Applying a Genetic Algorithm to the Variational Method in Quantum Mechanics

Roark Habegger

November 27, 2019

Abstract

This project adapts a Real-code Genetic Algorithm in Python to determine the minimum energy states of quantum mechanics problems. The algorithm uses the Distributed Evolutionary Algorithms in Python (DEAP) Library. After testing the algorithm on problems with known answers, we examine a quantum dot's ground state energy. This problem has no exact solution, which makes the variational method an effective way to determine the ground state energy. We use the results of Yan et al. (2005) as a comparison, to determine the accuracy of the results of the current algorithm. The code and example problem files are available for download on GitHub under the repository *roarkhabegger/QMGA*. The main program and a single problem file appear in the appendices of this paper.

1 Introduction

Genetic algorithms solve optimization problems from a variety of domains [3]. In quantum mechanics, problems hinge on finding a solution to the Schrodinger equation, a partial differential equation in time and space. Even in problems without time dependence there are situations and configurations which make it impossible to find an exact solution to the time-independent Schrodinger equation. In that case, one method of solving the problem is the *variational* method. This method finds an upper bound on the the minimum, or *ground*, energy state of the system. The variational method turns the differential equation into an optimization problem. Beginning with a test function consisting of n parameters, the variational method produces a *functional* $E[\phi]$. We minimize this functional with respect to the parameters of the test function to find an upper bound on the minimum energy value. In the simplest cases, such as single parameter test functions or functionals which reduce to two or three terms, the minimization problem becomes one of algebra. However, the variational method is general. The functional could be made of an arbitrary number of terms, and the test function could have a large number of parameters. In those situations, we turn to numerical methods to minimize the functional and determine an upper bound.

When computational optimization of the functional is necessary, genetic algorithms become effective solvers [2]. In the more complicated variational method problems, other numerical methods may identify a single local minimum, and never find the global minimum. Genetic algorithms avoid getting stuck in local minima, and this is particularly helpful if our test function has a large number of parameters. In that case, the parameter space the algorithm explores is large, and likely has multiple local minima. Even when the test function has a single parameter, the functional in certain problems has multiple minima.

2 Genetic Algorithms

A genetic, or evolutionary, algorithm explores the parameter space of a problem. The naming derives from *natural selection*, the biological theory concerned with the evolution of species, originally proposed by Charles Darwin (Introduction of [3]). The original idea for this algorithmic method came from that theory, since the method treats each parameter as a chromosome of an individual. Every *individual* of the species is a particular set of these parameters. ‘Exploring’ the parameter space means taking a number of individuals, rating them, and deciding what individuals to evaluate next. The algorithm rates each individual using a *fitness* function. The name for that evaluation derives from the aphorism “survival of the fittest” describing Darwin’s theory (Introduction of [3]). The algorithm, after evaluating every individual in the species’ population, decides what to do with the population. This step is the point where genetic algorithms separate from other minimization methods, such as gradient descent.

Other computational methods measure the ‘slope’ of the function to determine their exploration of the parameter space. They explore the direction of decrease (negative slope) expecting to find a minimum. Gradient descent, for example, finds the gradient (a multi-dimensional slope) of the function and locates the corresponding minimum in the space. It then iterates smoothly toward it. If the function traces out a paraboloid in the parameter space (Fig. 1), gradient descent is a particularly useful method. In Fig. 1, there is only one minimum value of the plotted function $f_1(x, y) = x^2 + y^2$. In this example, x and y are the ‘parameters’ of the function, and each individual in the population is a point in the (x, y) plane. If f_1 is the function we minimize, gradient descent is an effective method.

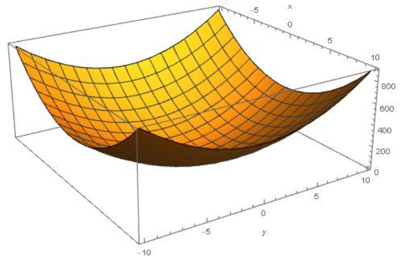

Figure 1: The plot above is a paraboloid described by the function $f_1(x, y) = x^2 + y^2$. This plot was created using Wolfram Mathematica 11.

