A Generic Approach to Electronic Structure Calculations in Nanoscopic Systems

The generic programming facilities of C++ allow the creation of efficient, yet flexible computational systems. The Diagon framework describes the computation of energies and states for general quantum nanoscopic systems. It uses the configuration interaction method, in principal obtaining all many-body correlations of the system.

An efficient use of time—computational or otherwise—is important for all of us. For basic research, the most relevant time is often that between initial idea and final results, and by no means is this duration always dominated by raw computation time. A more useful interpretation of efficiency than actual computation time should include development time as well. Arguably, having a flexible computing environment can sometimes be more important than blazingly fast code. (But having both is best!)

In this article, I outline a generic and flexible framework for computing energies and states for general nanoscopic systems with a focus on semiconductor quantum dots. The approach uses the configuration interaction (CI) method to obtain all the system’s many-body correlations. It also exploits the C++ template facility’s abstracting mechanism to produce efficient, yet general code.

The essential idea is that much of the CI machinery is independent of the actual single-particle states used, so a generic approach lets us explicitly separate algorithms and data types as well as reuse a great deal of code. My students and I have used generic programming techniques in our Diagon framework, which focuses on (but is by no means restricted to) 2D semiconductor quantum dots.

Configuration Interaction

The CI approach to electronic structure calculations is conceptually simple and potentially capable of being exact in the sense that we can exactly compute all the eigenstates and energies of the given model system given sufficient computational resources. The drawback is that this method requires a great deal of resources that grow exponentially with the number of particles in the physical system. Generally speaking, state-of-the-art calculations cannot go much further than about 10 particles because the CI method is essentially a direct diagonalization of the system’s Hamiltonian matrix, and the size of this matrix grows exponentially with the number of particles.

To see this, consider a quantum dot (equivalently, an atom or molecule) with $M$ orbitals and $N$ electrons ($M \geq N/2$). To find the system’s many-body eigenstates, we first build up the many-body basis states and then use them to diagonalize the Hamiltonian. To build up the basis states, we take the $N$ electrons and distribute them over the $M$ orbitals. Combinatorially, we have many different ways (or
configurations) in which to do this, hence the exponential increase. (Actually, because electrons are indistinguishable fermions, we need to antisymmetrize the configurations as well.\(^2\)) Think of the off-diagonal terms in the Hamiltonian matrix as an interaction between different configurations—hence, the name configuration interaction.

The CI technique, while conceptually simple, becomes computationally prohibitive very quickly. Because of this, other techniques, such as quantum Monte Carlo (QMC)\(^3\) and density functional theory,\(^4\) are often required in place of CI calculations. Of these three, QMC and CI methods are the two primary tools when correlations are strong, when many excited states in addition to ground states are required, and especially when the many-body states themselves (as opposed to the energies) are required. QMC has great potential for efficiency, but it's far more complex than CI both in concept and in practical aspects of coding. Thus, QMC is well suited for problems that require a large number of particles or for stable production code applied to a specific, fixed set of systems over a prolonged time period.

We can distinguish, however, between fast code written slowly—that is, that takes a long time to code, test, debug, and maintain—and slow code written quickly. QMC belongs in the former category, whereas CI belongs in the latter. A properly structured CI base can be an important piece of a research framework, particularly when flexibility is desirable.

By consolidating this dual need for efficiency and flexibility, generic programming can be of great benefit. At its most fundamental level, generic programming separates data types from algorithms. This separation makes sense—after all, data types and algorithms are orthogonal concepts. We can sort integers in the same algorithmic way that we sort apples; the only difference is how we define less than. Although the meaning is clear for integers, for apples, less than can mean a comparison of size, color, age, and so on. Once we define how to compare objects—any object—the sort proceeds in the exact same way.

A generic programming way to implement a sort routine is to supply the routine with the objects to be sorted and, additionally, supply a comparator that determines which object is less than the other—a sensible default is the “<” operator. This one routine can now work for all possible objects. To sort integers in increasing rather than decreasing order, use a “greater than” comparator; to sort student records by family name, use the same routine but supply a more involved comparator that looks at two student records and compares the family names. Sorting by grades? Use the same routine and the same data collection again, but supply a different comparator. The advantage is a gain in flexibility, plus you never need to write a sort algorithm again—instead, you just write comparator functions, which are usually fewer than five lines long.

