Numerical methods: Final project

Šimun Šopar

In final project we were given two tasks that rely on solving equations numerically using two methods: Bisection method and Newton-Raphson method.

First task – quantum particle in a box

First task was to calculate energies of a well known problem, a particle in a box-like potential. The equations for energies are well known and easily derivable, they are:

$$\sqrt{V_0 - E_B}\,tan\left(\sqrt{V_0 - E_B}\right) = \sqrt{E_B}$$
$$\sqrt{V_0 - E_B}\,cot\left(\sqrt{V_0 - E_B}\right) = \sqrt{E_B}$$
$$E_B = -E$$

Here the upper equation is for even wave functions and lower is for odd.

These equations, however, do not have analytical form of solution, they can be solved numerically (or graphically, but numerical solution gives better results). Methods used for solving them are Bisection method and Newton-Raphson method. In this task I calculated the even energies using these two methods.

It's worth noting that the energies E are in an interval $E \in \langle -V_0, 0 \rangle$, since energy cannot be lower than the lowest value of potential, and it cannot be higher than *0* because than the particle wouldn't be bound anymore, meaning all energies would be permitted. This can also be seen from equations, since for E out of that interval yields negative values under the square root, which is physically unacceptable solution.

For however deep or shallow potential, one bound state will always exist. However, we cannot be sure there will be more then one bound state.

Bisection method

For bisection method we need to find two points of opposite signs of function values around the root we are looking for. To do this, first we rewrite the equation in a different form:

$$f(E_B) = \sqrt{V_0 - E_B}\,sin\left(\sqrt{V_0 - E_B}\right) - \sqrt{E_B}\,cos\left(\sqrt{V_0 - E_B}\right) = 0$$

We can now sketch this function's dependence on E_B, this is done in Python code prob1.ipynb. The sketch helps us estimate inital values (guesses) for further calculations in C++.

C++ code is written in file final1.cpp. It contains two functions *even* and *odd* which are functions corresponding to equations for energies of states (written in *sin/cos* form like function $f(E_B)$ above, rather than with *tan* and *cot* to avoid division by zero). We will be using *even* function. The bisection method works the same for *odd* function, just with different initial guesses. The code also contains two functions, *d_even* and *d_odd*, for the derivatives of equations (they can be obtained analytically quite easily). Finally, it contains two functions for calculating the roots of equation, *bisec* and *raphs*.

Bisec takes four parameters. First two are two initial guesses. Next is the precision of our solution. Lastly, it takes the function whose roots we are searching for. The function returns error if values of inital guesses are not of different signs.

The function then calculates new point exactly between two initial guess points. It checks if the value of function at that point is close to zero (we can choose how close to zero we want it to be to be considered as the solution by changing *precision* variable. In my code I went with precision *1E-08*). If the value is aproximately zero, we found our solution. If not, it asigns new guess interval, keeping one inital guess, and changing the one with the same sign as the mid-point to the mid-point, and repeats the process until it finds the value.

Newton-Raphson

Function *raphs* works in a similar fashion to *bisec*. It takes only one initial guess. It also takes precision value, and two functions, one for the equation we are solving, and one for its derivative. It than performs the Newton-Raphson method. It finds a new value of *x* closer to solution then initial guess by linearising the function around initial guess. If the value of function there is aproximately zero (again, aproximately here depends on the value of precision), that is our solution. If not, the process is repeated by making the new value of x our new guess.

Results of computing

The problem was solved for three values of $V_0 = 10, 20, 30$. Figure 1. shows graphs for energy roots. Energies of eigenstates are located where the function depicted in graph is zero.

Each value of V_0 has two eigenstates. For the lowest V_0, the lowest energy is very close to zero. It doesn't look like there is more than one state for this case (function looks strictly positive for the first half of the energy interval), but it can be shown (see Python code), that value of function at zero is negative, so there is another state which could be missed if we would only evalute the problem graphically.