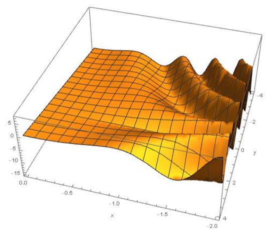

Figure 2: The plot above shows the function $f_2(x, y) = \sin(x^2|y-2|)e^{-x} + y - 4$. This plot was created using Wolfram Mathematica 11.

If we minimize $f_2(x, y) = \sin(x^2|y-2|)e^{-x} + y - 4$, plotted in Fig. 2, then gradient descent is ineffective, because there are multiple minima in the parameter space of (x, y). Using a genetic algorithm to minimize f_2 is more effective because a genetic algorithm does not only use a slope to determine its movement. Instead, it takes the concept of *mutation* from natural selection and uses it to randomly jump to other regions of parameter space, where it searches for other minima. Even if those minima are separated by a 'mountain' in the parameter space, like the folds in Fig. 2, the genetic algorithm finds them in its exploration of the space.

Genetic algorithms still use an evaluation of the 'slope' of the fitness function. This is known as a 'selection' operator. The selection operator determines the amount of the population deleted after a generation. Once the algorithm evaluates all the individuals in its population, the selection operator chooses the 'fittest' individuals, and they stay in the population. It removes the individuals who were less fit, and replaces them with either mutated individuals or 'crossed' individuals. Crossed individuals are also known as offspring. To produce offspring, the algorithm uses two individuals. It selects half of the parameters from one individual and half from the other individual. This produces a new individual, which appears in a different part of the parameter space. While selection allows the algorithm to explore closer to a nearby minimum, crossover and mutation operators allow it to explore other regions of parameter space. Crossover operators also produce individuals closer to a minimum, acting as a descent method. This happens if there is already a significant portion of the population near that minimum.

3 Distributed Evolutionary Algorithms in Python (DEAP)

The Python coding language offers access to libraries made by other users and professional software organizations. An open source library known as DEAP makes real code genetic algorithm production available to Python users. Furthermore, it offers pre-written algorithms. A highlight of these is the $(\mu + \lambda)$ algorithm. This algorithm asks for user-defined crossover, selection, and mutation operators. Then it evolves a population using those operators. This is the algorithm used to solve the quantum mechanics problems in the proceeding sections.

The main benefit of the DEAP library is it defines a toolbox object, which holds the operators, the fitness function, and analysis methods. It also offers pre-defined operators, such as Gaussian and polynomial mutation functions. For some problems, the only 'new' code that a user writes is the fitness function, which needs to take an individual and return a tuple of fitness values (a tuple is a Python data type similar to a list). Otherwise, the library is a complete framework for genetic algorithm codes.

4 Variational Method

The variational method in quantum mechanics begins with defining a functional $E[\phi]$. View the functional as a 'function' of a function, since it depends on the ϕ used in its calculation. In this case the functional acts on a function of space, $\phi(\vec{x})$. This $\phi(\vec{x})$ is an approximation of the wavefunction $\psi(\vec{x})$, and the accuracy of the ground state energy found with the variational method is limited by how similar ϕ is to ψ. This approximation ϕ (or 'guess' wavefunction), is known as a test function.

