

Computational Physics

Project – 2D FEM Schrödinger solver

Kassius Kohvakka, 586977

1 Background

The finite element method (FEM) is a numerical method for solving boundary value problems for partial differential equations used throughout engineering and the sciences. Numerical methods are required in cases where analytical solutions to the problems are intractable to obtain or require abstractions and simplifications which make the solution not applicable in real life. Particularly in engineering, such methods are vital in designing, for instance, load-bearing structures, complicated machine parts, and vehicle chassis which can safely withstand the forces at play and – if need be – give in without threatening the lives of the passengers. Problems relating to heat transfer, wave propagation, fluid flow, electromagnetic fields, and other physical phenomena governed by laws expressed in terms of partial differential equations are omnipresent in engineering as well as physics. [1]

One such problem is solving the time-independent Schrödinger equation. The stationary states of a quantum mechanical system, such as a particle in an external potential, are found by computing the eigenstates of the Hamiltonian operator \hat{H} defined by partial derivatives. While analytic solutions for a few simple potentials are known (e.g. particle-in-a-box and the harmonic oscillator), no analytic solution exists for an arbitrary potential. Thus, numerical computations are required to find approximate solutions in challenging geometries or non-trivial potentials.

The standard approach to the problem is the discretization of the derivatives present in the Hamiltonian, leading to a matrix representation of the Hamiltonian and an eigenproblem for an ordinary matrix. The two common discretization methods, namely, finite difference and FEM, both can be used to this end. The major advantage of the latter over the former is the ability to freely choose the size of the finite elements (triangles in 2D) at each point in the problem domain. This allows significant reductions in the computational cost and memory requirements by using a less dense mesh in regions of lesser importance. Therefore we may take full advantage of the accuracy of the dense grid where it matters while sacrificing less resources for unimportant aspects of the problem [2].

In this project, I implemented a FEM solver for the time-independent Schrödinger equation in two dimensions. The correctness of the solution is assessed by taking a look at the convergence of the energy eigenvalues. Subsequently, the solver is tested for a non-trivial potential. The code was written in python and takes advantage of some of the numerical computation libraries, such as numpy and scipy. The code is visible in a public GitHub repository in <https://github.com/kassiuskohvakka/CP-project>.

2 Theory and methods

The two-dimensional time-independent Schrödinger equation is given by [3]

$$-\frac{\hbar}{2m} \left(\frac{\partial^2 \psi(x,y)}{\partial x^2} + \frac{\partial^2 \psi(x,y)}{\partial y^2} \right) + V(x,y)\psi(x,y) = E\psi(x,y). \quad (1)$$

In order to use FEM to numerically solve Eq. (1), we expand the wave function $\psi(x,y)$ in a basis of tetrahedral hat functions $\{\phi_i\}_{i=1..N}$ as

$$\psi(x,y) \approx \sum_{i=1}^N \alpha_i \phi_i(x,y), \quad (2)$$

where N is the number of finite elements used in our computation and α_i , the coefficients in the linear combination are to be solved for. The linear combination then allows us to construct an approximate solution to the original problem. Setting, for simplicity, $\frac{\hbar}{m} = 1$, writing the partial derivatives more concisely as $\nabla^2 \psi(x,y)$, and substituting our basis expansion in Eq. (1), we get

$$\left(-\frac{1}{2} \nabla^2 + V(x,y) \right) \sum_{i=1}^N \alpha_i \phi_i(x,y) = E \sum_{i=1}^N \alpha_i \phi_i(x,y). \quad (3)$$

Rearranging and multiplying both sides by the basis function ϕ_j , we acquire

$$\sum_{i=1}^N \left[-\frac{1}{2} \phi_j(x,y) \nabla^2 \phi_i(x,y) + \phi_j(x,y) V(x,y) \phi_i(x,y) \right] \alpha_i = E \sum_{i=1}^N \alpha_i \phi_j(x,y) \phi_i(x,y). \quad (4)$$