My students and I have found these generic concepts of great use in our work on quantum condensed matter theory in general and in the CI method applied to nanoscopic quantum dots in particular.

### A Flexible Computing Environment

Flexibility is particularly required for computations involving the basic physics of synthetic nanostructures such as semiconductor quantum dots.\(^5\) Three main reasons account for this:

- the nonuniversality of synthetic structures,
- the insufficiency of ab initio methods for large systems, and
- the many different environmental couplings in heterogeneous solid-state devices.

We can explore these issues by comparing them to atomic systems, in which robust and highly efficient code is available and theoreticians have a tremendous amount of robust, clean, and precise experimental results on which to draw.

For definiteness, let’s consider semiconductor quantum dots, or artificial atoms,\(^6\) which are defined by electrostatic (lateral) confinement of electrons in a 2D electron gas. Unlike atoms, each such manufactured device is unique: two devices with ostensibly identical gate geometries and identical numbers of confined particles will nevertheless have unique confining potentials, tunneling barriers, and other parameters that significantly affect the device’s transport, electronic, and spin properties. Contrast this with atoms, in which every distinct atomic species is an indistinguishable particle: a large-scale, robust, sophisticated, and efficient code base\(^7\) eventually pays off because the resulting application is indefinitely applicable to all future problems of the same atomic species. In lateral quantum dots, the confinement potential is unknown beforehand, and what’s more, experimentalists can and often do change the confinement’s shape as they change the number of particles, the coupling to leads, and so on. If they also change the semiconductor growth characteristics, then the effective mass of the charge carriers can change as well as the strength of the Zeeman g-factors.\(^8\) A flexible computational
Numerical results are more properly viewed as aids to theoretical analysis rather than pure simulation, so flexibility is key.

to deal with certain questions (nor were they designed to do so). For example, recent experiments measure a certain decoherence time for spins and charge in a semiconductor nanodevice, but it’s unclear what the dominant contribution is to this decoherence. Candidates might be hyperfine interactions with nearby donor nuclei, spin–orbit interactions among the electrons inside the device, strong electron–electron interactions between electrons inside or outside the device, stray electric and magnetic fields, the specifics of the interface inhomogeneities of this particular device, the band structure, or a combination of some or all of these. From a first-principles viewpoint, all these mechanisms are equivalent because they’re all caused—indeed, all of chemistry and most of physics, to take an extreme view, is caused—by Coulomb interaction. But this doesn’t really tell us anything about what’s going on in the system. (Some might find my statements here controversial or at least debatable; supporting work offers a lengthier discussion on the topic.) In such cases, numerical results are more properly viewed as aids to theoretical analysis rather than pure simulation, so flexibility is key.

Furthermore, unlike atomic systems, semiconductor systems are dirty; isolating anything from its environment is next to impossible. Besides the
More than any other feature, correctness is the single most important requirement of all numerical code. An aid, but by no means a savior, to correctness is the language’s type safety, which is the degree to which the language limits and allows control over automatic conversion from one type to another—for example, converting an integer to a real number is probably (not always!) okay, but not the other way around. If the language allows for the creation of programmer-defined data types (or data structures), then conversions between these additional types should do the sensible thing, or at least, the compiler should issue warnings. Suppose we define a new type Position consisting of a triplet of real numbers \((x, y, z)\) and a Momentum, also of three reals \((p_x, p_y, p_z)\); then we should be able to add two of the same type naturally (say \texttt{position1+position2}). However, it makes no physical sense to add a position and a momentum, even though they’re both just three reals to the compiler. Different languages offer different levels of programmer control over such details; C++, for instance, allows complete control (should you wish it) over the level of automatic conversion between data types.