The functional is defined in terms of the expectation value of the energy of the test function $|\phi\rangle$ and its magnitude. For a given Hamiltonian $\hat{H}$,

$$E[\phi] = \frac{\langle\phi|\hat{H}|\phi\rangle}{\langle\phi|\phi\rangle} = \frac{\int d\tau\, \phi^*(\vec{x})[H \cdot \phi(\vec{x})]}{\int d\tau\, \phi^*(\vec{x}) \cdot \phi(\vec{x})} \tag{1}$$

the functional is with inner products of $|\phi\rangle$, which are integrals over a volume of space τ. Using a test function $\phi(\vec{x}|\alpha_1, \alpha_2, ..., \alpha_n)$ with n parameters, we integrate, and get the functional E as a *function* of the parameters α_i. Then, we minimize $E(\alpha_1, \alpha_2, ..., \alpha_n)$ with respect to the parameters. This gives an upper bound on the ground state energy of the problem. We plug the parameter values of this minimum energy into $\phi(\vec{x}|\alpha_1, \alpha_2, ..., \alpha_n)$ and we get the function that best approximates the ground state wavefunction ψ.

The above method is effective in approximating the ground state energy of a problem, and accurate if the chosen ϕ has the same form as the wavefunction ψ which solves the Schrodinger equation. To adapt it to the genetic algorithm, we need to define ϕ, then calculate an expression for E. An individual in the algorithm (member of the population being evolved) is an ordered set of parameter values, where we choose each parameter from an interval of real numbers. This kind of genetic algorithm is called *Real-Code* because it makes random number choices from a continuous interval. The other common type of genetic algorithm is *Binary-Code*, where an individual in the population is a binary string and the mutation and crossover operators perform binary operations on the individuals. Since the variational method requires Real-valued parameters (it is convention to have all $\alpha_i > 0$) and not boolean parameters, we use a Real-Code genetic algorithm.

The difficult part in using a genetic algorithm is that we could choose any square integrable test function ϕ and get a minimum energy. It can't tell us whether a another test function produces a better upper bound on the minimum energy. Also, when we define the function we also define intervals for the algorithm to pick values for the parameters from. If there is a value outside of our bounds where the approximation has a lower energy, the algorithm will not identify it since it is not in the parameter space the algorithm searches.

To address these difficulties and test the algorithm, we choose a test function with the exact form of the wavefunction solution to a problem. We also choose large parameter intervals. With these two adjustments, the algorithm provides exact energy values for the test, and it explores all of the parameter space.

Table 1: Problem File - Necessary Functions

Function appearing in Problem File	Description
def Func(ind): return (FitnessValue,)	This function takes a list individual and returns a float FitnessValue in a tuple. The tuple return is a DEAP library convention necessary for the algorithm to run. Note that the FitnessValue will be *maximized*. To minimize energy, it should return some positive value, e.g. $10000 - E[\phi]$ if $E \ll 10000$
def ParBounds(ind): return BooleanValue	This function takes a list individual; if it is a valid member of the desired parameter space then it returns True. If the individual is invalid, the function returns False.
def DistOutOfBounds(ind): return Float	This function takes a list individual labeled invalid by ParBounds(ind) and returns a float value representing the 'distance' the individual is from valid parameter space. The algorithm uses this value as an arbitrary weighting function, to determine which invalid individuals are most likely to move back into parameter space.
def plotter(ind): return	This algorithm provides the best individual after all generations. The user can plot a test function and calculate the energy. No return value is expected.

5 Testing the Algorithm

This code works for multiple variational method problems. The base file, *DeapGA.py* takes 4 arguments: the problem, number of generations, population size, and portion of population selected for the next generation. For a particular problem, the problem file (*QD.py*, *Well.py*, *QDFast.py*) requires 4 functions the base file calls during the course of the genetic algorithm (see Table 1). The functions are the fitness function, a parameter bounding function, a parameter correction function, and a plotting function. The bounding function returns a boolean value. It determines if a particular individual is a valid member of the parameter space. For example, the function returns false if the individual has a negative parameter, when we restricted that parameter to be positive. The correction function informs the algorithm how ‘far’ away an invalid individual is from being a valid individual. The plotting function allows a user to output information about the best individual (e.g. a plot of the corresponding minimum energy test function). It is called after the algorithm has evaluated all generations.