We can now integrate both sides over the domain Ω of our problem (and lighten the notation by getting rid of the cluttering (x,y) -silliness) to get

$$\sum_{i=1}^N \left[-\frac{1}{2} \left(\int_{\Omega} \phi_j \nabla^2 \phi_i \, dA \right) + \left(\int_{\Omega} \phi_j V \phi_i \, dA \right) \right] \alpha_i = \sum_{i=1}^N E \left(\int_{\Omega} \phi_j \phi_i \, dA \right) \alpha_i. \quad (5)$$

The integrals inside the ordinary parentheses are now matrices. The first of the three still needs to be rewritten by Green's first identity [4]:

$$\int_{\Omega} \phi_j \nabla^2 \phi_i \, dA = \underbrace{\oint_{\partial\Omega} \phi_j (\nabla \phi_i \cdot \hat{n}) \, dl}_{=0} - \int_{\Omega} \nabla \phi_j \cdot \nabla \phi_i \, dA, \quad (6)$$

where the vanishing of the indicated term can be achieved in practice by setting either the basis functions $\{\phi_i\}_i$ or the normal-directional derivatives $\{\nabla\phi_i \cdot \hat{n}\}_i$ at the boundary $\partial\Omega$ to 0 by use of Dirichlet or Neumann boundary conditions, respectively. We can then finally identify the matrices in Eq. (5) as the kinetic matrix T_{ji} , the potential matrix V_{ji} and the overlap matrix S_{ji} :

$$\sum_{i=1}^N \left[\underbrace{\frac{1}{2} \left(\int_{\Omega} \nabla\phi_j \cdot \nabla\phi_i \, dA \right)}_{T_{ji}} + \underbrace{\left(\int_{\Omega} \phi_j V \phi_i \, dA \right)}_{V_{ji}} \right] \alpha_i = \sum_{i=1}^N E \underbrace{\left(\int_{\Omega} \phi_j \phi_i \, dA \right)}_{S_{ji}} \alpha_i. \quad (7)$$

Since the summations on both sides of the equation are just the j^{th} elements of a matrix-vector product, the elementwise equality implies equality of the resultant vectors and we get

$$(T + V)\alpha = ES\alpha, \quad (8)$$

which is a generalized eigenvalue problem involving our known matrices. Solving this, we acquire as eigenvectors the coefficient vectors α approximating the true eigenstates as per the linear combination (2) and the corresponding approximate energies E of the eigenstates as eigenvalues.

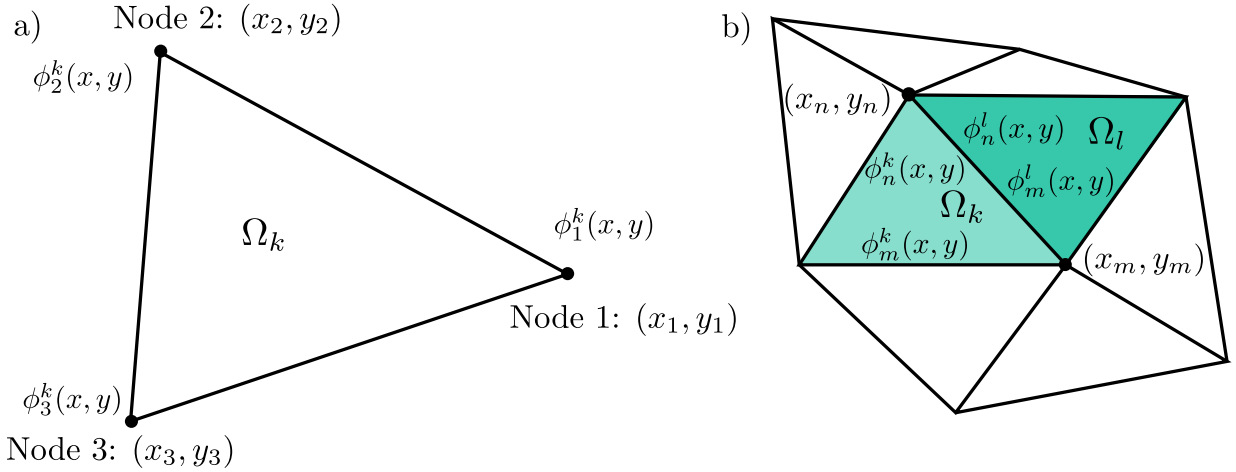


Figure 1: a) A triangular element Ω_k of the domain together with the three vertices or nodes, each being the center of a FEM basis function ϕ . Note that at each triangle, only three basis functions are nonzero, and each of them is affine over Ω_k . b) Illustration of the triangulation. The basis functions centered at (x_n, y_n) and (x_m, y_m) overlap only in two triangular elements Ω_k and Ω_l . Note that in both domains, the functions are affine but not the same as FEM basis functions are piecewise linear.

