationary waves on nonlinear quantum graphs: general framework and canonical perturbation theory. *Phys. Rev. E*, 93(3):032204, 19, 2016.

[30] R. H. Goodman. NLS bifurcations on the bowtie combinatorial graph and the dumbbell metric graph. *Discrete Contin. Dyn. Syst.*, 39(4):2203–2232, 2019.

[31] S. Gustafson, S. Le Coz, and T.-P. Tsai. Stability of periodic waves of 1D cubic nonlinear Schrödinger equations. *Appl. Math. Res. Express. AMRX*, 2:431–487, 2017.

[32] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, Aug. 2008.

[33] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[34] N. V. Hung, M. Trippenbach, and B. A. Malomed. Symmetric and asymmetric solitons trapped in H-shaped potentials. *Phys. Rev. A*, 84:053618, Nov 2011.

[35] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[36] I. Ianni, S. Le Coz, and J. Royer. On the Cauchy problem and the black solitons of a singularly perturbed Gross-Pitaevskii equation. *SIAM J. Math. Anal.*, 49(2):1060–1099, 2017.

[37] A. Kairzhan, R. Marangell, D. E. Pelinovsky, and K. L. Xiao. Standing waves on a flower graph, 2020.

[38] K. Kurata and M. Shibata. Least energy solutions to semi-linear elliptic problems on metric graphs. *Journal of Mathematical Analysis and Applications*, 491(1):124297, 2020.

[39] S. Le Coz, R. Fukuizumi, G. Fibich, B. Ksherim, and Y. Sivan. Instability of bound states of a nonlinear Schrödinger equation with a Dirac potential. *Phys. D*, 237(8):1103–1128, 2008.

[40] J. L. Marzuola and D. E. Pelinovsky. Ground State on the Dumbbell Graph. *Appl. Math. Res. Express. AMRX*, 2016(1):98–145, 2016.

[41] F. A. Mehmeti, K. Ammari, and S. Nicaise. Dispersive effects and high frequency behaviour for the Schrödinger equation in star-shaped networks. *Port. Math.*, 72(4):309–355, 2015.

[42] F. A. Mehmeti, K. Ammari, and S. Nicaise. Dispersive effects for the Schrödinger equation on the tadpole graph. *J. Math. Anal. Appl.*, 448(1):262–280, 2017.

[43] D. Noja. Nonlinear Schrödinger equation on graphs: recent results and open problems. *Philos. Trans. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci.*, 372(2007):20130002, 20, 2014.

[44] D. Noja, D. Pelinovsky, and G. Shaikhova. Bifurcations and stability of standing waves in the nonlinear Schrödinger equation on the tadpole graph. *Nonlinearity*, 28(7):2343–2378, 2015.

[45] D. Noja and D. E. Pelinovsky. Standing waves of the quintic NLS equation on the tadpole graph, 2020.

[46] A. Pankov. Nonlinear Schrödinger equations on periodic metric graphs. *Discrete Contin. Dyn. Syst.*, 38(2):697–714, 2018.

[47] D. Pelinovsky and G. Schneider. Bifurcations of standing localized waves on periodic graphs. *Ann. Henri Poincaré*, 18(4):1185–1211, 2017.

[48] D. Pierotti, N. Soave, and G. Verzini. Local minimizers in absence of ground states for the critical NLS energy on metric graphs. *Proceedings of the Royal Society of Edinburgh: Section A Mathematics*, page 1–29, 2020.

[49] K. K. Sabirov, Z. A. Sobirov, D. Babajanov, and D. U. Matrasulov. Stationary nonlinear Schrödinger equation on simplest graphs. *Phys. Lett. A*, 377(12):860–865, 2013.

[50] Z. Sobirov, D. Babajanov, and D. Matrasulov. Nonlinear standing waves on planar branched systems: shrinking into metric graph. *Nanosystems: Physics, Chemistry, Mathematics*, 8(1):29, 2017.

[51] G. Strang. On the construction and comparison of difference schemes. *SIAM journal on numerical analysis*, 5(3):506–517, 1968.

[52] A. Tokuno, M. Oshikawa, and E. Demler. Dynamics of One-Dimensional Bose Liquids: Andreev-Like Reflection at Y Junctions and the Absence of the Aharonov-Bohm Effect. *Phys. Rev. Lett.*, 100:140402, Apr 2008.

[53] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson,

C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[54] J. Weideman and B. Herbst. Split-step methods for the solution of the nonlinear Schrödinger equation. *SIAM Journal on Numerical Analysis*, 23(3):485–507, 1986.

(Christophe Besse and Stefan Le Coz) Institut de Mathématiques de Toulouse ; UMR5219,
Université de Toulouse ; CNRS,
UPS IMT, F-31062 Toulouse Cedex 9,
France
Email address, Christophe Besse: Christophe.Besse@math.univ-toulouse.fr
Email address, Stefan Le Coz: stefan.lecoz@math.cnrs.fr

(Romain Duboscq) Institut de Mathématiques de Toulouse ; UMR5219,
Université de Toulouse ; CNRS,
INSA IMT, F-31077 Toulouse,
France
Email address, Romain Duboscq: Romain.Duboscq@math.univ-toulouse.fr