The only other necessary part of the problem file is a global variable “NumPar” corresponding to the number of variables in the problem. If NumPar is greater than one, the algorithm allows for crossover of parent individuals. In that case, users must change the *DeapGA.py* file, specifically the fifth and sixth parameters in the function call to algorithms.eaMuPlusLambda. The 5th parameter is the fraction of new individuals produced through crossover and the 6th parameter is the fraction of new individuals generated by mutation of previous indviduals. In a problem with one parameter, any ‘crossover’ results in the copy of a single parent individual, so the default fraction generated by crossover is 0.

5.1 Harmonic Oscillator Test

In an effort to test the algorithm, we run the program with a problem file based on a quantum oscillator potential. The potential is

$$V(x) = \frac{1}{2}m\omega^2 x^2. \tag{2}$$

The test function is

$$\phi(x) = Ae^{-\frac{\alpha}{2}x^2}, \tag{3}$$

where m is the mass of the particle, ω is the angular frequency of the oscillator, and α is the variational parameter. The harmonic oscillator problem has a solution $\psi(x) \propto e^{-\frac{m\omega}{2\hbar}x^2}$. Therefore, the algorithm should return a value of $\alpha = \frac{m\omega}{\hbar}$. This test also offers a good chance to test the scalability of the algorithm, i.e. how its accuracy changes with population size and number of generations used. Instead of trying to evaluate a Gaussian integral of the test function $\phi(x)$ from $-\infty$ to $+\infty$ numerically, the file uses the value of the functional based on the parameter α. Explicitly, this functional is

$$E[\phi] = E(\alpha) = \frac{\hbar^2}{2m}\alpha + \frac{m\omega^2}{4}\alpha^{-1}. \tag{4}$$

Using this expression, the algorithm evaluates the fitness function rapidly. For example, with 50 generations and a 200 individual population, the algorithm reached a best fit individual accurate to 8 significant figures in less than 0.58 seconds (Fig. 3). After various runs like this with different population sizes and generation

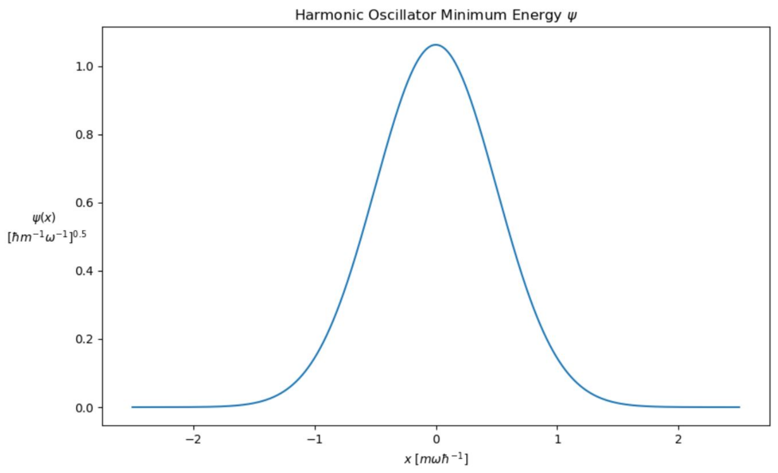

Figure 3: This scaled plot was the best fit test function for the Harmonic Oscillator problem. It matches the analytical soluion to machine precision.

numbers, the most effective way to increase precision is to increase population size. This is because a new generation takes more time to evaluate than more individuals, because the fitness function does not take long to evaluate. If there were a need for numerical integration in the fitness function, we expect more generations would be the effective way to increase precision.