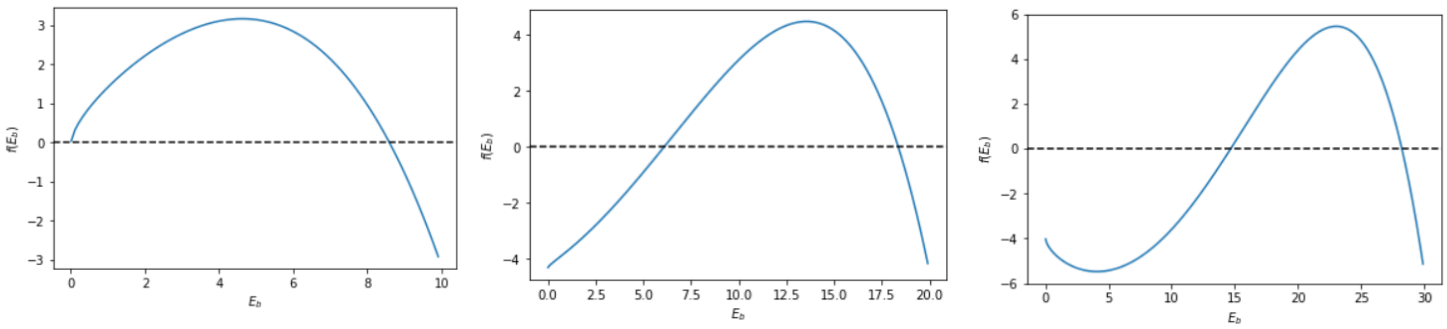

Figure 1. Graphs for locating eigenstate energies for value V0 = 10 (left), 20 (middle) and 30 (right)

Values of energies are given in table below (Figure 2.)

V_0	Energy of first state – bisec	Energy of first state - raphs	Energy of second state - bisec	Energy of second state - raphs
10	0.004019262269	0.004019262453*	8.592785276473	8.592785275230
20	6.108467020094	6.108467017548	18.360519852489	18.360519852467
30	14.666053429246	14.666053424209	28.241113483906	28.241113487946

Figure 2. Values of even state energies for various values of V0. Precision used was 1E-08

V_0	Number of steps for first energy (bisec)	Number of steps for first energy (raphs)	Number of steps for second energy (bisec)	Number of steps for second energy (raphs)
10	28	2*	26	4
20	25	3	26	4
30	25	4	25	3

Figure 3. Number of cycles before reaching the solution

*for energy near zero, if initial guess wasn't very close to exact value (example, initial guess 0.004 would work, but 0.1 wouldn't), the program couldn't compute the problem

The first and second solution E_B grow as V_0 grows. Ground state for $V_0 = 30$ is larger than for $V_0 = 10$, same goes for other states. Also, compared to V_0 = 10, the first state is more noticable. For lower V_0, we can expect only one state. For higher, we can expect more than two. We expect the general behaviour to be the rise of energy values than at some point new first state emerges, making previous first state the new second state, and second the third. We can see that graphically by imagining the function for V_0 = 30 (see Figure 1.) shifted to the right for some L. We now have to fill in the interval between zero and L. We can guess the trend of the function by going from $30 + L$ toward L, it looks like some oscilatory function. We expect the function to continue that behaviour as we approach zero. So for high enough L, the function will cross the x-axis once more, giving us a third energy value (which would now be the lowest, ground state). Then by using even higher V_0, we could get more and more energies. In limit of infinitely deep box, we get infinite energy states.

Bisection vs. Newton-Raphson method

Let's compare results of the two methods and how many times the method had to be executed to get to the result (how many corrections from initial guess were taken). We can see (Figure 3.) that bisection method takes more cycles to get to the solution. Newton-Raphson takes less cycles, however, the initial guess has to be closer. In case the guess is too far, the program will not be able to calculate the solution. For example, for case$V_0 = 30$, the second energy is around *28*. If the guess is *26*, it takes 5 cycles. If the guess is *25*, it will not reach a solution. The solutions given by Newton-Raphson methods are generally more accurate than bisection method (the way I checked this is by solving the problem with WolframAlpha). So the best approach to solving these problems would be to estimate the point graphically, than compute a better estimate using Bisection method than finally use that estimate with Newton-Raphson method to obtain precise results.