The triangulation of the domain $\Omega = [0,1] \times [0,1]$ was done by using `pygmsh`, a python wrapper for the mesh generator tool Gmsh [5]. Having constructed the triangular subdomains $\{\Omega_k\}_k$, construction of the required matrices can be achieved by integrating suitable functions over the subdomains. Integration over an arbitrary triangle can be done by a number of quadratures and was done in practice by utilizing `quadpy`, a python library for numerical integration. The generalized eigenvalue problem was solved using `scipy.linalg.eigh`, a library solver for eigenvalue problems involving Hermitian matrices.

The explicit formulas for the piecewise affine hat basis functions for each of the subdomains must be constructed to compute T , V and S . Over the arbitrary subdomain Ω_k the relevant basis functions located at the vertices are given by

$$\phi_1^k(x,y) = \frac{1}{2A_k} [(x_2y_3 - x_3y_2) + (y_2 - y_3)x + (x_3 - x_2)y] \quad (9)$$

$$\phi_2^k(x,y) = \frac{1}{2A_k} [(x_3y_1 - x_1y_3) + (y_3 - y_1)x + (x_1 - x_3)y] \quad (10)$$

$$\phi_3^k(x,y) = \frac{1}{2A_k} [(x_1y_2 - x_2y_1) + (y_1 - y_2)x + (x_2 - x_1)y] \quad (11)$$

where A_k is the area of the triangle Ω_k and the labelling of the nodes and basis functions is as in Fig. 1 a). These functions and their simple gradients are easily implemented by some modular arithmetic magic.

After defining a suitable potential function $V(x,y)$, we are ready to start constructing the matrices. Each of the matrices is of size $N \times N$, with row i containing nonzero elements on all columns j if there exists an edge connecting nodes i and j , i.e. if the basis functions ϕ_i and ϕ_j overlap. The value of each of the nonzero elements is the sum of the integral of the relevant functions over the two (or, on the boundary, just one) common triangular subdomains. See Fig. 1 b) and the colored relevant subdomains. Computing the integrals in Eq. (7) and carefully inserting them into the matrices, we are left with all the pieces needed to solve the generalized eigenvalue problem in Eq. (8).

3 Results

3.1 Testing solver and convergence

The initial tests were carried out with simple Gaussian potential minimum in a $[0,1] \times [0,1]$ unit square. The potential landscape is shown in Fig. 2 along with the mesh points created by the `pygmsh` triangulation. The solver was run for this potential and triangulation and the resulting eigenstates and -energies are shown in Fig. 3. Visual inspection reveals at least that there are no catastrophic errors in the matrices or the solver.

In addition to simple visual inspection, we can also, for instance, check whether the energy eigenvalues got from the solver seem to be converging to a given value as the dimension N of the FEM basis is increased. The apparent convergence of the five lowest energy eigenvalues is shown in Fig. 4. At around $N = 3500$, the eigenvalues seem to no longer be changing sporadically, implying that our FEM approximation may be getting quite close to the true solution. Given large enough basis dimension, our solver therefore seems to be solving the problem correctly.

The Dirichlet boundary conditions were enforced by removing the rows and columns corresponding to boundary points from the matrices H and S before solving the generalized eigenvalue problem, effectively setting the coefficients of the FEM basis functions centered at nodes on the boundary to zero. In some simulations and with some mesh sizes, this seemed to create stability issues in the solver. To counteract these problems, a high potential barrier was introduced to the boundaries of the problem domain. Approximating the infinite potential barrier normally achieved by simply enforcing the zero boundary condition, this seemed to be enough to stabilize the solutions.