6 Quantum Dot Bound State Problem

Quantum Dots are semiconductors whose band structure is similar to that of atoms. Experimental physicists excite these dots and observe the resulting spectrum to measure the energy differences in the band structure [1]. These dots are approximately 3-dimensional infinite square wells, with an electrostatic potential at their center. This potential is the result of an electronic donor in the dot. Making the approximation that the dot is an infinite well in three dimensions, we calculate the ground energy level of an electron inside the dot. We consider the specific example of a GaAs-(Ga,Al)As quantum dot with a hydrogenic donor. The approximated potential is

$$V(r) = \left\{ \begin{array}{cc} -\frac{e^2}{\varepsilon r} & 0 \le r \le R \\ 0 & r > R \end{array} \right\} \tag{5}$$

where $e = 1.602 \cdot 10^{-19}$ C is the unit of elementary charge, and ε is the dielectric constant of the material in the quantum dot [5]. Using this potential, we write the Hamiltonian in a GaAs-(Ga,Al)As quantum dot as

$$\hat{H} = -\frac{d^2}{dr^2} - \frac{2}{r}\frac{d}{dr} - \frac{2}{r}, \tag{6}$$

where the radius r is in units of the Bohr radius $a_B = \frac{\varepsilon \hbar^2}{me^2} = 9.87\,\text{nm}$ and the Hamiltonian is in units of the Rydberg energy $\text{Ry} = \frac{e^2}{2\varepsilon a_B} = 5.83\,\text{meV}$ (for more detail, see [4]). In those unit calculations, m is the effective mass and $\hbar$ is the reduced Planck constant.

Solving this problem is difficult because there is no explicit wavefunction solution to the Schrodinger equation with the above Hamiltonian. Therefore, we turn to the variational method to calculate the ground state energy. First, we need a test function to use in the method. The function we use, from [5], is

$$\phi(r) = \left\{ \begin{array}{cc} \frac{N}{r}\sin\left(\frac{\pi r}{R}\right)e^{-\lambda r} & 0 \le r \le R \\ 0 & r > R \end{array} \right\}. \tag{7}$$

Using the above test function, we calculate the expectation value of the energy as a functional $E[\phi]$. This functional is dependent on the parameters of the given test function. In this problem, the only parameter is λ, since N is the normalization factor and R is the radius of the quantum dot. Using the symmetric nature of the test function ϕ and Hamiltonian $\hat{H}$, we turn the three dimensional integration into a radial integral:

$$E[\phi] = \frac{\langle\phi|\hat{H}|\phi\rangle}{\langle\phi|\phi\rangle} = \frac{\int_0^R dr\left[r^2\phi^*(r)\hat{H}\phi(r)\right]}{\int_0^R dr\left[r^2 \cdot |\phi(r)|^2\right]} = \frac{-\int_0^R dr\left[r\sin\left(\frac{\pi r}{R}\right)e^{-\lambda r}\left(\frac{d^2\phi}{dr^2} + \frac{2}{r}\frac{d\phi}{dr} + \frac{2\phi}{r}\right)\right]}{N\int_0^R dr\left[\sin^2\left(\frac{\pi r}{R}\right)e^{-2\lambda r}\right]}. \tag{8}$$

Note that the extra normalization factor N will cancel when we evaluate the derivatives of ϕ and substitute them into the expression.

The integration in this functional leaves us with two methods for evaluation. We use a numerical integrator for every individual the algorithm evaluates, or we solve the integrals for arbitrary λ and use the resulting expression. The main difference between these methods is the time it takes the genetic algorithm to run each one. Performing a numerical integration takes more time than calculating an algebraic expression. We use Wolfram Mathematica 11 to solve the integrals for arbitrary λ, and implement this method in the problem file *QDFast.py*. We also write a problem file with numerical integration in the fitness function, *QD.py*, where a Runge-Kutta 4th order integrator is used to evaluate the integrals in Eq. 8.

7 Results

The algorithm took more time to run with the numerical integration file than with the expression from Mathematica. However, the algorithm reached the same minimum energy values with both problem files. The table in this section shows the minimum energy for different values of R and compares them to the results in [5]. There is not exact agreement, but this program was run on a personal computer and not a professional workstation. We expect we could run the algorithm over a limited parameter space and one a more precise computer to achieve higher precision and better agreement.

