An object-oriented scripting interface to a legacy electronic structure code

Bahn, Sune Rastad; Jacobsen, Karsten Wedel

Published in:
Computing in Science & Engineering

Link to article, DOI:
10.1109/5992.998641

Publication date:
2002

Document Version
Publisher's PDF, also known as Version of record

AN OBJECT-ORIENTED SCRIPTING INTERFACE TO A LEGACY ELECTRONIC STRUCTURE CODE

The authors have created an object-oriented scripting interface to a mature density functional theory code. The interface gives users a high-level, flexible handle on the code without rewriting the underlying number-crunching code. The authors also discuss design issues and the advantages of homogeneous interfaces.

Object-oriented programming is widespread in almost all computing fields, but in computational physics and chemistry its use has been quite modest until recently. One reason for this lag is the dominant focus on speed and a common disregard for the user interface. Another reason is that many of the production codes in computational physics and chemistry trace their histories back to when OOP was immature. This situation is gradually changing as schemes for separating the code into low-level, numerically demanding parts and high-level steering become available.[1]

This article describes how we created an OO interface to a mature density functional theory (DFT) code. Other researchers have done this in a similar context for a molecular dynamics code in the SPaSM project,[2] but our approach differs because we don't change the underlying code. Instead, we create a Python framework around the already implemented file-based interface. A similar approach that shows how to use encapsulation of legacy code in connection with OO databases appears elsewhere.[3] Both our interface and its underlying code are available free under the GPL license.

1521-9615/02/$17.00 © 2002 IEEE

SUNE R. BAHN AND KARSTEN W. JACOBSEN
Technical University of Denmark

The Dacapo code

The Dacapo DFT code was developed at the Center for Atomic-scale Materials Physics (CAMP) to describe atomic system structure and dynamics. On the basis of a quantum mechanical description of the electronic motion, we can calculate the energy and forces acting on a collection of atoms and then use this information to determine equilibrium structures or rates of atomistic–molecular processes. The quantum mechanical laws of nature that govern electronic behavior are the same for all atomic systems. So, the code's applications cover a broad spectrum ranging from calculation of reactivity and diffusion on metal surfaces to biomolecular chemical activity.[4–6]

Quantum mechanical calculations performed in the DFT framework involve determining a set of wave functions that describe the electronic motion.[7] These wave functions are solutions to an eigenvalue equation that turns into a matrix form by expanding the wave functions on plane waves. Two factors make the problem computationally demanding: the matrices are usually

quite big (say, 10,000 × 10,000), and self-consistency is involved. The matrix that appears in the eigenvalue equation depends on the equation's solutions, but we need only the few solutions with the lowest eigenvalues. An iterative algorithm can help us obtain them.[8]

The Dacapo code originated in the 1980s and was written in Fortran 77. Recent modernization brings in Fortran 90 elements. CAMP researchers have implemented several iteration schemes along the way, and the code is still under development with an emphasis on implementing the most up-to-date algorithms.

Given the code's complexity, it should come as no surprise that the number of input parameters describing a calculation is quite large. In addition to structural information (the unit cell's shape and size, the atoms' positions and species), all sorts of more technical parameters arise: plane-wave cutoff, number of electronic bands, exchange-correlation functional, *k*-point sampling, minimization schemes, charge mixing, and so on. Dacapo's current version uses the NetCDF file format for both input and output. This means that to set up a new calculation, the user must create a file with suitable parameters. After the code is fed with this file, it will produce another file containing the output data.

From input file to OO interface

The traditional input to a simulation code is a file of parameters that control the simulation. In the case of a text file format, this could resemble

```
340.000000 Cutoff energy (eV)
     9 Number of Electronic bands
     1 Number of atoms
...
```

In our case, we use the slightly more advanced NetCDF format because it gives us random access combined with a compact and machine-independent binary format.

Supplying a file interface with some sort of graphical interface to create those files has become customary. This interface helps users by presenting only the allowed choices of input and responding to user input to avoid input inconsistencies.

This setup lacks flexibility. Although performing a single calculation is easy, a longer chain of interdependent simulations quickly becomes time-consuming. Because the latter is what we usually need in a research project, we need to control the code in a more advanced way. A typical situation encountered when using the code is the need to test its convergence by performing many series of calculations with varying energy cutoffs, unit cell sizes, and numbers of *k*-points. Another scenario could be to find the bulk modulus of a material by varying the unit cell's volume.

