Appendix A

Numerical Methods

A.1 Rootfinder

To actually locate the critical values of λ, we need to find the place where a new bound state appears. Numerically, this looks like a new negative eigenvalue in our list of eigenvalues, which suggests we approach the problem with rootfinding. Our function has a "zero" whenever the number of negative eigenvalues increases.

Here it is in *Mathematica* code:

```
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]];
    Return[res]
  ]
```

This function takes a value of lambda and returns some number of eigenvalues (here, 25) 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.

Increasing the number of negative eigenvalues found will also ensure we don't miss negative eigenvalues, but is very expensive. When searching for initial eigenvalues,

we chose to get the 100 with the lowest absolute value in order to increase accuracy, but for the main calculation, 25 was fine.

The rootfinder actually used was a recursive bisection routine. In this rootfinder, we stopped recursing when the difference between the two endpoints was less than some δ, instead of when the values of the function at the endpoints were within ϵ. Because our function just looks at the number of negative eigenvalues, $f(\lambda_f) - f(\lambda_0)$ should always be equal to 1 in a range with a zero and ϵ is meaningless.

In *Mathematica* code:

```
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]]; (*progress indicator*)
    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]]),
            Return[Bisect[F, consts, x0, mid, ϵ]];,
            Return[Bisect[F, consts, mid, xf, ϵ]];
     ];,
   vah = F[xf, consts];
   Return[mid, lnegs, CountNegs[vah]];
   ]
]
```

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 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).

To find the ranges to bisect, we modify the rootfinder so it keeps bisecting as long as there is a zero in one of the ranges, and returns a list of points instead of just one. For efficiency's sake, this calculation is run at a low ρ_∞, and a range is returned instead of a single point. That range can then be bisected more thoroughly in order to identify the most likely point.

As is clear from the code, the function doing the actual work is just Mathematica's Eigenvalues[ ]. Because our Hamiltonian matrix only has entries on the diagonal and off-diagonals, we can define it as a SparseArray to speed up calculation.

Finding a single λ_n takes quite a bit of time – it's expensive to find the eigenvalues of a large matrix, even such a sparse one as ours. We would like to study a wide

range of λs, and the computation begins to look prohibitively expensive. However, the problem is embarrassingly parallel. Each bisection to find a value of λ_n is independent of all the others, allowing us to easily split this problem into chunks. We used Mathematica's Lightweight Grid to run the computation on a cluster of machines. Distributing these calculations among multiple cores speeds the process up by about 50% per core added.

A.2 Fits

A.2.1 Linear

Our weighted linear least-squares fit was computed following the methods outlined in Taylor [6]. We first compute weights for each point, based on its error, then use that to weight the least-squares calculation.

Here it is in Mathematica code:

```
weights = Table[1/(logerrs[[i]]^2), {i, 1, Length[logerrs]}];

x[i_] := loglog[[i, 1]];
y[i_] := loglog[[i, 2]];
w[i_] := weights[[i]];
len = Length[loglog]

Delta = Sum[w[[i]], {i, len}]*Sum[w[[i]]*x[i]^2, {i, len}] -
        (Sum[w[[i]]*x[i], {i, len}])^2;

A = 1/Delta*(
        Sum[w[i]*x[i]^2, {i, len}] *Sum[w[i]*y[i], {i, len}] -
        Sum[w[i]*x[i], {i, len}]*Sum[w[i]*x[i]*y[i], {i, len}])

B = 1/Delta*(
        Sum[w[i], {i, len}]*Sum[w[i]*x[i]*y[i], {i, len}] -
        Sum[w[i]*x[i], {i, len}]*Sum[w[i]*y[i], {i, len}])
```

A.2.2 Quadratic

For our quadratic fit, we first decided to ignore data weights as the errors were largely on similar orders of magnitude. Instead of calculating the coefficients of the fit independently, we did it with a matrix inversion, because the expressions for each component are complicated. More precisely, we solved the equation

$$\begin{pmatrix} N & \sum n_i & \sum n_i^2 \\ \sum n_i & \sum n_i^2 & \sum n_i^3 \\ \sum n_i^2 & \sum n_i^3 & \sum n_i^4 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} \sum \lambda_i \\ \sum n_i \lambda_i \\ \sum n_i^2 \lambda_i \end{pmatrix}. \tag{A.1}$$

for A, B, and C. For the matrix inversion, we just used the one built into Mathematica.

When we turned this into a weighted fit, we picked up some extra $\Delta\lambda_i$ terms:

$$\begin{pmatrix} \sum \frac{1}{\Delta\lambda_i^2} & \sum \frac{n_i}{\Delta\lambda_i^2} & \sum \frac{n_i^2}{\Delta\lambda_i^2} \\ \sum \frac{n_i}{\Delta\lambda_i^2} & \sum \frac{n_i^2}{\Delta\lambda_i^2} & \sum \frac{n_i^3}{\Delta\lambda_i^2} \\ \sum \frac{n_i^2}{\Delta\lambda_i^2} & \sum \frac{n_i^3}{\Delta\lambda_i^2} & \sum \frac{n_i^4}{\Delta\lambda_i^2} \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} \sum \frac{\lambda_i}{\Delta\lambda_i^2} \\ \sum \frac{n_i\lambda_i}{\Delta\lambda_i^2} \\ \sum \frac{n_i^2\lambda_i}{\Delta\lambda_i^2} \end{pmatrix}. \tag{A.2}$$