Table 2: Results from Quantum Dot Problem

Radius $R\,(a_{\mathrm{B}})$	E_{gs} (Ry) from current algorithm	Literature Value of E_{gs} (Ry) [5]
0.1	938.0136	938.004 01
0.3	93.21266	93.189 89
0.5	29.52425	29.497 14
0.8	9.11537	9.087 49
1.0	4.7779	4.748 66
1.2	2.56775	2.539 71
1.4	1.3232	1.294 53
1.8	0.093919	0.065 82
2.0	-0.22160	-0.249 77

8 Conclusions

The algorithm is effective for solving the time-independent Schrodinger equation for an arbitrary potential. We showed this by recreating the results of [5], doing so with a fitness function which performs a numerical integration. Therefore, for problems which have more complicated integrals in the functional, we know the algorithm would be effective in finding the ground state energy. We also observed that when the fitness function contains a single expression, with no numerical integration, the algorithm achieves high precision with increased population size faster than with corresponding increases to number of generations. This is because the algorithm's framework takes longer to execute than the fitness function. However, increasing the number of generations attains a higher precision in less time when the fitness function takes more time to evaluate, e.g. when it involves a numerical integration.

9 Acknowledgments

I would like to thank Dr. Christian Iliadis for helping me complete this project, as well as for coming up with initial project idea and plan. He also provided writing help, by reviewing this paper once before I submitted it for final review.

References

[1] M. Bayer, V. B. Timofeev, F. Faller, T. Gutbrod, and A. Forchel. Direct and indirect excitons in coupled GaAs/Al$_{0.30}$Ga$_{0.70}$As double quantum wells separated by AlAs barriers. *Phys. Rev. B*, 54:8799–8808,

Sep 1996.

[2] I Grigorenko and M.E Garcia. An Evolutionary Algorithm to Calculate the Ground State of a quantum system. *Physica A: Statistical Mechanics and its Applications*, 284(1):131 – 139, 2000.

[3] Patrick Sutton and Sheri Boyden. Genetic algorithms: A general search procedure. *American Journal of Physics*, 62(6):549–552, 1994.

[4] Y.P. Varshni. Accurate wavefunctions for hydrogenic donors in GaAs-(Ga,Al)As quantum dots. *Physics Letters A*, 252(5):248 – 250, 1999.

[5] Hai-Qing Yan, Chen Tang, Ming Liu, and Hao Zhang. Real-code genetic algorithm for ground state energies of hydrogenic donors in GaAs-(Ga,Al)as quantum dots. *Communications in Theoretical Physics*, 44(4):727–730, oct 2005.