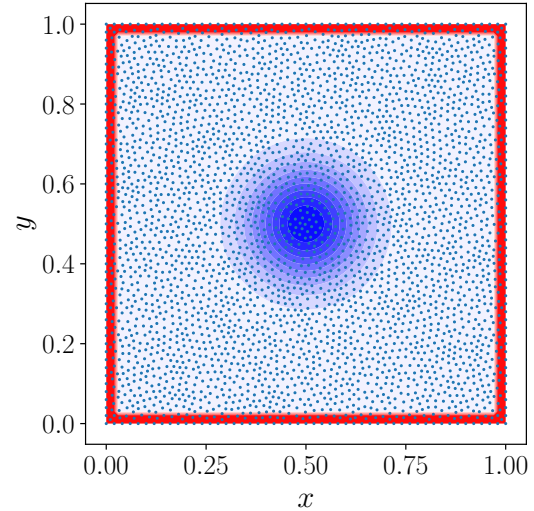


Figure 2: The potential landscape used in initial tests of the FEM solver. The blue region in the middle is the Gaussian potential minimum and the red region is a high potential barrier approximating an infinite potential well. The blue dots are the triangulation nodes for $N = 3436$.

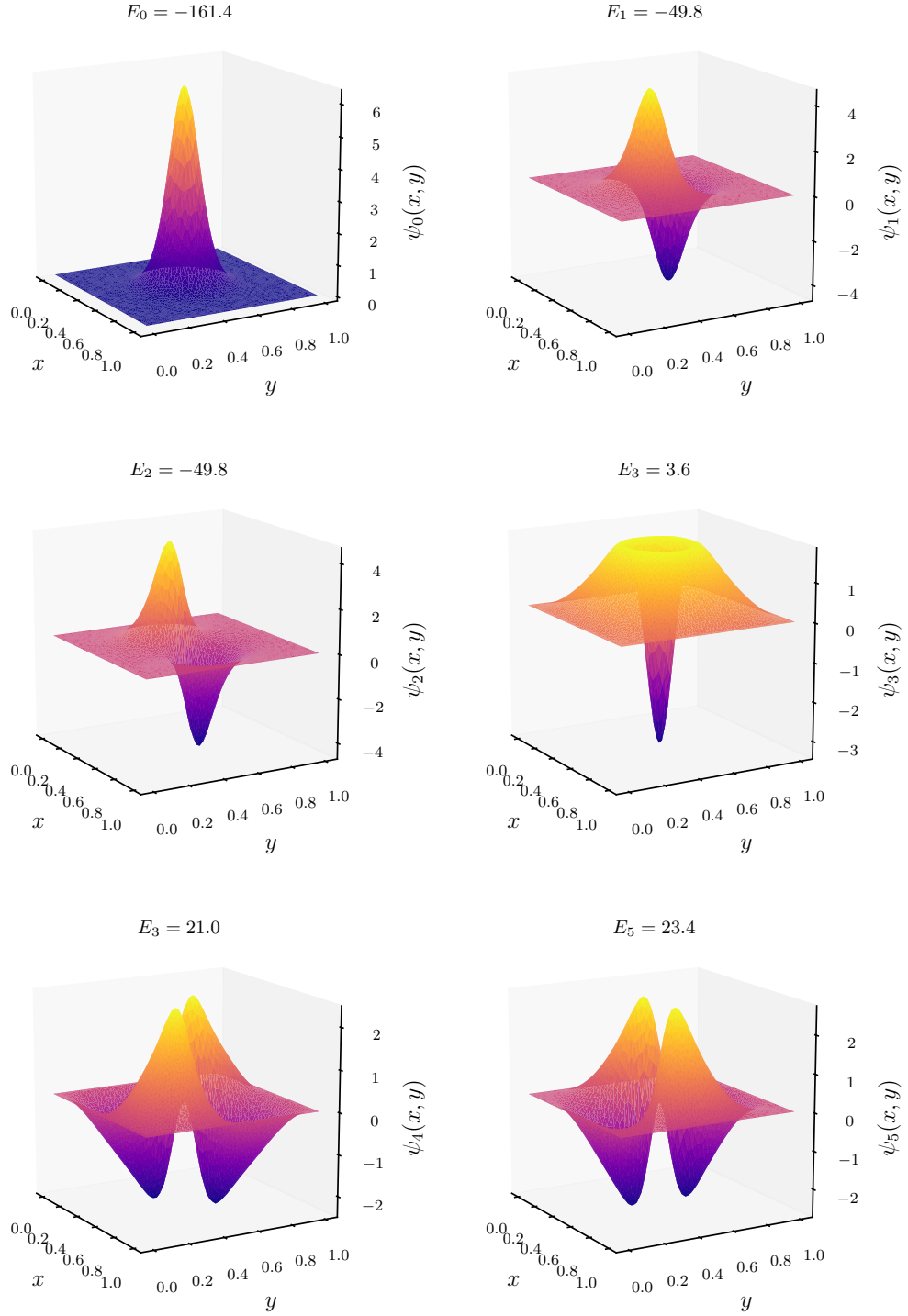


Figure 3: The six lowest-energy eigenstates in the Gaussian potential well with $N = 3436$. Note that it seems based on our FEM solution that only three of the states are bound to the potential minimum. Notice also the near-degenerate state pairs (ψ_1, ψ_2) and (ψ_4, ψ_5) .

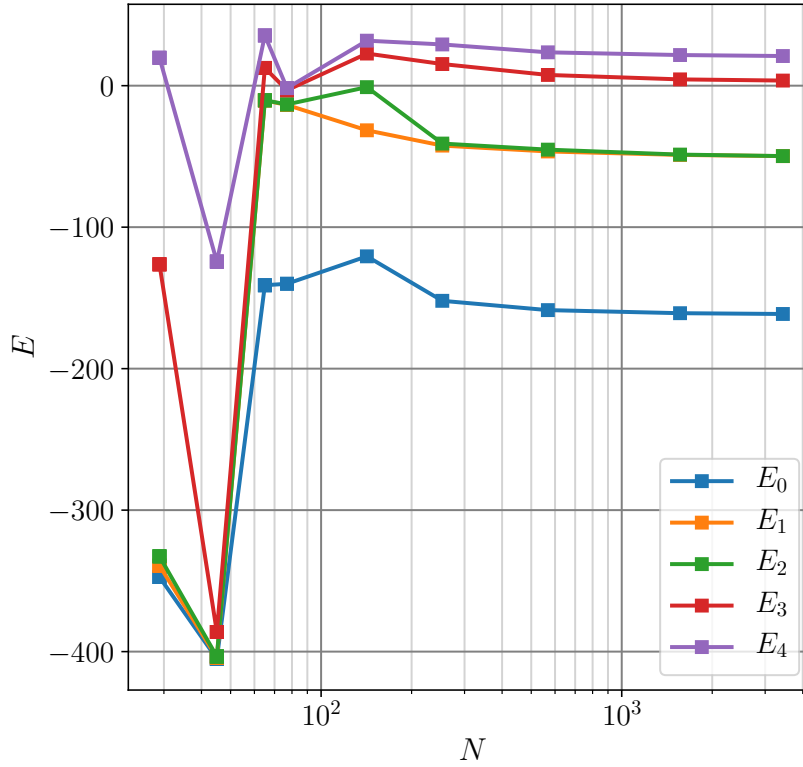


Figure 4: The convergence of the energy eigenvalues as the FEM basis dimension N increases.

3.2 More interesting potential – less uniform mesh

To inspect the behaviour of the solver in a different potential landscape, we define a 2D ring-shaped potential well, surrounded on the inside and outside by a high potential. Next, we would like to achieve something like an adaptive meshing scheme to try out our solver with a noticeably non-uniform mesh and to keep our FEM basis dimension manageably low while spending our mesh points where it matters, around areas of interesting eigenstate behaviour.

I quickly realized that as it stands, the `pygmsh` tool kit used in the triangulation lends itself quite poorly to adaptive meshing, so the next best thing was done. The triangulation software was more manually instructed to create smaller finite elements around the ring-shaped potential well, while creating larger triangles both outside and inside the ring. Both the potential landscape and the non-uniformly distributed mesh points created by the modified triangulation algorithm are shown in Fig. 5.