One of the present work's goals was to provide a high-level interface for running simulation series. By high level, we mean that the interface should offer simple constructs for common tasks. The interface should also be flexible enough to make uncommon tasks only slightly more demanding. Furthermore, we want an easily extendable structure so that new tasks can become an integrated part of the interface.

Command shell scripts facilitate more advanced control. By automating the creation of the parameter files, these scripts can carry out a series of simulations with no human intervention. This process involves four steps:

1. Create an input file
2. Run a simulation
3. Analyze the output
4. Choose whether to repeat from Step 1

Although an ordinary command shell is suitable for creating files and analyzing simple text, it is inconvenient for analyzing more advanced data. To address this problem, some simulation codes provide their own custom command line interpreter, which allows for the most common type of analysis and plotting.

Useful URLs

Center for Atomic-Scale Materials Physics
Gnuplot for Python
  http://gnuplot-py.sourceforge.net
  www.fysik.dtu.dk
NetCDF file format
  www.unidata.ucar.edu/packages/netcdf
Numerical extensions to Python
Python graphical widget sets
  http://wxpython.org
  www.python.org/topics/tkinter
  www.thekompany.com/projects/pykde
  www.pfdubois.com/numpy
Python programming language
  www.python.org
Simplified Wrapper and Interface Generator
  www.swig.org
Visualization Toolkit
  http://public.kitware.com/VTK

We want the functionality of such an interface without writing yet another *command line interface*. Fortunately, general-purpose scripting languages are available that can provide us with a CLI and let us write extensions to meet our needs, such as advanced on-the-fly analysis. By choosing Python, we can further profit from the language's heavy object orientation to easily create an OO interface. In addition to flexibility, the language's object orientation lets us access all the advantages of encapsulation, code reuse, and so other elements found in traditional OOP. This makes extensions easy to write and integrate.

For the interface to be flexible and easy to use and extend, we therefore need only to create a set of modules that make doing the most common tasks easier. We obtain a smooth transition by writing Python modules, gradually building up an interface between the original code and the user to the point where essentially all functionality is accessible through the interface. In addition to providing a simple interface for doing common tasks, this encapsulation of the original code also gives us freedom to later move from an input-file-based scheme to a more seamless integration. One example of such a tight integration is the SPaSM code at Los Alamos,[2] where researchers have used SWIG to wrap C-code so that it's directly callable from Python.

The modules

Figure 1 shows the classes used in a typical simulation. As we mentioned earlier, using Dacapo means setting up a sometimes very large set of calculational parameters. The interface takes care of this by providing a `Simulation` class, which acts as a container for any set of parameters. The parameters are added as simple Python attributes and are totally independent of each other in that they have no direct knowledge of the existence or value of the simulation object's other attributes. When the simulation class is asked to perform a calculation, it creates a NetCDF file and asks each of its attributes to write its parameters to the file. This file is unique to the calculation, so several simulations can execute simultaneously on the same computer. The NetCDF format allows independent writing of entries, so it fits well with our approach. This distributed way of creating the input file is an example of weak coupling between objects. The advantage is that we can add any kind of new object as long as it has proper methods for writing itself. Problems occur if two objects must write the same parameter. In this case, it is not clear what the user intends, so the policy is to inform the user of the conflict and abort the calculation.

The most important example of an object that goes into the `Simulation` container is `ListOfAtoms`. The `ListOfAtoms` class contains the calculation's structural information and is hence the class that is most directly manipulated by the user. Most other classes in the interface are associated with this class, so proper design is important.

To give an impression of the scripting involved in setting up a simulation, let's look at a script that sets up a simple total energy calculation for bulk magnesium using the Dacapo code:

```
from Simulations.Dacapo import *
mysim=Simulation()
mysim.bands=ElectronicBands(9)
mysim.config=ListOfAtoms(
   atoms=[Atom(Mg_GGA,Vector
       ([0,0,0]))],
   unitcell=BravaisLattice
       ([[-1.425, 1.425, 1.425],
          [1.425,-1.425, 1.425],
          [1.425,  1.425,-1.425]]))
mysim.plancut=PlaneWaveCutoff(340)
mysim.Execute()
```