10 Appendix A: Main Algorithm Code

```
#DeapGA.py File used for running variational method genetic alg

#Libraries used in code
import random
import numpy as np
import sys
import argparse
from argparse import RawTextHelpFormatter
import importlib

#Using DEAP python Library
from deap import base
from deap import creator
from deap import tools
from deap import algorithms
from time import time

def Main():
    t0 = time()
#Parse arguments for prob, ngen, npop, percKeep
    parser = argparse.ArgumentParser(formatter_class=RawTextHelpFormatter)
    parser.add_argument("prob",type=str,default='QD',
                        help="Which Problem File?\n")
    parser.add_argument("numGens",type=int,default=10,
                        help="How Many Generations to do?\n")
    parser.add_argument("numInds",type=int,default=20,
                        help="How big of a population?\n")
    parser.add_argument("percKeep",type=float,default=10,
                        help="What percentage of the population survives?\n")

    args = parser.parse_args()

#Store input values
    ngen = args.numGens
    npop = args.numInds
    if args.percKeep >=1:
        print("Can't keep >= 100% of the population!")
        return
    nKeep = int(npop*args.percKeep)

#Open problem file and import it
    #NOTE:: different users need to redefine this mypath definition
    mypath = r"/mnt/c/Users/Roark Habegger/OneDrive"
    mypath += r"/Documents/UNC/Senior Year/PHYS 521/GA Project"
    probFile = args.prob
    sys.path.insert(0,mypath)
    p = importlib.import_module(probFile)
```

```
#Get problem variables and functions
    npar = p.NumPar
    E = p.Func
    feasible = p.ParBounds
    dist = p.DistOutOfBounds

#Create framework for toolbox
    creator.create("FitnessMax",base.Fitness,weights = (1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMax)

#create toolbox elements
    toolbox = base.Toolbox()
    toolbox.register("attr_float",random.random)
    toolbox.register("individual", tools.initRepeat, creator.Individual,
                    toolbox.attr_float, n = npar)
    toolbox.register("population", tools.initRepeat, list,toolbox.individual)

    #Only include crossover if 2 parameters to crossover
    if npar>=2:
        toolbox.register("mate", tools.cxTwoPoint)
    toolbox.register("mutate", tools.mutPolynomialBounded, eta = 10, low = 0,
                    up = 1, indpb = 0.4)
    toolbox.register("select", tools.selBest,fit_attr='fitness')
    toolbox.register("evaluate", E)
    toolbox.decorate("evaluate", tools.DeltaPenalty(feasible,0.0,dist))

#define statistics objec
    stats = tools.Statistics(key=lambda ind: ind.fitness.values)
    stats.register("max", np.max)

#What to print at end of generation and record in logbook
    logbook = tools.Logbook()
    logbook.header = "gen", "evals", "fitness"

#Initial population creation
    pop = toolbox.population(n=npop)

#used to return best individual
    hof = tools.HallOfFame(1)

#RUN Mu+Lambda algorithm
    result,log = algorithms.eaMuPlusLambda(pop,toolbox,nKeep,npop-nKeep,
                0.0,1.0 ,ngen, stats = stats, verbose = True, halloffame = hof)
    #Print best individual after ngens
    print(hof)
    print(time()-t0)
#Allow problem to plot/output important statistical info
    p.plotter(hof)

#Run the above function
Main()
```