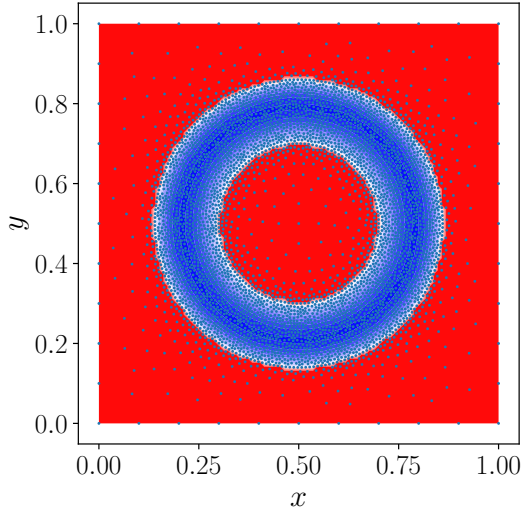


Figure 5: The ring-shaped potential landscape. The blue (red) color denotes low (high) potential. The blue dots are the triangulation nodes for $N = 4763$. Note that the distribution of the nodes is highly non-uniform.

The modified mesh does little to make solving the generalized eigenproblem any harder. Fig. 6 shows the six lowest-energy eigenstates inside the ring-shaped potential well. The eigenstates seem to obey intuition as increase in energy generally translates directly in an addition of a symmetrically-distributed antinode. Interestingly, the more accurate mesh seems to be able to differentiate between the two near-degenerate states ψ_1 and ψ_2 , whose only difference is the orientation of the line connecting the antinodal points. Possibly due to the boundary being so close by and rectangular, the diagonally-oriented antinodes of ψ_1 lead to an ever so slightly lower energy in comparison to ψ_2 whose antinodes are more oriented along the x -axis. Similarly, in eigenstates ψ_3 and ψ_4 the only difference is the diagonal vs. axial orientation. Here as well the breaking of the

symmetry is due to the proximity of the rectangular boundary and should vanish by increasing the simulation box size or by e.g. making the box itself spherically symmetrical.

4 Conclusions and further studies

Both the visual inspection of the eigenstates produced by the FEM solver and the apparent convergence of the energy eigenvalues seem to indicate that the solver is working as intended. However, the solver is not terribly efficient especially since all attempts to multithread the construction of the matrices in python seemed only to increase the total running time. This was to be anticipated, however, since python's Global Interpreter Lock [6] effectively prevents multicore execution of separate threads. The performance was not completely awful, as solving the ring-shaped potential well problem with $N = 4763$ took 76.68 seconds. Increasing the FEM basis dimension further would quickly result in an untrackable running time. With proper parallelization using a suitable (non-interpreted) language, such as C++, the performance could be improved significantly. The vectorization capabilities present in, for instance, the library functions used to integrate over the triangular domains lended themselves poorly to our needs as the integrands differ significantly from domain to domain.