Additionally, constants in your code (a particle’s mass, for example) should be just that—constant and unchangeable. We want these to be fixed once the variable is created, so the compiler should never allow this quantity to change in any way. This is particularly important when passing arguments to functions. Parameters to functions can be passed by value (a local copy is automatically made, leaving the original unchanged, which is the default in C and C++), by reference (the object itself is passed and can thus be altered, accidentally or otherwise, which is the default in Fortran), or my favorite, by constant reference (passed by reference, but changes aren’t allowed). These often obviate the need for raw pointers to memory and all their error-prone complexity.

Authors should have at their disposal (and their discretion) a great deal of control over how memory is precisely allocated and how objects are created, assigned, copied, or converted to other types. These language features are great tools for producing flexible, yet robust numerical frameworks.

**Generic Programming**

None of my previous discussion has anything to do with object-oriented programming, the paradigm most frequently associated with C++. Many definitions attempt to determine what constitutes object-oriented programming, but the four most common concepts are classes, objects, inheritance, and runtime polymorphism. Of these, the first two have become ubiquitous in virtually every programming language, so it is the latter two that really distinguish object-oriented programming from other styles. Because we’re discussing generic programming, we need classes and objects but not inheritance or runtime polymorphism, so I won’t discuss them here.

Briefly, a class is a construct that defines a data structure and the functions that can be called on that data structure—for example, a class Particle might define a Position, Momentum, and Mass as data members and energy() and incrementMomentum() as member functions. Objects are particular instantiations of a class—for example, the statement Particle \(x, y\) defines two objects, \(x\) and \(y\), of class Particle. Our code might therefore have statements such as \(x\).energy(), which will return the particle’s energy, or \(y\).incrementMomentum().

Although C++ is largely known as an object-oriented programming language, it supports several programming styles, including procedural and functional, as well as object oriented. However, in my opinion, the single most useful feature of C++ with regard to numerical computation is its generic programming facilities because they can bridge expressiveness on the one hand with efficiency on the other. Particularly for scientific programming, recent developments in generic programming have clearly shown that flexibility and abstraction need not necessarily incur a runtime penalty, thereby moving the state of the art far beyond simple parameterized types.

At its most basic level, generic programming separates data types from algorithms. As I mentioned earlier, we can write a single sort algorithm that sorts objects of an arbitrary class, and because of operator overload, the class’s author can define the less-than operator “\(<\)”. This ability makes generic programming code forward-compatible in time, rather than simply backward-compatible with previous versions. If a particular routine compares two objects using statements such as \(A < B\), future authors simply (re)define the “\(<\)” operator for their particular class and use the original routine as is for classes that the original author never dreamed of. If a specific type requires a more efficient algorithm, that particular type can be made into a special case through partial template specialization.

Importantly, this great flexibility comes with little or no performance penalty, especially compared to dynamically typed languages such as Python (www.python.org) or Ruby (www.ruby-lang.org).
org). Dynamically typed languages are typically interpreted rather than compiled. Here, the type of an object, and therefore the amount of memory it requires for storage, the functions on which it can be called, and so on, are all determined at runtime, once the execution path encounters the instruction. A statically typed language, such as C++ and Fortran, builds this information into machine code at compile time, so that all this information is already known at runtime. These programs typically run much faster (possibly orders of magnitude faster), but they take longer to program and are less flexible. Fortunately, we can use languages such as Python and C++ in concert to great success. (For an example, see the 1999 CiSE article by Paul F. Dubois.18)

C++’s generic facilities also produce statically typed compiled code. In effect, a generic template function is a program that writes programs and is essentially a type-safe metaprogramming facility. Upon compilation, the compiler takes a template function and instantiates a specific version for whatever data type is required. Thus, a single-sort template function can produce several custom-made (by the compiler) versions in the executable; one each for, say, integers, complex numbers, apples, and many-body state vectors. The important fact is that this happens at compile time, not runtime, so the produced code is more efficient. The compiler can turn this single generic function into an unbounded number of actual functions operating on types that the original author couldn’t have conceived.