11 Appendix B: Example Problem File

```
#QD Problem file for varational method genetic algorithm

import numpy as np
import matplotlib.pyplot as plt

#Parameter infor for problem
NumPar = 1

#QD radius
R = 2.0

#search for values between 0 and SearchPar for lambda in wavefunction
SearchPar = 20

#Runga-Kutta 4th order time integrator
def rk4(x0,y0,df,h,params):
    k1 = df(x0, params)*h
    k2 = df(x0+h/2.0, params)*h
    k3 = df(x0+h/2.0, params)*h
    k4 = df(x0+h, params)*h
    return (k1 + 2.*k2 + 2.*k3 + k4)/6.0+y0

#Data processing after algorithm
def plotter(indLst):
    n = 2000
    plt.figure(1)
    rArr = np.linspace(R/(2*n),1.05*R,int(3*n/5))

    #plot all individuals and their minimized fitness
    for ind in indLst:
        par = ind[0]*SearchPar
        plt.plot(rArr, psi(rArr, par))
        plt.axhline(0)
        print(par)
        print(10000-FuncPrecise(ind)[0])
    plt.show()

#Fitness evaluation function
def Func(ind):
    par = ind[0]*SearchPar
    n = 2000
    x0 = R/n
    #Resolve near 0 more than at R
    xArr = np.linspace(x0,R*1.01,n)
    val1 = 0.0
    val2 = 0.0
    #integrate the numerator and denominator
    for i in range(n-1):
        val1 = rk4(xArr[i],val1,f1,xArr[i+1]-xArr[i],par)
        val2 = rk4(xArr[i],val2,f2,xArr[i+1]-xArr[i],par)
        val2 = rk4(xArr[i],val2,f2,xArr[i+1]-xArr[i],par)

    return (10000-val2/val1,)

#Force parameter to be positive
def ParBounds(ind):
    if ind[0] > 0 :
        return True
    return False

#How invalid is the given individual
def DistOutOfBounds(ind):
    return 0.0-ind[0]

#approximation Wavefuntion (the test function)
def psi(r,par):
    #parameters of model
    k = np.pi/R
    l = par
    #Normalization constant
    Norm = np.exp(l*R)*np.sqrt(l*(np.power(k,2)+np.power(l,2)))/np.sqrt(np.pi)
    Norm *= 1/np.sqrt((-1+np.exp(2*l*R))*np.power(k,2)-np.power(l,2)+
            np.power(l,2)*np.cos(2*k*R)-k*l*np.sin(2*k*R))

    #Value of wavefunction form at r
    val = np.sin(k*r)*np.exp(-1*l*r)*np.power(r,-1)

    #Return normalized phi
    return val * Norm

#derivative of wavefunction
def dpsi(r,par):
    #parameters of model
    k = np.pi/R
    l = par
    #Normalization constant
    Norm = np.exp(l*R)*np.sqrt(l*(np.power(k,2)+np.power(l,2)))/np.sqrt(np.pi)
    Norm *= 1/np.sqrt((-1+np.exp(2*l*R))*np.power(k,2)-np.power(l,2)+
            np.power(l,2)*np.cos(2*k*R)-k*l*np.sin(2*k*R))

    #Value of derivative of phi at r
    val = np.cos(k*r)*np.exp(-1*l*r)*k*(np.power(r,-1))
    val -= np.sin(k*r)*np.exp(-1*l*r)*(np.power(r,-2))
    val -= np.sin(k*r)*np.exp(-1*l*r)*l*(np.power(r,-1))

    #return normalized derivative
    return val *Norm

#2nd derivative of wavefunction
def d2psi(r,par):
    #parameters of model
    #parameters of model
    k = np.pi/R
    l = par

    #Normalization constant
    Norm = np.exp(l*R)*np.sqrt(l*(np.power(k,2)+np.power(l,2)))/np.sqrt(np.pi)
    Norm *= 1/np.sqrt((-1+np.exp(2*l*R))*np.power(k,2)-np.power(l,2)+
            np.power(l,2)*np.cos(2*k*R)-k*l*np.sin(2*k*R))

    #Value of 2nd derivative at r
    val  = 2.0*np.sin(k*r)*(np.power(r,-3))
    val += np.power(r,-2)*(2*l*np.sin(k*r)-2*k*np.cos(k*r))
    val += np.power(r,-1)*((l**2)*np.sin(k*r)-2*k*l*np.cos(k*r)-
                           (k**2)*np.sin(k*r))
    val = val*np.exp(-1*l*r)

    #Return normalized value
    return val * Norm

#Magnitude of approximation function (should be 1)
def f1(r,params):
    return np.power(psi(r,params),2)*4*np.pi*np.power(r,2)

#Expectation value of approximation (or test function)
def f2(r,params):
    val = -(d2psi(r,params)*np.power(r,2)+(2*np.power(r,1))*dpsi(r,params))
    val -= 2*np.power(r,1)*psi(r,params)
    val *= psi(r,params)
    val *= 4*np.pi
    return val

#evaluate the energy more precisely
def FuncPrecise(ind):
    par = ind[0]*SearchPar
    n = 10000
    x0 = R/n
    xArr = np.linspace(x0,R,n)
    val1 = 0.0
    val2 = 0.0
    for i in range(n-1):
        val1 = rk4(xArr[i],val1,f1,xArr[i+1]-xArr[i],par)
        val2 = rk4(xArr[i],val2,f2,xArr[i+1]-xArr[i],par)

    return (10000-val2/val1,)
```