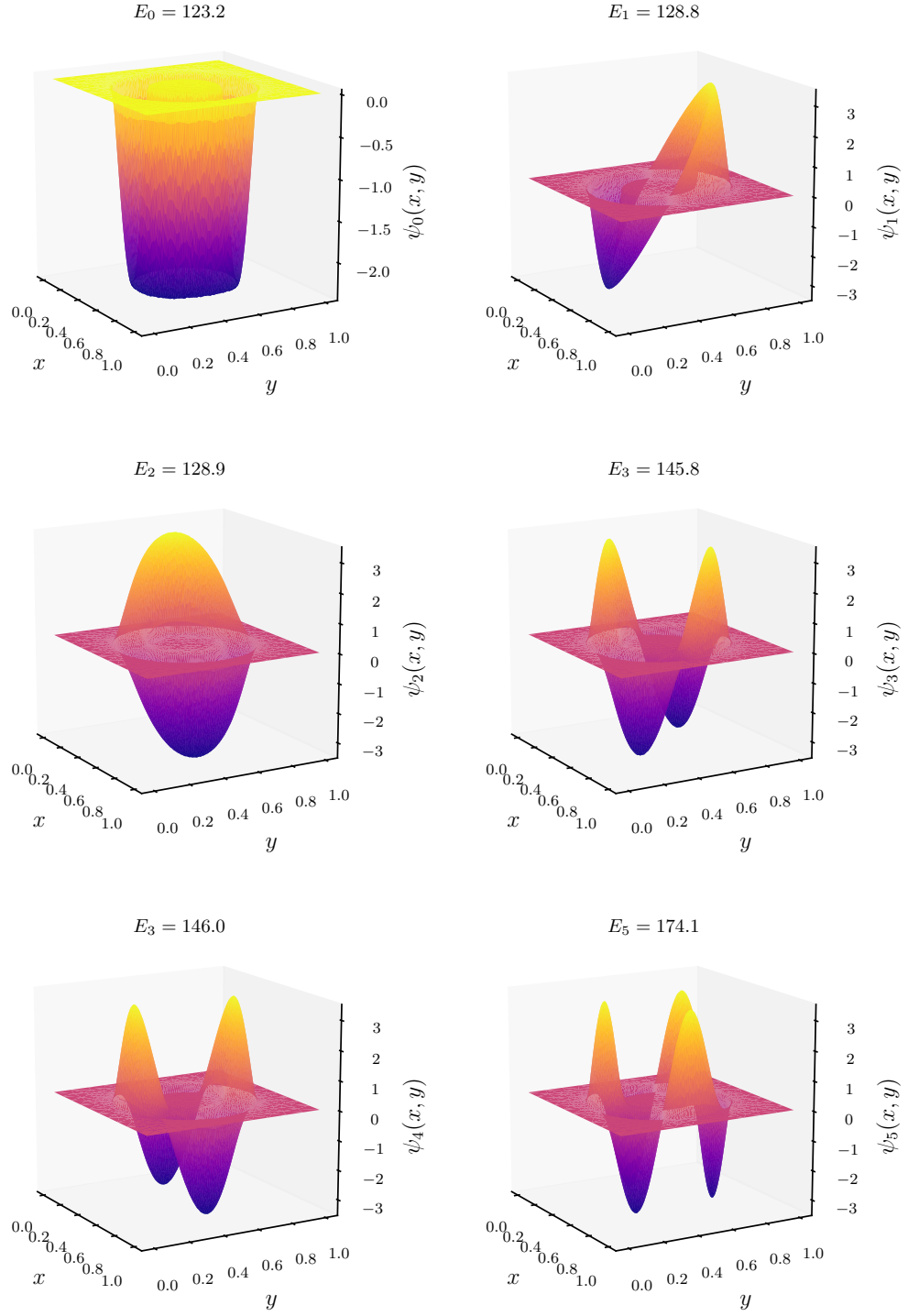


Figure 6: The six lowest-energy eigenstates in the ring-shaped potential well with $N = 4763$.

One interesting avenue for further studies would be the implementation of a more automatic adaptive meshing scheme. As stated before, the `pygmsh` library does not offer ready-made tools for mesh refinement, but programmatically inserting additional points in the middle of some of the finite element triangles based on e.g. a large value of the laplacian of the eigenstate in the domain might be reasonably straight-forward to implement. The implementation of adaptive meshing would have raised the performance issue in the present code to new heights as the matrix constructions and eigenvalue solutions would have to be redone for each mesh size.

A less manual mesh creation software would also allow for a reasonably achieved increase in the simulation box size. The boundary effects – not too worrisome even in our tiny simulation domain – would be further suppressed by having a large domain with a comparatively small region of lower potential. With a larger domain, we could more reliably simulate, say, the electron stucture of a 2D hydrogen atom by introducing a $1/r$ -type potential to the system.

References

- [1] Singiresu S. Rao. *The Finite Element Method in Engineering*. Elsevier, 2004. ISBN: 0750678283.
- [2] David V. Hutton. *Fundamentals of Finite Element Analysis*. McGraw-Hill, 2004. ISBN: 0072233411.
- [3] David J. Griffiths. *Introduction to Quantum Mechanics*. Prentice Hall, 1995. ISBN: 0131244051.
- [4] David J. Griffiths. *Introduction to Electrodynamics*. Cambridge University Press, 2017. ISBN: 9781108420419.
- [5] *Pygmsh 5.0.1 documentation*. <https://pypi.org/project/pygmsh/>. Accessed: 2019-05-23.
- [6] *Python documentation, GlobalInterpreterLock*. <https://wiki.python.org/moin/GlobalInterpreterLock>. Accessed: 2019-05-19.