The Variational Method for Non-Hermitian Quantum Mechanics
Automatic Differentiation with Complex Numbers

Daniel Kraft

August 8th, 2013

Abstract

The framework of non-Hermitian quantum mechanics can be used to describe and calculate complex resonance energies corresponding to poles of the S-matrix in scattering problems. When a variational method is employed to solve the resulting complex eigenvalue problem, the criterion of minimising the energy expectation can no longer be applied because of the complex energies—instead the principle becomes a stationary one, where the gradient of the energy expectation value with respect to all variational parameters should in theory vanish and in practice be made as small as possible. I will describe how automatic differentiation based on operator overloading in C++ with the ADOL-C package can be used for this task. The main difficulty here lies in the fact that complex numbers must be handled, which work not entirely out-of-the-box with ADOL-C. With some understanding and tricks it can be made to work, though. For this particular problem, the derivatives can also be calculated symbolically, and it turns out that both methods yield the same results. With automatic differentiation one can then calculate also second derivatives which become very unwieldy to handle manually.

1 The Problem

1.1 Introduction to Quantum Mechanics

The usual framework of quantum mechanics is based on functional analysis in Hilbert spaces (based on $L^2(\mathbb{R}^n)$ over the complex numbers as scalars), see also [8] for a general introduction to this framework. States of a quantum system are represented by (normalised) vectors, and measurable observables by self-adjoint or at least Hermitian operators. When a measurement is performed, the measured value will be an eigenvalue of the operator—thus Hermiticity guarantees that all outcomes of measurements are real numbers as one expects for physical quantities. In the so-called "Schrödinger picture" of quantum mechanics, the time-evolution of a system is described by states changing over time according to the famous *Schrödinger equation*:

$$i\hbar \frac{d}{dt}\psi = \hat{H}\psi \tag{1}$$

Here, $\hat{H}$ is a special Hermitian operator corresponding to the *total energy* of the system (the "Hamiltonian operator"). Usually, it is made up of two terms: One for the kinetic energy which always has the same form and one for the potential energy which depends on the system (or external potential) one is interested in. Thus, one has

$$\hat{H} = -\frac{\hbar^2}{2m}\Delta + V, \tag{2}$$

where $V : \mathbb{R}^n \to \mathbb{R}$ is simply a multiplication operator describing the potential. In the following, I will always assume that $\hbar = m = 1$ to simplify the equations further. This is in line with the common practice of choosing "natural units" for quantities and doesn't restrict the underlying theory in any way.

The time-evolution of (1) can be expressed in an abstract way via the exponential of the operator in the differential equation, $e^{-\frac{i}{\hbar}\hat{H}t}$, which is often called the "time-evolution operator". It is unitary for self-adjoint $\hat{H}$, preserving the normalisation property of states. From this representation and also the correlation between eigenvalues and measurements it follows that the spectrum of $\hat{H}$ is of particular importance. In fact, via appropriate ansatz functions the solution to (1) can be expressed analytically

in terms of the eigenfunctions of $\hat{H}$. They correspond to "bound states" of the problem. Thus one commonly investigates a quantum mechanical problem by solving the *time-independent* Schrödinger equation instead:

$$\hat{H}\psi = E\psi, \tag{3}$$

which is nothing else but the eigenvalue equation of $\hat{H}$ for eigenstates ψ and corresponding eigenenergies E. (3) is well-researched for a wide class of interesting systems.

1.2 Resonances and Complex Scaling

In contrast to "ordinary" quantum mechanics as described above, certain phenomena (resonances or "instable states") can be related to *complex* energy-eigenvalues of $\hat{H}$. Roughly speaking, the real part of such an eigenvalue is the energy at which the resonance state appears in experiments, and the imaginary part is related to its average life-time. The closer to the real axis the value lies, the longer this time is (with real energies corresponding to perfectly stable states as in the classical framework) and the sharper such a resonance appears in scattering experiments. Resonances can be investigated from classical scattering theory (see for instance [9, ch. 13, p. 238ff]) or by means of time-dependent calculations, but a different and mathematically elegant approach is the use of *non-Hermitian* operators to solve (3) for complex energies. See [6] for more details about this approach.

