Mathematica Notebook for Finding Critical Values of λ

Here is my actual code that I actually ran to get my thesis data. For easier data collection, this was run as a loop once all the bugs were worked out. I present here the expanded version, which can be compressed and looped if desired.

Finding candidate ranges

- **Setup**

```
l = 3; (*Angular momentum quantum number*)

RoughCritical[λ_] := Module[{H, res, dρ, ρinf, nn},
  dρ = 0.1;
  ρinf = Round[20 * λ];
  nn = Round[ρinf / dρ];
  H = SparseArray[{{n_, n_} → 2. / dρ^2 - 2. / (n * dρ) e^(-n*dρ/λ) + l * (l + 1) / (n * dρ) ^ 2,
     {n_, m_} /; Abs[n - m] == 1 → -1. / dρ^2}, {nn, nn}, 0.];
  res = Sort[Eigenvalues[H, -100]];
  Return[res]
 ]
```

This function takes a value of lambda and returns the 100 eigenvalues of the hamiltonian which have the lowest absolute value. nn is the matrix size, dρ is the step size, and ρinf is numerical infinity for the problem.

The results of this calculation are extremely sensitive to ρinf, and a larger ρinf makes for less error. However, for larger values of ρinf, more *positive* eigenvalues are found near 0, because the positive eigenvalues are in a continuous spectrum. As the grid becomes a better approximation of reality, more of these are picked up. This squeezes the negative eigenvalues (whose spectrum starts near -1) out, especially at higher values of λ. In order to ensure that there are *some* negative eigenvalues, we therefore pin ρinf to some constant factor times λ.

```
CountNegs[list_] := Module[{neg, i},
  neg = 0;
  For[i = 1, list[[i]] < 0, i++,
   neg++;
  ];
  Return[neg]
 ]
```

```
FindLs[l0_, lf_, val_, vaf_, range_] := Module[{mid, vam, lnegs, mnegs, hnegs, res},
  res = {};
          lf - l0
  mid = l0 + ------- ;
            2
  lnegs = CountNegs[val];
  vam = RoughCritical[mid];
  mnegs = CountNegs[vam];
  hnegs = CountNegs[vaf];
  If[lf - l0 > range,
   If[lnegs < mnegs || (lnegs == mnegs && val[[1]] < vam[[1]] && mnegs > 0),
    res = Join[res, FindLs[l0, mid, val, vam, range]];
    Return[res]
   ];
   If[mnegs < hnegs || (hnegs == mnegs && vam[[1]] < vaf[[1]] && hnegs > 0),
    res = Join[res, FindLs[mid, lf, vam, vaf, range]];
    Return[res]
   ];
   ,
   Print["Found one!"]; (*this lets you know it's working*)
   Return[{{l0, lf}}];
  ]
 ]
```

This is my multiple rootfinder, to find ranges where a zero occurs. Again, when we bisect, we are looking at the number of negative eigenvalues as a function of lambda. In this routine, we assume multiple zeroes, and keep bisecting any range with a zero in it.

The complicated conditional for bisection is meant to deal with the problem of negative eigenvalues getting squeezed out by positive ones. Given a half of the input range, it bisects that half if either the number of negative eigenvalues increases (a regular zero), or the number of negative eigenvalues stays constant, but the value of the first negative eigenvalue jumps. This indicates that one negative eigenvalue was squeezed out, but another was added. In general, as λ increases, the negative eigenvalues should decrease in value (coming closer and closer to the Coulomb energies).

```
RangeChunk[low_, high_] := Module[{res, val, vaf, lnegs, hnegs},
  res = {};
  val = RoughCritical[low];
  lnegs = CountNegs[val];
  vaf = RoughCritical[high];
  hnegs = CountNegs[vaf];
  If[lnegs < hnegs || (lnegs == hnegs && val[[1]] < vaf[[1]] && hnegs > 0),
   res = FindLs[low, high, val, vaf, 1];
  ];
  Return[res]
 ]
```