The script resembles a regular input file, but a special program does not parse it—rather, it is fed into the Python interpreter. The first line tells the interpreter to load the custom-built modules we created. The next step is to create a `Simulation` container for the parameters we want to set (parameters are added in the form of ordinary Python attributes). The following lines illustrate this, where we set the number of electronic bands, define the structure we want to calculate, and set the cutoff to be used. The attribute names (`config`, `bands`, `plancut`) are arbitrary—only the object they refer to (such as `ListOfAtoms`) is important. We start the calculation by calling the simulation class's `Execute` method. Although it looks like a simple input file and certainly is not much harder to create, it's actually a small Python program that gives the user Python's full power. For instance, to test convergence with respect to plane-wave cutoff, we just change the last two lines to

```
for cutoff in [250,300,350,400]:
  mysim.plancut=PlaneWaveCutoff
    (cutoff)
```

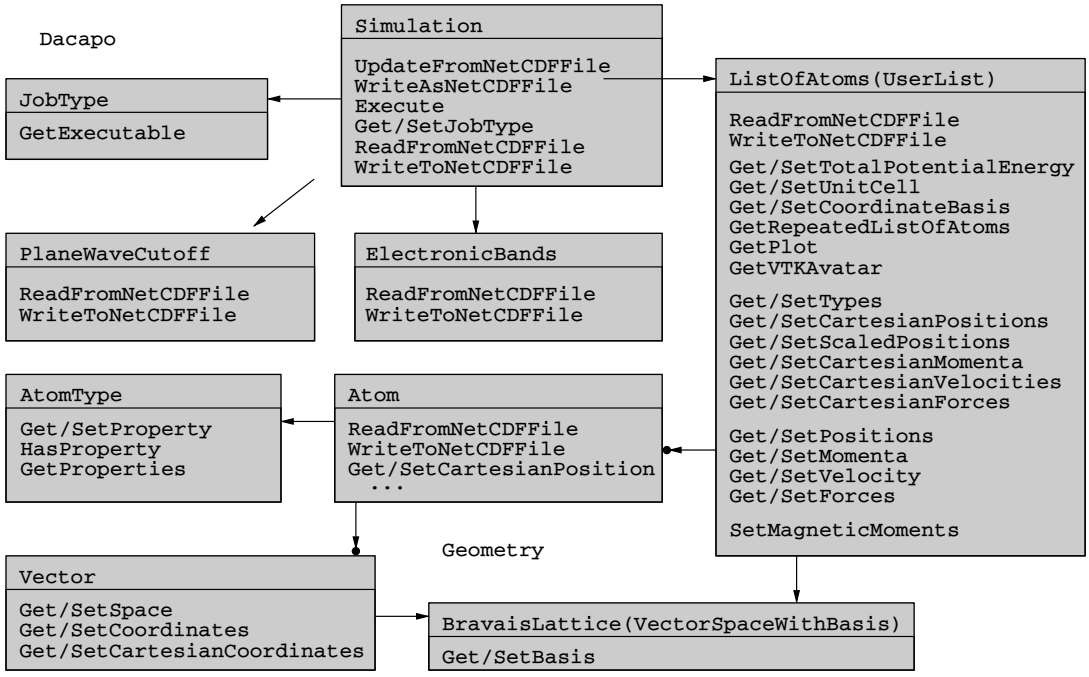

Figure 1. A class diagram. Arrows indicate that a certain class has pointers to instances of the class it points to. A circle at the end of an arrow indicates that a class can have references to more than one instance.

```
mysim.Execute()
print mysim.config.GetTotalEnergy()
```

The script now calculates and prints the total energy using four different values of the cutoff: 250, 300, 350, and 400. Extensions to this example are obvious and show how flexible a scripting interface is. There are, of course, drawbacks: Some new users might feel intimidated by having to write scripts instead of using a GUI. We can remedy this by providing a graphical interface for scripting. Python has an excellent track record when it comes to creating GUIs. Its OO nature and the many available widget sets make GUI creation a rather pleasant experience. We can create graphical representations of the objects in our interface and make the user interact with the objects through graphical widgets. In this way, the user sees the GUI not as a separate layer but as an integrated part of the interface.

Choosing objects and interfaces

An important part of the interface development was to identify which parts of the problem constitute objects and where to put interfaces between them. Our line of thinking was not to build a complex algorithm from atomic logical statements, but rather to break down the problem into smaller parts that we could solve independently.