One may ask how complex energies can actually arise for an operator as in (2), which is Hermitian (at least as long as V is real). To answer this, note that an operator like $\frac{d^2}{dx^2}$ is only Hermitian ("symmetric") as long as the boundary terms that appear from integration by parts in the L^2-inner product vanish. For functions in L^2, this is the case—and there such a Hamiltonian is Hermitian. However, the eigenfunctions corresponding to resonances *do not lie in* L^2, instead they are only *locally* in L^2: From the general scattering theory it is expected that those functions behave like a plane wave

$$\psi(x) \sim e^{ikx}$$

for $x \to \infty$ far away from the scattering centre. If the energy, which is related to the wave-number k via $E \sim k^2$, is now complex such that $\mathrm{Im}\,(k) < 0$, then $|\psi(x)| \to \infty$ for $x \to \infty$, so that surely $\psi \notin L^2(\mathbb{R})$.

This also prevents us from applying methods based on the Hilbert space semantics of L^2 directly to calculate those eigenvalues, because they are "invisible" from within L^2. To resolve this problem, one can apply *complex scaling* [6, sec. 5.2, p. 120ff]: For suitable functions (which can be analytically continued onto the complex plane) define

$$(\hat{S}_\theta\psi)(x) = \psi\left(e^{i\theta}x\right).$$

This is a linear and invertible map, with the inverse given as $\hat{S}_\theta^{-1} = \hat{S}_{-\theta}$. Consequently, eigenvalues are preserved when a similarity transformation based on $\hat{S}_\theta$ is applied to $\hat{H}$,

$$\hat{H}_\theta = \hat{S}_\theta \hat{H} \hat{S}_{-\theta}. \tag{4}$$

However, the eigen*functions* are changed by this procedure, and when $\hat{H}\psi = E\psi$ was the case before scaling, now $\hat{H}_\theta\psi_\theta = E\psi_\theta$ where $\psi_\theta = \hat{S}_\theta\psi$. Note also that for the asymptotic behaviour of the *scaled* resonance eigenfunction we now have

$$\psi_\theta(x) \sim \exp\left(ie^{i\theta}kx\right) \to 0$$

for $x \to \infty$ as long as $\mathrm{Im}\left(e^{i\theta}k\right) > 0$. In other words, by the transformation (4) we get a modified operator $\hat{H}_\theta$ which still has the same complex eigenvalues we're interested in but whose eigenfunctions are now again in L^2, as long as the "scaling angle" θ is large enough (larger than the argument of k to be precise). It is easy to see that for a Hamiltonian according to (2) the scaled version is

$$\hat{H}_\theta = -\frac{1}{2}e^{-2i\theta}\Delta + V\left(e^{i\theta}x\right).$$

This is still a "symmetric" but now also genuinely complex operator, thus no longer Hermitian (which allows it to have complex eigenvalues even inside the Hilbert space L^2 we work in).

1.3 The Variational Method

For *any* state ψ, which needs not be an eigenstate, one can calculate the expectation value of the Hamiltonian (the "mean energy" $\overline{E}$) via the so-called *Rayleigh quotient*:

$$\overline{E} = R(\psi) = \frac{\left\langle \psi|\hat{H}|\psi \right\rangle}{\langle \psi|\psi \rangle} \tag{5}$$

It is clear that if (3) is fulfilled, then $\overline{E} = E$ and thus $R(\psi)$ gives just the energy level for every eigenstate. The *Rayleigh-Ritz variational principle* states now that if E_0 is the lowest energy level of $\hat{H}$, then $R(\psi) \geq E_0$ for all possible (also non-eigen)states ψ. This can be seen easily because for a Hermitian Hamiltonian the eigenstates usually form an orthonormal basis in which ψ can be expanded:

$$\psi = \sum_i c_i \psi_i \quad \Rightarrow \quad \left\langle \psi|\hat{H}|\psi \right\rangle = \sum_i |c_i|^2 E_i \geq E_0 \cdot \sum_i |c_i|^2 = E_0 \langle \psi|\psi \rangle$$

Thus, at least to calculate E_0, a common method in quantum mechanics is to choose some ansatz for $\psi(\alpha_i)$ containing a number of parameters α_i and then simply minimising $R(\psi(\alpha_i))$ over the set of possible parameter values. This gives good approximations to the ground state energy E_0 (and an exact upper bound) if the set of possible functions $\psi(\alpha_i)$ is rich enough. This is described in [3, ch. 43, p. 302ff] and many other introductory texts about quantum mechanics.