■ Calculation

Here the parallelization starts. All of those calls to Eigenvalues aren't cheap, and the range-finding can be paralellized as well as the actual bisection calculation. RangeChunk, above, is just a wrapper for the function that finds ranges. It does the initial calculation and makes sure there's a zero inside the the range before sending it off to be bisected.

```
NoKernels = 21; (*Number of kernels in the Lightweight Grid*)

int = (5. * 10^5 - 0.25 / 2) / NoKernels

chunks = {{0.5, Sqrt[2 * int + 0.25]}}
For[i = 1, i < NoKernels, i++,
 chunks = Append[chunks, {chunks[[i, 2]], Sqrt[chunks[[i, 2]]^2 + 2 * int]}]
]
```

The range -- $\lambda = 0.5$ to 1000 -- gets divided up based on the length of time it takes to run Eigenvalues there. The timing of Eigenvalues is roughly linear with ρinf. The amount of time it takes to find a range will therefore be approximately

the integral of this line over the range.

```
DistributeDefinitions[RangeChunk, CountNegs, chunks, FindLs, RoughCritical, l]
(*Every kernel needs all of the functions and lists being used*)
```

```
rangesrough =
  ParallelTable[RangeChunk[chunks[[i, 1]], chunks[[i, 2]]], {i, 1, Length[chunks]}];
```

This comes out as a list of lists of pairs, and we just want a list of pairs. So we massage it a little.

```
ranges = {};
```

```
For[i = 1, i < Length[rangesrough], i++,
 ranges = Join[ranges, rangesrough[[i]]];
]
```

Rootfinding for Lcs

Setup

```
Critical[λ_, consts_] := Module[{H, res, dρ, ρinf, nn},
  dρ = consts[[1]];
  ρinf = Round[consts[[2]]];
  nn = Round[ρinf / dρ];
  H = SparseArray[{{n_, n_} → 2. / dρ^2 - 2. / (n * dρ) e^(-n*dρ/λ) + l (l + 1) / (n * dρ) ^ 2,
     {n_, m_} /; Abs[n - m] == 1 → -1. / dρ^2}, {nn, nn}, 0.];
  res = Sort[Eigenvalues[H, -25]]; (*Went from 100 down to 25 eigenvalues,
  which sped this calculation up immensely with no apparent loss of information*)
  Return[res]
 ]
```

This is another version of the RoughCritical function above that only takes the 25 eigenvalues closest to 0. It also allows dρ and ρinf to be varied. Decreasing the number of eigenvalues taken sped up the calculation a lot, and if on every range we have 0 eigenvalues at the lower end and 1 on the upper end, that's no big deal -- that's a zero.

```
Bisect[F_, consts_, x0_, xf_, ϵ_] := Module[{mid, val, lnegs, vam, mnegs, vah},
  mid = x0 + Abs[x0 - xf] / 2;
  Print["Bisecting. Midpoint is " <> ToString[mid]];
  (*Again, lets you know it's working/progress*)
  val = F[x0, consts];
  lnegs = CountNegs[val];
  vam = F[mid, consts];
  mnegs = CountNegs[vam];
  If[xf - x0 > ϵ,
   If[lnegs < mnegs || (lnegs == mnegs && val[[1]] < vam[[1]]),
     (*if there is a zero on the lower half*)
     Return[Bisect[F, consts, x0, mid, ϵ]];, (*bisect it*)
     Return[Bisect[F, consts, mid, xf, ϵ]]; (*otherwise try the upper half*)
    ];,
   vah = F[xf, consts];
   Return[{mid, lnegs, CountNegs[vah]}];
  ]
 ]
```

Bisection rootfinder, finding where a new negative eigenvalue appears. It returns not only the midpoint, but the number of negative eigenvalues at the low and high ends of the final range. This allows for a quick eyeball check if

necessary to verify that there is in fact a zero where the function says there is.

```
Findlc[input_, i_, eps_] := Module[{val, vah, lnegs, hnegs, res},
  Print["***Beginning to calculate the " <>
    ToString[i] <> "th lambda, part " <> ToString[input[[1]]]];
  val = Critical[input[[3]], {0.1, input[[2]], input[[2]] / 0.1}];
  vah = Critical[input[[4]], {0.1, input[[2]], input[[2]] / 0.1}];
  lnegs = CountNegs[val];
  hnegs = CountNegs[vah];
  If[lnegs < hnegs || (lnegs == hnegs && val[[1]] < vah[[1]]), (*doublechecks for zero*)
   Print["Pass!"];
   res = {input[[1]],
     Bisect[Critical, {0.1, input[[2]], input[[2]] / 0.1}, input[[3]], input[[4]], eps]};
   , (*else*)
   Print["Fail!"];
   res = {input[[1]], Null};
  ];
  Return[res]
 ]
```