Task 2 – Two masses on strings

Second task was to solve a problem of masses on strings. Given two masses and three strings, it is required to find conditions under which static equilibrium is reached. That means we are looking for values of string tension and angles (as seen in Figure 2. of the prof. Kosuke's PDF). This can be done with Newton-Raphson method in higher dimensions. The process is the same as in previous task, however some matrix manipulation is required. Values of tensions and angles are saved in a column vector and the change in that vector during Newton-Raphson method is given by:

$$\Delta \mathbf{x} = -F'^{-1}\mathbf{f}$$

Here, matrix F' and column vector **f** are given with:

$$\boldsymbol{f} = \begin{pmatrix} f_1(\boldsymbol{x}) \\ \vdots \\ f_9(\boldsymbol{x}) \end{pmatrix}$$

$$F' = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_9} \\ \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_2}{\partial x_9} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_9}{\partial x_1} & \cdots & \frac{\partial f_9}{\partial x_9} \end{pmatrix}$$

Where x_i denotes the *i*-th component of our current guess ***x***, and ***f*** is the multidimensional function whose roots we are looking for. As we can see, in calculating change in vector ***x,*** inverse of F' is required, which can be calculated numerically.

The problem is solved in C++ code final2.cpp.

Method of looking for inverse

First step in programming a matrix inverter is to program an algorithm for calculating determinants of square matrices. This is given by function *determinant* in C++ code. It calculates determinants recursively. We know how to calculate determinant of 2x2 matrix. All higher-dimension determinants can be calculated using Laplace expansion and minors. Function *determinant* takes $n \times n$ matrix and expands it over first row. To do this, it calculates corresponding minors by calling itself (recursion). That way we have simplified the problem from $n \times n$ to $n-1 \times n-1$ matrices. If $n-1 > 2$, the process is repeated.

Once we have our algorithm for calculating determinants, we can calculate the inverse using following method:

First we calculate the determinant of matrix *A* we wish to invert.

Next, we define matrix *A'*. Values $A'_{i,j}$ are given as minors, that is we calculate the determinant of a square matrix given when taking out *i-th* row and *j-th* column out of matrix *A*. This matrix is called matrix of minors. Next step is to multiply each $A'_{i,j}$ with $(-1)^{i+j}$, this results in matrix of coefficients. The last step is to take the adjoint of matrix of coefficients and multiply it with one over determinant of initial matrix. This gives the inverse.

Obviously, if the determinant of matrix is zero, it has no inverse. This will be visible in code as dividing by zero, resulting in NaN values of inverse matrix elements.

Function *inverse* finds the inverse of matrix *A* of dimension $n \times n$ and saves it in pointer *Ai*. It relies heavily on previously defined function for calculating determinants.

Function f

Function ***f*** is given as a column vector:

$$\mathbf{f} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \end{pmatrix} = \begin{pmatrix} 3x_4 + 4x_5 + 4x_6 - 8 \\ 3x_1 + 4x_2 - 4x_3 \\ x_7x_1 - x_8x_2 - 10 \\ x_7x_4 - x_8x_5 \\ x_8x_2 + x_9x_3 - 20 \\ x_8x_5 - x_9x_6 \\ x_1^2 + x_4^2 - 1 \\ x_2^2 + x_5^2 - 1 \\ x_3^2 + x_6^2 - 1 \end{pmatrix}$$

We are looking for values of x_i at which ***f*** is a null-vector.

In code, ***f*** is implemented as function *f*, it takes column vector ***x*** (represented with a pointer) and an integer value *n*. For given ***x*** and *n* it will return *n-th* row of ***f***. This is implemented using *switch-case* command.

Derivative of *f* and product of matrices

For Newton-Raphson method to work, it is required to know derivatives of function so we can build our matrix *F'*. This partial derivatives are implemented numerically as forward derivatives with a function *deriv*.

C++ code also implements matrix multiplication with function *prod* which takes three matrices, calculated the product of first two matrices and saves them in third. This function is suitible for calculating product of square $n \times n$ matrix with any $n \times m$ matrix. For our purposes m will be 1, that is a column vector (it could also be *n* if we wish to check that the inverse was correctly calculated, the product of inverse and initial matrix should be unity matrix).

Solution

In my computing I used the following initial guess:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
0.7	0.7	0.7	0.7	0.7	0.7	15	15	15

The result, with precision 1E-06, is:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
0.761002 69	0.264953 81	0.835705 83	0.648748 72	0.964261 10	0.549177 35	17.16020 978	11.54527 968	20.27152 804

Which, in terms of forces and angles, is:

θ_1	θ_2	θ_3	T_1	T_2	T_3
$49°33'9.6''$	$15°21'51.2''$	$56°41'21.9''$	17.16020978	11.54527968	20.27152804

We can check that the solution satisfies the set of equations, I did it in code, but it can also easily be checked with a calculator.