For our special *non-Hermitian* setting however, we can't apply this method directly because for obvious reasons with complex energies, there's no longer any way to decide which set of parameters is "better" based on "minimising" something. However, one can find a remedy to this problem. First, we have to define a new "inner product" termed the *c-product* by Moiseyev, see [6, ch. 6, p. 174ff]:

$$(\phi|\psi) = \int_{\mathbb{R}^n} \phi(x)\psi(x)\, dx \tag{6}$$

The difference to the ordinary inner product in L^2 is that here none of the factors is conjugated. This leads to the result that it actually isn't a proper (positive-definite) inner product and also that there are non-trivial self-orthogonal vectors. However, the important property is that with (6) also a complex scaled Hamiltonian is still "symmetric" in the sense that

$$\left(\phi|\hat{H}_\theta \psi\right) = \left(\hat{H}_\theta \phi|\psi\right) = \left(\phi|\hat{H}_\theta|\psi\right),$$

which can be seen easily by a short calculation and integration by parts. Second, one can derive a *generalised variational principle* that also holds true for complex scaled Hamiltonians based on the c-product:

Theorem 1. *Let $\psi(\alpha) \in H^k(\mathbb{R}^n)$ be a parameter-dependent wave function, where $\alpha \in \mathbb{C}$. Let $\psi(\cdot)$ be differentiable and assume that $\hat{H}_\theta \psi(\alpha_0) = E\psi(\alpha_0)$, $\|\psi(\alpha)\|_{H^k} = 1$ for all α considered (normalise as necessary), and $(\psi(\alpha_0)|\psi(\alpha_0)) \neq 0$. We consider the Rayleigh quotient in the c-product*

$$R(\psi) = \frac{\left(\psi|\hat{H}_\theta|\psi\right)}{(\psi|\psi)}$$

now as function $R(\alpha) = R(\psi(\alpha))$. Then

$$\frac{\partial R(\alpha_0)}{\partial \alpha} = 0. \tag{7}$$

In other words, instead of the one-sided bound we at least get a stationarity property *of the energy expectation for eigenstates of the form*

$$R(\psi + \epsilon\delta\psi) = E + O\left(\epsilon^2\right).$$

Proof. See [6, (7.17), p. 214] or [5]. □

A first application of Theorem 1 is the case of *linear* parameters (in other words, using a fixed basis with unknown coefficients as ansatz for $\psi(\alpha_i)$). There, it can be seen that the resulting condition can be reformulated as generalised eigenvalue problem for matrices: Let $(\chi_i)_{i=1}^n$ be some fixed basis, and denote by

$$H_{ij} = \left(\chi_i | \hat{H}_\theta | \chi_j\right), \quad S_{ij} = (\chi_i | \chi_j) \tag{8}$$

the Hamiltonian and overlap matrix elements. Then the stationarity condition (7) is equivalent to solving

$$Ha = ESa$$

as generalised eigenvalue problem for the coefficient vector $a \in \mathbb{C}^n$ and the resonance energy $E \in \mathbb{C}$. The remaining question is how to choose the basis of χ's. A method that has been proven to be efficient in practice for certain problems is the *stochastic variational method* [10], whose main idea is to enlarge the basis step by step: If it currently has length k, create *randomly* a set of candidate functions for χ_{k+1} and then pick that one to actually add which minimises the energy expectation value—in our case based on Theorem 1, a good generalisation seems to be to pick that one which produces the *smallest norm of the gradient* of R with respect to all non-linear parameters that specify the basis functions. To further concretise the method, a good choice as basis functions are *Gaussians* of the form

$$\chi_{\alpha,\mu}(x) = e^{-\alpha|x-\mu|^2}. \tag{9}$$

They are quite flexible to approximate both regions of a target function where it is smoothly varying over long distances (with small α's) as well as points of sharp changes (with large α's and μ's to localise a basis function there). Furthermore, they are nice, smooth, analytic functions, for which the expressions required in (8) (at least the kinetic part of the Hamiltonian which does not depend on the specific problem at hand) can be evaluated symbolically as Gaussian integrals.