The Diagon framework I describe later in this article only scratches the surface of what generic programming can do. We can use templates to keep track of dimensional quantities19 so that, for example, the system knows that length/time is a velocity and can give a compile-time error when a length is assigned to a velocity, even though all quantities are real numbers. In fact, scientific programmers recognized early on the benefits of generic programming.20

To examine generic programming’s flexibility, let’s consider the inner product of various quantum state vectors describing electrons in a quantum dot. At least three types of states are generally required. At the single-particle level, we have the base single-particle orbital states \( \ket{\alpha} \), where \( \alpha \) denotes a full set of quantum numbers, which we label generically as \( \text{SPState} \). A many-body Fermion system is described by antisymmetrized vectors \( \ket{\alpha_0, \alpha_1, \alpha_2, \ldots} \) in accordance with the Pauli exclusion principle.2 These are essentially determinant wavefunctions labeled \( \text{AntiSymmState} \).

Correlated states

\[
\ket{\psi} = \Sigma_i c_i \ket{\alpha_0^i, \alpha_1^i, \alpha_2^i, \ldots},
\]

induced by Coulomb interactions or spin symmetry,21 require a description in terms of coherent superpositions of Slater determinants. In Diagon, these are encapsulated by the generic class \( \text{LinCombState} \).

\( \text{SPState} \)’s inner product clearly depends on the particular system under consideration, but the algorithmic computation of inner products of either \( \text{LinCombState} \) or \( \text{AntiSymmState} \) are essentially identical and independent of the underlying \( \text{SPState} \). Thus, once supplied with an inner-product function for the particular \( \text{SPState} \), the calculation for inner products involving more complex state vectors can proceed automatically, without rewriting the functions.

The object-oriented (nongeneric) solution to this problem is to define a class hierarchy and pass pointers to the functions; the execution thread will then determine the appropriate inner product at runtime. Besides the extra runtime cost involved in dereferencing objects for dynamic polymorphism, this approach will almost certainly lead to an ever-growing hierarchy of states, with commensurate costs in maintenance and testing. In addition, we must alter the core code base with every new \( \text{SPState} \) introduced to the system. As the number of states known to the system grows, maintenance and continued testing for correctness become even more of a burden.

In a generic approach, the code base remains small. Here, we define generic containers \( \text{AntiSymmState}\{\text{SPState}\} \) and \( \text{LinCombState}\{\text{SPState}\} \), which can hold any type of state. (Note that a \( \text{LinCombState} \) can describe linear combinations of \( \text{AntiSymmState} \)‘s with different sizes. This would be required when the particle number isn’t conserved—for example, when studying transport through a quantum dot.22) Provided the user implements an \( \text{SPState} \) class and defines single-particle inner-product functions, we don’t need to (re)write inner products involving many-body states because the compiler will write a custom version of the appropriate inner-product function. The template functions themselves indicate the algorithm, not the data types.

Because we can combine generic components as required, a small set of generic components can produce combinatorially many functions. Thus, with a small set of components, we can define a large array of composite objects, maximizing the numerical approach’s flexibility. This flexibility
need not incur a runtime penalty, however, because all template instantiations are implemented at compile time. This is a tremendous advantage both in the speed of developing a custom application from a set of generic components and in maintaining efficient code at runtime.

**Diagon Framework**

We have implemented the generic Diagon framework to compute eigenstates and eigenvalues of semiconductor quantum dots for arbitrary potentials and with arbitrary numbers of particles and Hamiltonians. This framework, meant for developing applications rather than to serve as a tool for end users, consists of a set of generic components useful in manipulating many-body quantum states and the computations involving them. It currently has components for dealing with various types of fermion states and computing matrix elements of one- and two-body operators, generators for returning proper spin eigenstates given a set of singly and doubly occupied single-particle orbitals, and components for computing eigenstates and eigenvalues of Hermitian operators. These components are all generic, so the single-particle states themselves must be provided.

**Generic Quantum Many-Body States**

Three classes of many-body states exist: \texttt{AntiSymmState<SPState>}, \texttt{LinCombState<State, Coeff>}, and \texttt{StateSet<State, Coeff>}.