This is the wrapper for parallel bisection. It makes sure that there is in fact a zero in the range, because the ρinf dependence means that sometimes the critical λ drifts out of the range. Any bisections done where there is no zero in the range would return the upper endpoint and be useless.

```
ListRandomize[list_] := Module[{order, res},
  order = Table[{RandomReal[], i}, {i, 1, Length[list]}];
  order = Sort[order];
  res = Table[list[[order[[i, 2]]]], {i, 1, Length[list]}];
  Return[res]
 ]
```

This is a quick-and-dirty optimization for the parallel rootfinding. All it does is mix up the inputs so a kernel is less likely to get two long or two short calculations.

```
rhoinfs = {40, 80}; (*Scaling factors for λ*)
```

■ First calculations: refinement

To reduce the number of failed rootfinding attempts, we refined the rough ranges above a bit more at different values of ρinf. The first gets down to a range of width 1/4, the second, a range of width 1/32.

```
inputslow = Table[{i, rhoinfs[[1]] * ranges[[i, 1]],
    ranges[[i, 1]] - 2, ranges[[i, 2]] + 2}, {i, 1, Length[ranges]}];
```

In order to deal with randomization later, the inputs are tagged with a number so they can be sorted again. This number is retained by Findlc.

```
inputslow = ListRandomize[inputslow];

DistributeDefinitions[Critical, Bisect, CountNegs, Findlc, inputslow, l]

refinement = ParallelTable[Findlc[inputslow[[i]], i, 0.25], {i, 1, Length[inputslow]}]

refinement = Sort[refinement];

inputshigh = Table[{i, rhoinfs[[2]] * refinement[[i, 2, 1]],
    refinement[[i, 2, 1]] - 0.75, refinement[[i, 2, 1]] + 0.75}, {i, 1, Length[refinement]}];

inputshigh = ListRandomize[inputshigh]

DistributeDefinitions[inputshigh]

refinement2 = ParallelTable[Findlc[inputshigh[[i]], i, 1 / 32.], {i, 1, Length[inputshigh]}]

refinement2 = Sort[refinement2];
```

Second calculation: lcs for real

```
ref1ins = Table[{i, rhoinfs[[1]] * refinement[[i, 2, 1]], refinement[[i, 2, 1]] - 1 / 4.,
    refinement[[i, 2, 1]] + 1 / 4.}, {i, 1, Length[refinement]}];

ref2ins =
  Table[{i + Length[ref1ins], rhoinfs[[2]] * refinement2[[i, 2, 1]], refinement2[[i, 2, 1]] -
     1 / 64., refinement2[[i, 2, 1]] + 1 / 64.}, {i, 1, Length[refinement2]}];
```

And now we combine the refined ranges, make input lists out of them, and find the values of the λcs to within 10^{-6}.

```
newins = Join[Sort[ref1ins], Sort[ref2ins]];

newins = ListRandomize[newins];

DistributeDefinitions[newins]

lcs = ParallelTable[Findlc[newins[[i]], i, 10^-6], {i, 1, Length[newins]}]

lcs = Sort[lcs]
```

Fitting and Data Analysis

```
data = Table[
   {lcs[[i, 2, 1]], lcs[[i, 2, 1]] - lcs[[i + Length[lcs] / 2, 2, 1]]}, {i, 1, Length[lcs] / 2}];
```

This bit exports the data to a file for further analysis. The directory should be set to wherever this notebook is running.

```
SetDirectory["~/Desktop/Ellen/Thesis/Calculations/Angular Momentum/"]

Export["l" <> ToString[l] <> ".csv", data]

Needs["ErrorBarPlots`"];

dataplot = ErrorBarPlots`ErrorListPlot[data]
```

Quadratic fit

```
fit = FindFit[Table[data[[i, 1]], {i, 1, Length[data]}], a x^2 + b x + c, {a, b, c}, x]
```

```
{a → -0.0220443, b → 48.2671, c → -36.59}
```

```
F[x_] := a x^2 + b x + c

fitplot = Plot[F[x] /. fit, {x, 0, 35}]

Show[dataplot, fitplot]
```