1.4 An Example Problem

For my numerical calculations, I used a simple one-dimensional potential problem with the Hamiltonian (2) and the potential

$$V(x) = \left(\frac{1}{2}x^2 - J\right) e^{-\lambda x^2}, \tag{10}$$

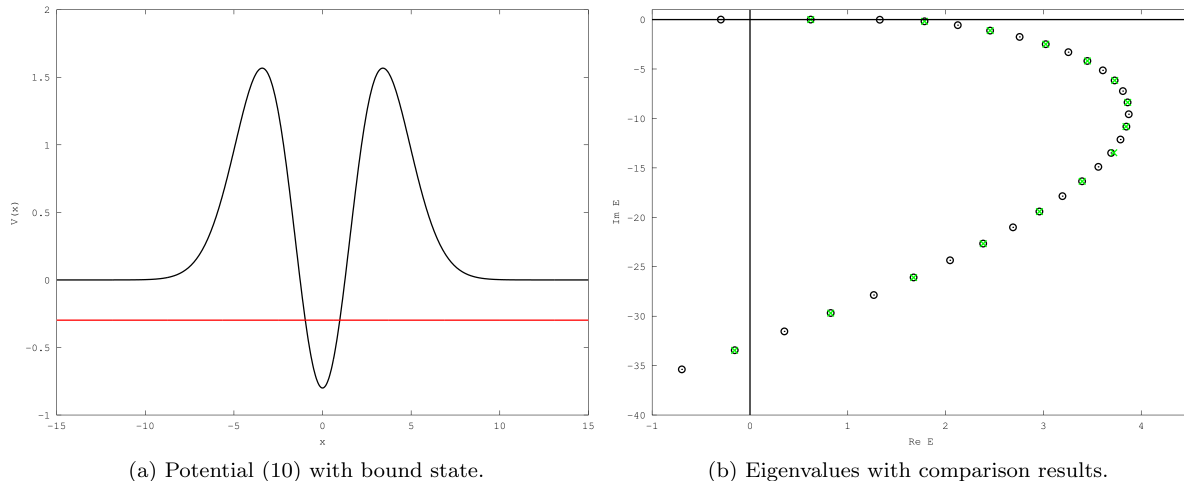
where $x \in \mathbb{R}$ and the parameters were chosen as $J = 0.8$, and $\lambda = 0.1$. For this problem, the correct resonance energies are already published in [4] and can be used to check my own calculation. Note though that in this paper, the potential used has an additional offset J added to it in comparison to (10). This results in a corresponding shift of the eigenvalues, which has to be taken into account when comparing my results to theirs. The potential (10) as well as its eigenvalues are shown for a quick overview in Figure 1. These plots are taken from [5], which gives some more details about them.

2 Calculating Derivatives

It should be clear that calculating derivatives of the Rayleigh quotient (5) is a very important ingredient in order to make the variational method work. For my code I implemented two different methods, which gave the same results and thus confirmed that both implementations are viable. The high-level code is implemented using GNU Octave [1], while both differentiation methods described in the following yield native code. These routines were then accessed from Octave using its C++ interface based on so-called *oct-files*.

2.1 Manual Symbolic Derivatives

Since the expressions appearing in (8) are mostly only Gaussian integrals that can be evaluated symbolically, the resulting matrix elements can also be differentiated symbolically. For the potential part of the H_{ij} matrix elements, symbolic integration is not possible but at least the integrand can again be differentiated symbolically, so that also the derivatives of those expressions can be calculated by applying numerical quadrature on the differentiated integrands. This is described in more details in [5]. Once the



(a) Potential (10) with bound state.

(b) Eigenvalues with comparison results.

Figure 1: The potential (10) with its single bound-state energy on the left, and the corresponding eigenvalues on the right. Green crosses mark the values given in [4], which are available for resonances with odd index.

derivatives of all matrix elements can be calculated, it is easy enough to use the linear structure and the quotient rule to find the derivatives of the full expression in (5): It is a little tedious to fit all pieces together correctly, but it can be done—in my particular case, I used Maxima [2] to perform symbolic differentiation, since the expressions involved grow quite long. From Maxima, I exported the derivative expressions as Fortran code using the `fortran` routine, and the resulting code was then compiled and used from a C++ oct-file routine using Fortran's interface to C, aka `ISO_C_BINDING`. This C++ code then assembles the full derivative from the individual matrix elements and does the quadrature, returning everything to Octave.