- \texttt{AntiSymmState<SPState>} is the primary class for antisymmetrized product states (in real space, these are Slater determinants). Upon instantiation, the (generic) parameter \texttt{SPState} must name an existing class for encapsulating a known single-particle state—for example, in 2D parabolically confined quantum dots, the single-particle states might be the well-known Fock-Darwin states \(|\ell m n\rangle|^{23}\). Operations are provided by Diagon for creating and destroying particles in these states, as well as for computing inner products if supplied with the inner product for the underlying single-particle states.

In addition to the (user-provided) single-particle states and the (Diagon-provided) \texttt{AntiSymmState}, a generic class \texttt{LinCombState<State, Coeff>} is provided by Diagon for encoding linear superpositions of states, \(|\text{LinCombState}\rangle = \sum \alpha_i |i\rangle|\). Here, the template parameter \texttt{State} is the type of component state (which can, for example, be an \texttt{SPState} or \texttt{AntiSymmState}), and \texttt{Coeff} is a template parameter encoding the type of coefficient, which will usually include real or complex numbers, but because the class is generic, it could be more exotic. Diagon provides operations for adding and removing terms from the sum, checking and setting normalization of the state as a whole, indexing a particular term, and iterating over all terms. If class \texttt{State} defines an inner product, then Diagon defines both \texttt{(LinCombState<LinCombState>)} and \texttt{(LinCombState<LinCombState>)}.

The final class of many-body states is \texttt{StateSet<State, Coeff>}, which encapsulates a set of \texttt{LinCombState<State, Coeff>} objects that share a common basis. We use it, for example, as a return type from a diagonalization routine in which each \texttt{LinCombState} is an eigenstate. We can also use it to describe a set of spin eigenstates. Diagon provides operations to add or remove a basis vector to the \texttt{StateSet} or to add or remove a \texttt{LinCombState}.

With these three generic components, we only need to define a particular single-particle state and an inner product between these single-particle states. Thereafter, many-body states and linear superpositions of them become available, as do a full suite of inner products and other manipulations and computations.

**Generic Operator Functions**

A general quantum operator \(\hat{O}\) can be described as a matrix in which the basis vectors can be \texttt{SPState}, \texttt{AntiSymmState}, or \texttt{LinCombState}. The Diagon framework provides generic operators for one- and two-body operators.

We can write a general one-body operator in second quantized form as

\[
\hat{O} = \sum_{\ell p} O_{\ell p} \hat{c}_\ell^\dagger \hat{c}_p,
\]

where \(\hat{c}_\ell^\dagger\) creates a particle in the single-particle state \(|\ell\rangle\), and \(\hat{c}_p\) destroys one in state \(|p\rangle\). We can write a general matrix element of \(\hat{O}\) as \(\langle \psi | \hat{O} | \psi' \rangle\), where the states \(|\psi\rangle\) and \(|\psi'\rangle\) are, in general, many-body states \(|p_1, p_2, \ldots, p_n\rangle\) or linear superpositions of such states. Whatever they are, \(\langle \psi | \hat{O} | \psi' \rangle\) evaluates to a scalar of the same type as the set of numbers \(O_{\ell p}\) which are usually real or complex. The Diagon framework implements matrix...
elements of the operator $\hat{O}$ generically as a function 
\[ \text{oneBodyOp}(\text{bra}, \text{ket}, \text{matelem}), \]
which returns the matrix element $\langle \psi | \hat{O} | \psi \rangle$. Here, $\text{bra}$ and $\text{ket}$ are the states $| \psi \rangle$ and $| \psi \rangle$ and can be any 
(combination) of the three basic types of states from the last section, and $\text{matelem}$ is a user-defined 
fraction evaluating $O_{ij}$ in Equation 1. That is, $\text{matelem}$ must have the signature 
\[ \text{ReturnType} \text{matelem}(\text{SPState} \text{bra}, \]
\[ \text{SPState} \text{ket}), \]
where $\text{ReturnType}$ is an arbitrary type. The generic function $\text{oneBodyOp}$ returns the same type 
that $\text{matelem}$ returns.

