Physics 468: Computing Project 6

January 24, 2020

1D Approximate Energies, Numerical Approach to WKB Method

We've been discussing the WKB approximation. This project is to *use* the WKB approach to estimate the energy eigenvalues for a 1D system of your choice. It's a bit of a tricky numerical problem, with lots of pitfalls! To make it more tractable for you, I've provided a series of more or less general purpose utilities that can be used in problems like this. Your job is to *weave* them together to solve the problem at hand. This handout consists mostly of a description of the meaning/purpose of these utilities that may prove useful in completing the project.

Please get the `cp6.py` file and contemplate it while reading this document.

Numerical Integration

The WKB method for bound states focuses on the integral of phase change over one half cycle in the potential.

$$\int_a^b k(x)dx \tag{1}$$

as a result we need a numerical procedure to evaluate this integral. One of the simplest, yet quite effective techniques is Simpson's method (which you've no doubt encountered at some point before?) To review, the idea is to sample the function on a regular grid of points and estimate the integral by replacing with a quadratic interpolation function on each successive set of three points. The polynomial can be integrated exactly, and the result is an approximation to the integral of the original function. It turns out like this:

$$\int_a^b f(x)dx \approx (b-a)\left(f(0)+4f(1)+2f(2)+4f(3)+\ldots+4f(N-1)+f(N)\right)/(3N) \tag{2}$$

One dumb way to do this in python is to use numpy arrays to store the x values, then use the 'stride' feature of array indexing to build the 1-4-2-4...4-1 pattern in the coefficients, like so:

```
def IntegrateSR(f, N, a, b):
    """
    Simple implementation of Simpson's Rule. Use 2N+1 points
    to estimate the integral of f from a to b.
    """
    x=linspace(a,b,2*N+1)
    coefs = ones(2*N+1,x.dtype)  # the end coefficients will be 1.0
    coefs[1::2] = 4.0   # set coefs 1,3,5,.... to 4.0
    coefs[2:-2:2] = 2.0 # set coefs 2,4,6,...  to 2.0
    return (f(x)*coefs).sum()*(b-a)/(3.0*2*N)
```

Root Finding

There are two different root finding functions included the first is Newton's Method (which we've seen before so no need to review in detail):

```
def NewtonsMethod(f, fp=None, x0=0.0, eps=1.0e-6, bounds=[-1e9,1e9]):
    """
    Find a zero of the function f near x0 to within eps.
    If fp is provided, it is a python function that returns the derivative of f.
    If fp is not provided.. please create a function to evaluate the derivative
     numerically
    """
    if fp is None:
        fp = lambda x: NumericalDerivative(f, x, eps)

    done = False
    x = x0
    fv = f(x0)

    while abs(fv)>eps and x>bounds[0] and x<bounds[1]:
        fpv = fp(x)
        x += -fv/fpv
        fv = f(x)

    return x
```

The next method uses the 'Binary Search' approach. This particular version assumes

that the function is a monotonically increasing function of its argument. The intent here is to use it to find the value of energy that produces a particular change in phase. Since phase *is* a monotonically increasing function of E, this should work!

```
def MonotonicBinarySearch(f, xmin, xmax, eps):
    """
    Look for a zero in a monotonically increasing function f between xmin and xmax.
    return the value of x for which f evaluates within eps of zero.
    """
    maxF = f(xmax)
    minF = f(xmin)

    if (minF > 0) or (maxF < 0):
        raise ValueError, (MBSMsg % (minF, maxF))

    x = (xmin + xmax)/2.0
    fv = f(x)

    while abs(fv)>eps:
        print xmax, xmin, fv
        if fv >= 0:
            xmax = x  # we're too high
        else:
            xmin = x  # too low

        x = (xmin+xmax)/2.0  # reset x to the middle
        fv = f(x)

    return x
```

Note that the idea is begin with markers at xmin and xmax. As the search progresses the markers are moved to the middle on each iteration halving the distance between them. If the function is "well behaved" this process should converge fairly quickly.

Finding the Net Phase

The actual integral of Eq. 1 needs to be evaluated many times to find the energy that produces a particular total phase change. This needs to be encapsulated in a function that can be called repeatedly by a root finder. Notice that this function calls another utility to find the turning points. We'll get to that one in a bit, but see that the current energy and the potential function need to be passed in as well as the place to start looking, and the step size to use in the search have to be provided.

```
def NetPhase(E, Vfunc, xstart=0.0, N=1000, VfuncPrime=None):
```

```
    """
    compute the net phase integral from a to b in "N" samples.
    (actually we leave the actual number of samples up to the numerical
    integration implementation).
    xstart is where to begin looking for turning points"
    N is the number of points to use in numerical integration
    findLowTP is a function that returns the low turning point.
    """

    a = FindTurningPoint(E, Vfunc, xstart, -0.1, VfuncPrime)
    b = FindTurningPoint(E, Vfunc, xstart, 0.1, VfuncPrime)

    return IntegrateSR(lambda x: sqrt(2.0*m*(abs(E-Vfunc(x))))/hbar,
                       N, a, b) # integrate k(x)*dx from a to b
```