2.2 Automatic Differentiation

The same derivatives can also be calculated using automatic differentiation, which allows me to get rid of the manual assembly of the subexpressions as well as calculating the symbolic derivatives of the matrix elements myself. I used ADOL-C [7] for this task, which is based on operator-overloading in C++. The automatic differentiation was performed in *reverse mode* based on the high-level driver function `jacobian` of ADOL-C, because for calculating the gradient of our scalar expectation value, the number of independent variables is much larger than the number of dependent ones and thus the reverse mode is more efficient than the forward mode. Since ADOL-C is based on C++, it is straight-forward to use it from code that is then compiled into an oct-file to be used from Octave. All that needs to be done is implementing (5) itself in C++, then the gradient of this expression is deduced by ADOL-C upon request. This is of course much easier than manually building up the derivatives, and in particular once the code is set up correctly, it is trivial to calculate also the Hessian of (5)—which would require much more tedious assembly of expressions when done manually (although that would be possible in theory, too). The reader is referred to [7] or the ADOL-C manual [11] for more details of how ADOL-C works and can be used in general, but in the following I will describe a particular difficulty (and its solution) I faced for my problem: The use of *complex numbers* for dependent variables.[1]

In order to switch existing C++ code over to ADOL-C, (real) floating point numbers must be given the type `adouble` instead of `double`, which is a special class that implements the steps necessary for automatic differentiation in overloaded operators and mathematical functions. For complex numbers in C++, the template class `std::complex<T>` of the STL can be used (and is in my case)—this class implements complex number operations for arbitrary underlying numerical types `T`, which are used for the complex

[1]The independent variables were still real (in my case, the parameters α and μ of (9)), so no complex derivatives as in Cauchy-Riemann differential equations are involved.

number's real and imaginary parts. Thus ordinarily, one would use for instance `std::complex<double>` as type for complex numbers in calculations, and to allow differentiating them with respect to some real independent variables, I had to use `std::complex<adouble>`. Since `std::complex` already "reduces" the complex number operations to corresponding expressions involving just real quantities (the real and imaginary parts), this mostly plays very well together with `adouble` to differentiate those expressions later with respect to our independent variables.

Example 1. Let $t \in \mathbb{R}$ be a parameter and define $a = 2t + 5i$, $b = -t + \frac{t}{2}i$. Then

$$x = x(t) = \frac{a}{b} + ab \in \mathbb{C}$$

is a parameter-dependent complex number. The operations involved in the definition of x are simple enough to allow ADOL-C to differentiate it right out-of-the-box. The corresponding code to calculate the derivative $\frac{dx}{dt} \in \mathbb{C}$ is shown in Listing 1. Everything related to ADOL-C is marked in red to emphasise the steps necessary. We have the usual boiler-plate code related to first tracing the execution and then replaying the resulting "tape" to calculate the derivatives, but also some steps are required in order to construct `a` and `b` from `t` via the active variable `myT` first, and to read out the real and imaginary parts of `x` into `re` and `im` after the calculation. Since ADOL-C doesn't know complex numbers natively, we have to pretend that there are *two* (but real) output variables. The actual calculation with the complex numbers (which is the assignment to `x` in this example) is straight-forward, though, for this simple expression. It can be easily verified (since the derivative can also be calculated symbolically) that the shown code gives correct results.

Example 2. One important part of my calculations with (9) is of course the exponential function.[2] This gives us another example: Let again be $t \in \mathbb{R}$ a single parameter, and define $a = 2t - ti$. We want to calculate $x = e^a$ this time. The naive code to do this (only the relevant part performing the calculation) is shown in Listing 2. However, this code fails to compile[3] with the following error:

```
complex:738:52: error: no matching function for call to polar(adub, adouble)
```

The relevant piece of code from gcc's implementation of the complex exponential function is:

```
return std::polar(exp(z.real()), z.imag());
```

This uses the fact that the complex exponential function is directly related to polar coordinates for complex numbers, since with

$$z = a + bi \quad \Rightarrow \quad e^z = e^a \cdot e^{bi} = re^{i\phi}$$

it is clear that the magnitude of e^z is e^a and the argument is b. However, digging into the internals of ADOL-C it becomes evident that it not only uses `adouble` to represent numerical values with operator overloading added, but also `adub` for internal purposes which is used to represent "temporary results" of operations. Both classes are siblings to each other in the inheritance hierarchy, being derived from a common base class `badouble` and with appropriate conversion routines to allow, for instance, assigning an `adub` result to an `adouble` variable. This makes the existence of `adub` *mostly* hidden to the user, but in the concrete case above it surfaced because `exp(z.real())` is of type `adub` while `z.imag()` is `adouble`—but the template implementation of `std::polar` can only be resolved if both arguments are of the same type. To resolve the issue, one can rewrite Listing 2 into the code shown in Listing 3, which duplicates the implementation of `std::exp` but forces a conversion of the `adub` intermediate result back to `adouble`. This works perfectly fine, and also for this example the result of automatic differentiation can of course be easily verified in comparison to the known symbolical derivative.