We can see the tension forces are positive, which is the only physically acceptable solution. In magnitude, they are similar to the weights of the masses, the gravitational force acting on masses is 10 for the lighter one and 20 for the heavier. This also makes sense. We can expect the forces to be close to gravitational force since tension and gravitation have to cancel out in static equilibrium. However, the forces are not exactly the same since gravity and tension, due to constraints, do not look in the same direction.

The angles also look like a physically acceptable solution. The first mass is at an angle of $50°$. This is acceptable. The angle is inside the regime that makes physical sense, $\langle 0, 90° \rangle$. Large angles from this regime would mean this mass is significantly heavier than the second mass so the second mass doesn't contribute much to off-set of mass one, it's like mass one sees only the gravity (for angle approaching $90°$). However, the two masses are fairly similar in weight, so the angles are smaller than that. The second angle is $15°$, which is expected. The angle is noticable, but not to large. This is because the second mass is heavier, but not largely heavier than first, so off-set exist, and is noticable, but it is not extreme. Finally, the third angle is $57°$, which is again acceptable. It is larger than the first angle, since the mass is heavier. It is also in the acceptable regime of angles.

It should be noted that the lengths of ropes are not the same, so that also, with the weight of the masses, plays a role in values of the angles.

Different initial conditions

We know from first task that Newton-Raphson method is sensitive to initial guesses. That is why in my first attempt of solving it I used guesses which are physically plausible. Mainly, the tensions are halfway between the gravitational forces, and angles are at about 45° each (45° is good guess for the first and third angle, the second we can expect to be smaller. However, this guess works).

In this section I will try to change initial conditions and see when the program will be unable to find the solution.

First we can temper with tensions and see how the program responds to negative guess of initial tensions. I used -15 instead of 15 as initial guess of tension, while the angles remained the same. The program was still able to get to a correct solution.

Next I checked the program's sensitivity to initial angles. I used 15 as initial guess for tensions, and used 1.5 as guess for sines and cosines (note that this is mathematically imposible since sine and cosine are bound functions). Again the program was able to find the solution. Then I tried using negative values, I tried the program with initial guess -2.5 for sines and cosines. This time the program computed the solution, but it differed from the original solution. Namely, $x_1, x_2, x_3, x_7, x_8, x_9$ had a sign change (their modulo stayed the same). So this initial guess lead to negative tensions and negative angles (if the cosine remains the same and sine changes sign, which happened here, it corresponds to changing the angle from θ to $-\theta$). Although this is the solution to the given set of equations, it is not a physically acceptable solution. This shows how initial guess plays an important role in finding the physical solution.

Finally, let's try to set of initial conditions for which the program cannot find the solution. One such set is obtained by making all the initial guesses of sine and cosine 0 (the initial guess of tension here plays no role). In doing so, the program returns the solution $x_i = -1.\#IND0000$ (on Windows operating system) for each *i*, meaning there is no solution. This happens because the matrix *F'* was non-invertible, it's determinant was zero. The code still tries to invert the matrix and, in doing so, performs division by zero, leading to NaN result.

When choosing initial conditions, we have to be careful not to choose the ones that lead to non-invertible matrix at any step. We cannot know before hand which set of guesses leads to NaN solution, so the only way to check is by trial-and-error. We also have to choose guesses which yield physical results. So the best approach to solving such problems is by trying plausible initial guesses and computing the problem for many initial guesses to get a sense of which guesses work and give good results and which do not. Once the solution is obtained, it can easily be checked with a calculator by plugging in the obtained values, which is also a good way to check that the code is working properly.

Conclusion

We have seen in this two tasks two methods for numerically solving equations or set of equations. In solving sets of equation, Bisection method would prove to be unefficient since we would be dealing with with multivariable functions, which would be impossible to draw (we could draw certain components of function, but that would require much work), so when looking for initial guesses, we would have to temper with initial guesses for quite some time before getting desired results (it is not impossible, just time consuming) so Newton-Raphson method works better here.

Both in solving one equation, or set of equations, initial guesses play important role in obtaining the solution for both methods. So the key to success of these methods is to find a good way of guessing initial values that yield correct results, be it graphically, by looking at physical conditions and constraints or some other methods.

At last, it is important to always check the final solution meets the required conditions, solves the equations and, most importantly, has physical meaning.