Once the user defines single-particle properties, which will differ from system to system, the 
Diag framework implements the many-body functionality, which is, in a sense, a universal 
function of the single-particle physics.

We can similarly implement two-body operators; the difference is in how we treat spin. In 
paticular, we can write any spin-independent two-body operator (such as the Coulomb interaction) as 
\[ \hat{U} = \sum_{i,j,k,l} U_{ijkl} \sigma_{i,j} \sigma_{k,l}, \]
where the user must provide the $U_{ijkl}$, with indices $i, j, k,$ and $l$ denoting single-particle orbital states, 
and $\sigma$ and $\sigma'$ denoting spin. This is implemented as 
\[ \text{twoBodyOp}(\text{bra}, \text{ket}, \text{matelem}), \]
but the two states must now contain at least two particles (AntiSymmState or LinCombState), and 
\[ \text{matelem} \] must have the signature 
\[ \text{ReturnType} \text{matelem}(\text{SPState} \text{bra1}, \text{SPState} \text{bra2}, \]
\[ \text{SPState} \text{ket1}, \text{SPState} \text{ket2}). \]

Shortly, I’ll describe a specific example using these generic components. Before I do, however, 
let’s review the generic facilities Diag provides for computing eigenstates of total spin and other 
more general Hermitian operators.

**Generic Spin States**
Because the CI technique is so computationally intensive, it’s important to take advantage of 
every significant symmetry in the system because doing so affords a possibility to block-diagonalize the 
Hamiltonian matrix, drastically reducing the computational load. Simple symmetries, such as 
conservation of spin or angular-momentum projection along a given axis (say, $S^z_\text{tot}$ or $L^z_\text{tot}$), are 
simple to implement because they don’t produce correlations, and we can always encode their con-

versation in a single Slater determinant. Other symmetries—total spin, $S^z_\text{tot}$ being the most prominent—do induce correlations, so a single 
Slater determinant (AntiSymmState) cannot use $S^z_\text{tot}$ as a good quantum number.

In such cases, we use a correlated basis that preserves the many-body symmetry. In general, this 
would require a prediagonalization step to first get the spin eigenstates. However, the algebra of 
angular momentum—and therefore of spin—is well known, so we can write down all eigen-
states of spin for arbitrary orbital configurations, essentially relying on the appropriate products of Clebsch-Gordan coefficients.

Such a facility is provided in Diag through the $\text{spinGen}$ generic function,
\[ \text{spinGen}(\text{AntiSymmState}<\text{SPState}>, \text{config}, \]
\[ \text{int} \text{twoS}, \text{int} \text{twoSz}), \]
which returns a $\text{StateSet}$. Each element of the 
StateSet is a $\text{LinCombState<AntiSymmState}<\text{SPState}>, \]
which is an eigenstate of $S^z_\text{tot}$ with the appropriate spin quantum number. To keep 
the inputs as integers and therefore exact, input to $\text{spinGen}$ is two times the spin $S$ and two times the 
projection $S_z$. Input is also the orbital configuration as an AntiSymmState<SPState>.

**Generic Diagonalization**
The CI method eventually requires a diagonalization. Currently, Diag uses the uBLAS linear 
algebra library of the Boost project (www.boost.org), along with a bindings library that lets C++ 
directly interface with the Lapack algorithms.

We can extend this to other diagonalization routines without much trouble. The diagonalization 
function has the signature
\[ \text{vector<double>} \text{diagon}(\text{Matrix} \text{H}, \]
\[ \text{StateSet<} \text{State, Coeff}> \text{eigenVecs,} \]
\[ \text{size_t} \text{numEigs}), \]
where $\text{H}$ is a matrix. The final two arguments are optional, but if the first is provided, then the function 
calculates the eigenvalues of $\text{H}$ and places them in $\text{eigenVecs}$; otherwise, it computes only 
the eigenvalues. The final argument $\text{numEigs}$ indicates how many eigenvectors and eigenvalues to 
compute; if omitted, they’re all computed. The function itself returns the eigenvalues of $\text{H}$ in a 
vector<double>}.
To construct $H$, we would normally call `oneBodyOp` and/or `twoBodyOp` for each element. To aid in this, Diagon provides a function `matrixOp`:

$$\text{Matrix matrixOp(basis, matelem),}$$

which returns the matrix obtained by applying `matelem` to each basis vector in `basis`. This function calls `oneBodyOp` and `twoBodyOp` as appropriate.