3 Results

Both methods described above work well to calculate the gradient of the energy expectation value (5) with respect to the non-linear variational parameters, and give consistent results. The symbolic calculation

[2] It is not necessary to evaluate it at complex values for my application, but my code is implemented to support that.
[3] At least for ADOL-C version 2.4.1 and with g++ 4.7.2, which is the setting I used.

```
/**
 * Perform a simple calculation with complex numbers which
 * doesn't require any tricks to do with ADOL-C.
 * @param t Value of the independent variable at which to evaluate.
 * @return The t-derivative of the function value there.
 */
static std::complex<double>
simpleTest (const double t)
{
  /* First, ADOL-C "tapes" the execution of the calculation code.  This
     records all operations, and allows to later "replay" them to perform
     automatic differentiation in a variety of modes and configurations.  */

  trace_on (1);

  adouble myT;
  myT <<= t;
  std::complex<adouble> a, b;
  a = std::complex<adouble> (2.0 * myT, 5.0);
  b = std::complex<adouble> (-myT, myT / 2.0);

  std::complex<adouble> x;
  x = a / b + a * b;

  double re, im;
  std::real (x) >>= re;
  std::imag (x) >>= im;

  trace_off ();

  /* Now we have the tape and can actually replay it to calculate the derivative
     of the result with respect to t.  Behind-the-scenes this works by
     assuming *two* dependent variables (real and imaginary part) and
     calculating the (real) Jacobian of this two-dimensional function.  Later
     on, both coordinates are again combined to a single but complex-valued
     derivative result.  */

  double* jac[2];
  jac[0] = new double ();
  jac[1] = new double ();
  jacobian (1, 2, 1, &t, jac);

  std::complex<double> res(*jac[0], *jac[1]);

  delete jac[0];
  delete jac[1];

  return res;
}
```

Listing 1: Simple function using ADOL-C on complex numbers for Example 1.

```
std::complex<adouble> a;
a = std::complex<adouble> (2.0 * myT, -myT);

std::complex<adouble> x;
x = std::exp (a);
```

Listing 2: Naive implementation of Example 2.

```
std::complex<adouble> a;
a = std::complex<adouble> (2.0 * myT, -myT);

std::complex<adouble> x;
const adouble magn = exp (std::real (a));
const adouble arg = std::imag (a);
x = std::polar (magn, arg);
```

Listing 3: Working implementation of Example 2.

is however more efficient by a factor of about 42,[4] but that is no surprise since it is clear that most of the time a symbolic derivative, *if it is possible and feasible*, is best. The big advantage of automatic differentiation is that it is easier to implement and can also calculate second derivatives without much additional effort, which would be very hard and tedious to get correct with symbolic expressions.

On a higher level, the stochastic variational method I developed in [5] gives good results as can be seen from Figure 2a: It shows how a particular resonance eigenvalue (for the $n = 3$ resonance) converges to the exact value with increasing length of the stochastic basis. Black is the curve corresponding to completely random basis selection (i. e. without applying any selection criterion like minimising the gradient norm), blue is the described method based on the gradient norm and green is a different method I propose (see [5] for the details). One can easily see that both methods outperform random basis selection by a large margin. In Figure 2b the same error is shown if the Hamiltonian $\hat{H}_\theta$ is approximated using finite differences and the eigenvalues of the resulting (tri-diagonal) matrix are calculated directly. It can be seen that in order to achieve the maximum accuracy of about 10^{-8} of the stochastic variational method with $N = 80$ basis functions, about 100,000 grid-points would be necessary. For higher-dimensional problems the finite-difference approximation would become even much more expensive. Note also in particular that the basis length axis is *not logarithmic* in Figure 2a while *it is* in Figure 2b.