Getting the Turning Points

This is really crude... but it works! Of course you could improve upon it easily I'm sure. The motivation was to have a fail proof method that doesn't require you to specify *bounds* for the search. It can/will get into trouble if there *is no* turning point however. The idea is to do a linear search with step size `dx` until a sign change is found, then use Newton's Method to zero in on the actual turning point. If you pass in a negative dx, the search will proceed to the left, otherwise it will proceed to the right. A smaller `dx` will get you closer, but take more steps. Since Newton's Method converges quickly if you're fairly close to the turning point, a very small dx is not really needed. Of course you can imagine lots of pathological functions that would cause trouble for this simple minded approach. Luckily, most physically interesting potentials aren't that pathological (but beware of "square" wells of any kind.. they don't suffer Newton's Method kindly!).

```
def FindTurningPoint(E, Vfunc, x0, dx=0.1, VfuncPrime=None, xmin=-100, xmax=+100):
    """
    Find the turning point of the potential.

    First do a linear search for the sign change with low resolution dx.

    Then do a Newton's Method search for a more exact value.
    xmin: The lowest reasonable value of x possible
    xmax: The highest reasonable value of x possible
    """
    x = x0
    s = sign(E - Vfunc(x))
    while sign(E - Vfunc(x)) == s and x>xmin and x<max:
        x += dx
```

```
    if (x <= xmin) or (x >= xmax):
        raise ValueError, (FindTPMsg % x)
    #
    # We're close! Now do Newton's method to zero in on exact value
    #

    return NewtonsMethod(lambda x: Vfunc(x)-E, VfuncPrime, x)
```

Anonymous Functions

One issue that is often encountered when attempting to use root finders, integrators and other "standardized" solvers is that the problem they are designed to solve (e.g., root finding, or integration) is not exactly the problem we are interested in solving (e.g., turning point determination, or phase determination respectively). Sometimes there is a trivial mapping of one to the other, but sometimes it's more complicated. We see both situations in this project. For example notice that when we need to find the turning point in the `FindTurningPoint` function, we call the `NewtonsMethod` function with a weird argument: `lambda x: Vfunc(x)-E`. This is an *anonymous* function. In fact it's exactly equivalent to:

```
def f(x):
    return VFunc(x)-E
```

but we didn't actually need to define a function to do it! Generally the rule(s) of thumb are:

1. If you only use the function to pass as an argument to another function, or as an element in a data structure
2. OR if you never need to call the function "by name"
3. AND if the function is simple enough that it can be done as a "one liner".

So, what are we supposed to do?

Pick a 1D potential (you might consider one of the homework problem potentials provided by Griffiths). Use the WKB method to estimate the lowest 5 eigenstate energies *numerically*. It would be nice to use a potential that has exact solutions so you can compare with known exact results. You are free to use the utilities provided in `cp6.py` in any way you like to achieve this. If you wish to, you may improve upon these to make your code faster or more accurate, but you needn't if that's not personally interesting for you.

Be sure to explain carefully and completely how you choose the phases associated with your estimated eigenstate energies.

You can check that your approach is working by computing the Energy vs. Phase graph for the given S.H.O. potential:

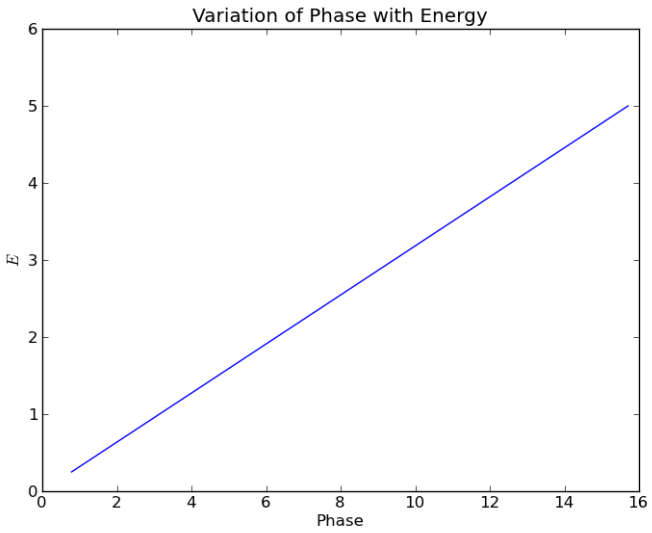

Figure 1: Energy vs. Phase for the S.H.O. potential

Questions

Please answer these questions at the end of your report.

1) Explain how the anonymous function `lambda x: Vfunc(x)-E` converts the turning point problem into the root finding problem.

2) Why couldn't we use an anonymous function to call `IntegrateSR` directly rather then creating a custom function `NetPhase`?

3) How is using the WKB approach similar and/or better/worse than simply integrating the TISE numerically?