**An Example**

Using the Diagon framework, we can set up a complete diagonalization program and run it remarkably quickly. Once we have the basic (non-generic) components, the package produces a custom-made set of classes and functions dealing with linear superpositions of many-body states at compile time. We can use these as a simple diagonalization to obtain spectra or use the eigenstates for further computations in, for example, problems of quantum dynamics and decoherence that require the actual states or when correlations in the system play an important role.

Let’s look specifically at a 2D quantum dot in a GaAs/AlGaAs heterojunction parabolically confined in the plane and in the presence of both spin-orbit interactions and a magnetic field perpendicular to the dot’s plane. We can write the Hamiltonian as

$$\hat{H} = \hat{H}_\text{qd} + \hat{H}_\text{so},$$

where the quantum dot Hamiltonian is given by two harmonic oscillators plus a Zeeman term

$$\hat{H}_\text{qd} = \hbar \Omega_+ \left( a^\dagger a + \frac{1}{2} \right) + \hbar \Omega_- \left( b^\dagger b + \frac{1}{2} \right) + g\mu_B B^2 S^z,$$

with

$$\Omega_\pm = \left[ (\omega_0^2 + 4\omega_0^2)^{1/2} \pm 1 \right]/2.$$ 

Here, $\omega_0$ is the confinement frequency characterizing the parabolic confinement, and $\omega_0 = eB/(mc)$ is the cyclotron frequency. The final term in Equation 3 is the spin–orbit interaction. We take a linearized model including both Dresselhaus and Rashba terms, given by $\hat{H}_\text{so} = \beta(S_z p_x + \sigma_y p_y) + \alpha(\sigma_x p_z - \sigma_z p_x)$, where $\sigma_k$ are the Pauli matrices, and $\beta$ and $\alpha$ are the Dresselhaus and Rashba coefficients, respectively. In terms of the Bose operators in Equation 4, we can write

$$(\hat{H}_\text{so} = \Lambda S_z [\Omega_+(\alpha a + i\beta b^\dagger) + h.c.],$$

where $h.c.$ is the Hermitian conjugate, $S_z$ is the spin raising operator, and $\Lambda^2 = (\hbar m/2)(\omega_0^2 + 4\omega_0^2)^{1/2}$. The objective in this example is to diagonalize Equation 3 in the basis of Equation 4’s eigenstates, given by the Fock-Darwin states $|nms\rangle$. Figure 1 shows a minimal function that does this with the Diagon framework; I’ve stripped the header files and additional comments here for brevity. Sec-

```cpp
int main() {
    // 1. Raw parabolic parameters (GaAs)
    double omega0 = 1; // meV
    double bField = 1; // Tesla
    Param fockdarwin = genParams(omega0, bField);

    // 2. Raw spin-orbit parameters (GaAs)
    double rashba = 0.1; // meV nm
    double dressel = 0.6; // meV nm
    ParamSO spinorbit = genParamsSO(fockdarwin, rashba, dressel);

    // 3. Basis vectors
    vector<FDState> basis(1000); // 1000 element
    GenStates gen(fockdarwin);
    generate(basis.begin(), basis.end(), gen);

    // 4. Hamiltonian
    MatElem matelemFD(fockdarwin);
    Matrix hamilFD = matrixOp(basis, matelemFD);
    MatElemSO matelemSO(spinorbit);
    Matrix hamilSO = matrixOp(basis, matelemSO);
    Matrix hamil = hamilFD + hamilSO;

    // 5. Diagonalization
    StateSet<FDState, Complex> eigenvectors;
    eigenvectors.addBasis(basis.begin(), basis.end());
    vector<double> eigenvalues = diagon(hamil, eigenvectors);
    // Print the results...
    return 0;
}
```

Figure 1. The Diagon framework in action. This example code shows the minimal `main()` function for computing spin-orbit eigenvalues and eigenvectors. There are only 20 lines of code.
tion 1 of the code simply creates parameters for Equation 4. The genParams() function takes $\omega_0$ and the external field and computes $\Omega$ and the Zeeman energy in Equation 4 via GaAs material parameters. Section 2 performs a similar task for Equation 5.