References

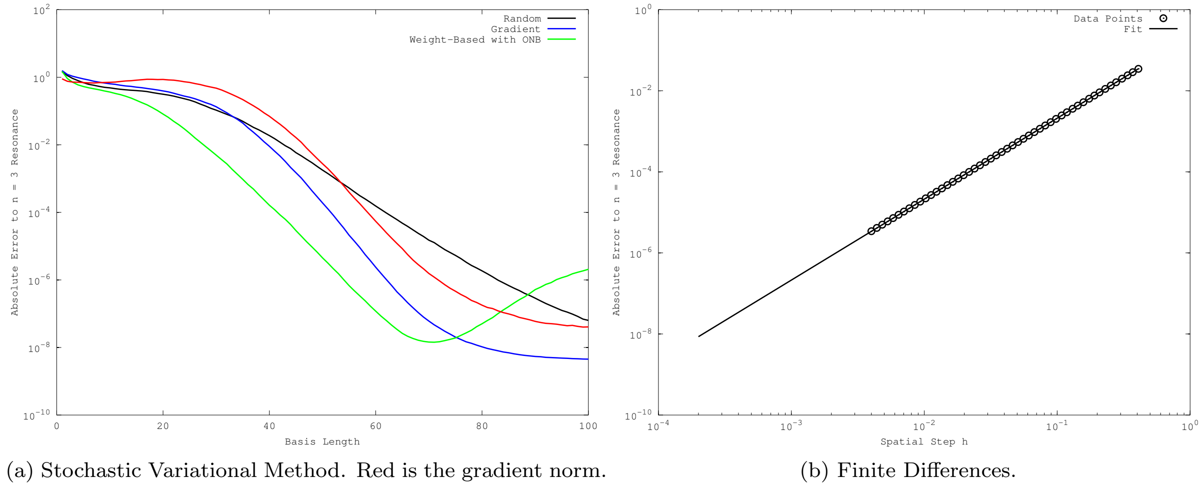
[1] GNU Octave. `https://www.gnu.org/software/octave/`.

[2] Maxima: A Computer Algebra System. `http://maxima.sourceforge.net/`.

[3] Torsten Fließbach. *Quantenmechanik*. Number 3 in Lehrbuch zur Theoretischen Physik III. Springer, Heidelberg / Berlin, third edition, 2000.

[4] H. Jürgen Korsch, Helmut Laurent, and Ruth Möhlenkamp. Comment on "Weyl's theory and the complex-rotation method applied to phenomena associated with a continuous spectrum". *Physical Review A*, 26(3), September 1982.

[4] For a particular representative calculation run, profiling showed that 155 seconds were spent for the gradient calculation with the symbolic code and about 6,562 with automatic differentiation. Note that this already includes reuse of a recorded tape for different values of input variables as appropriate for my application. In both cases, this was by far the most expensive part of the calculation.



(a) Stochastic Variational Method. Red is the gradient norm.

(b) Finite Differences.

Figure 2: Results for my stochastic variational method calculations in comparison to a finite-difference approximation of the scaled Hamiltonian $\hat{H}_\theta$.

[5] Daniel Kraft. Stochastic Variational Approaches to Non-Hermitian Quantum-Mechanical Problems. Master's thesis, Universität Graz, 2013. Forthcoming, will be available at `http://www.domob.eu/research.php`.

[6] Nimrod Moiseyev. *Non-Hermitian Quantum Mechanics*. Cambridge University Press, 2011.

[7] U. Naumann and O. Schenk, editors. *Combinatorial Scientific Computing*, pages 181–202. Chapman-Hall CRC Computational Science, 2012. A. Walther and A. Griewank: Getting started with ADOL-C.

[8] E. Prugovecki. *Quantum Mechanics in Hilbert Space*. Academic Press, New York / London, second edition, 1981.

[9] John R. Taylor. *Scattering Theory: The Quantum Theory on Nonrelativistic Collisions*. John Wiley & Sons, New York / London / Sydney / Toronto, 1972.

[10] K. Varga and Y. Suzuki. Precise solution of few-body problems with the stochastic variational method on a correlated Gaussian basis. *Physical Review C*, 52(6):2885–2905, December 1995.

[11] A. Walther and A. Griewank. *ADOL-C: A Package for the Automatic Differentation of Algorithms Written in C/C++*. `https://projects.coin-or.org/ADOL-C/browser/stable/2.1/ADOL-C/doc/adolc-manual.pdf?format=raw`.