Section 3 generates the basis states with which to perform the diagonalization. The object gen() is a function created with the fockdarwin object; successive calls of gen() return the Fock-Darwin state (nms) with the next-highest energy. Thus, calling gen() 1,000 times will yield the 1,000 lowest-energy states with the given material and model parameters; the line generate(basis.begin(), basis.end(), gen); does exactly this. The generic function generate() is part of the C++ Standard Template Library (see www.sgi.com/tech/stl). It takes three arguments and returns nothing. The first two arguments delineate an array, and the third argument, which must be a function that takes zero arguments, is repeatedly called, with the results assigned to each element in the array. In this case, the net result is that the basis states are stored in the vector basis.

Section 4 of Figure 1 creates the Hamiltonian matrix, Equation 3, by using the matrixOp function. The functions matelemPD and matelemSO calculate matrix elements. MatrixOp() will call these functions using the states in the vector basis and will return the respective Hamiltonian $\hat{H}_{qd}$ and $\hat{H}_{so}$ in matrix form.

Finally, in section 5 of Figure 1, the diagonalization occurs. The first line creates an empty StateSet called eigenvectors, and the second line adds the basis states to it. The third line then calls diagon() and places the eigenvalues in the vector eigenvalues, whereas it places all the eigenstates, each an orthogonal superposition of the basis vectors, in eigenvectors. We can then print these results or otherwise process them. That’s it. We’re done.

Again, this simple example is meant for illustrative purposes only, although it is working code. Many of the functions in Figure 1 take optional arguments and support different interfaces. I haven’t explicitly mentioned much of the Diagon framework in this example. Nevertheless, it illustrates how a properly constructed generic framework can support a flexible computing environment. For example, we could easily add an elliptic anisotropy (elliptic paraboloid) to this example. We would simply need to define an additional matelem function object that computes the appropriate matrix elements, and an additional parameter object containing the eccentricities, and so forth. Each of these is probably fewer than 10 lines of code. We could then add this additional term to the Hamiltonian through the matrixOp function and then solve a whole new group of problems. Additionally, adding Coulomb interactions among the particles would proceed along essentially identical lines.

Although we have focused here on computing many-body eigenstates and eigenvalues, our day-to-day use of the Diagon framework employs little of the actual diagonalization routines. The rich set of classes in Diagon encapsulating quantum states and operators is used more as a general-purpose library for further manipulation of high-dimensional and often unwieldy eigenstates after the diagonalization phase. For example, we’re currently looking at the real-time dynamics of quantum spin and how precisely relaxation and decoherence occurs due to spin-orbit, phonon, and hyperfine interactions. Doing so requires high-precision numerical results, much higher than the standard double precision available on all present-day machines. An unexpected but happy surprise is that we were able to quickly (about a day of one person coding) plug in an arbitrary precision numerical library and still use all of our existing code base, save for a few lines in our main() function and a recompile of our code. We then replaced all double precision numbers with 300-bit precision numbers. This kind of “forward-compatible” software is, I think, one of the most powerful features of generic programming in a scientific environment.

Acknowledgments

This work is supported by the National Science and Engineering Research Council of Canada, by the Canadian Foundation for Innovation, and by a very talented group of graduate and advanced undergraduate students whom I have had the pleasure of supervising.

References

Jordan Kyriakidis is an associate professor of physics at Dalhousie University. His research interests include quantum nanoelectronics, quantum computing, and quantum dissipation and decoherence more generally. Kyriakidis has a PhD in theoretical condensed matter theory from the University of Basel, Switzerland. He is a member of the American Physical Society. Contact him at jordan.kyriakidis@dal.ca.