Our guiding light in this segmentation was not computational details (the problem's syntax) but the structure of the physical theory (the problem's semantics). Our starting point was therefore to identify the physical entities that constitute the problem. For a density functional calculation, the problem's natural constituents are the atoms, wave functions, *k*-points, and so forth.

Having decided which objects constituted the problem's main constituents, the next step was to decide which interfaces these objects should have. For instance, we had to decide what makes an object worthy of the label *atom object*. Natural requirements could include methods for accessing the atom's position, momentum, and so on. Deciding which methods to require eventually set our idea of what an atom is. This process was a cardinal point in the project, and the decisions made influenced the further development substantially. We were aware that a too-narrow specification would make the framework less adaptable to other problems; a too-broad specification would give too much freedom to the implementations, eventually leading to inconsistencies. Unfortunately, any misjudgments made in this process will only reveal themselves as the

code matures, but we can take some steps to avoid the most obvious pitfalls.

What's in a name?

The discussion of naming schemes might seem trivial, but we found the choice of naming to be the most important step in the design phase. This is particularly true in an interpreted language such as Python. In compiled languages, naming is used only to ensure convenience for the programmer, whereas in the present case, the user directly refers to the names. Naming is therefore not easily changed, because this would break backward compatibility.

It seems obvious that an atom object should have a `GetPosition` method to provide information on the atom's placement. However, an atom can have its position specified in different ways: Cartesian coordinates, polar coordinates, scaled coordinates, or even more exotic specification, such as the distance from a central molecule. How do we make sure that the interface we provide does not hinder important future improvements, yet is concrete enough for immediate implementation?

Specifying the interface using abstractly defined objects to avoid specifying a too-restricted structure is tempting. A problem with this approach is that often we need a simple set of numbers to specify the position rather than some abstractly defined position object. If every data structure passed around is an object, we never get concrete, simple, data type handles on the objects and end up in a vicious "everything is an object" circle.

The solution we propose is to provide several interfaces, reserving the general names for the general interface and creating descriptive names for the more specific interfaces. Instead of just having one interface, we construct several:

- `GetPosition` should return a position object with suitable interfaces.
- `GetCartesianPosition` should return the position in the Cartesian frame of reference.
- `GetScaledPosition` and `GetCoordinateBasis` should return a set of coordinates and a basis in which these are given (relative to the Cartesian basis).

We give the methods descriptive names that uniquely specify what the interface provides. Hence, specific methods tend to have long names, describing in detail what to expect from the method, and more general methods tend to have shorter names that vaguely indicate their use. In this way, we avoid giving a specific interface a name that ought to be reserved for a more general one.

At first, this naming scheme might seem a bit clumsy, involving the typing of long method names for basic instructions, but the payoff is multifaceted. We have already seen how this scheme makes it possible to both have our cake (the possibility of future extensions) and eat it (concrete, ready-to-implement interfaces). Another nice feature is that using long names creates more or less self-documented code. Moreover, consistency in naming makes it easier for users to guess method names, and the descriptive character hinders them from making incorrect assumptions about the methods.

Developing in Python

For Python, with its dynamic type checking of objects, there really is no difference between class inheritance and interface inheritance. Consequently, we can claim that the only real class hierarchy is the one a consistent naming scheme provides.

The two forms of inheritance, direct class inheritance and inheritance by name, are indistinguishable from a user's viewpoint, but they are used somewhat differently in the code. General types tend to be inherited from abstract classes through simple class inheritance, whereas more specific types are realized directly by implementing the class's proper methods. This difference between general and specific types is also reflected in the order in which we implement the interfaces. Specific interfaces are really not meaningful without a concrete implementation, but a general type need not have a concrete implementation and can remain abstract until it is actually used in a concrete setting. This lets us postpone implementing large parts of the interface until we actually need that functionality.

Another aspect of Python's dynamic type checking is that it invites prototyping and top-down design. Although the strict class hierarchy of C++ is useful in enforcing a certain design, the Python approach is outstanding when it comes to writing working code right away, avoiding the danger of spending too much time in building a complete foundation. Once we pass the design phase, we can start writing code that uses the defined interfaces even though they might be only partially implemented. One typical situation in

which this is an advantage is when we not only know what would in theory be a good solution to a given problem but also realize that fully implementing this solution would be time-consuming. Instead of settling for a less satisfactory but more readily implemented solution, we can choose the ambitious solution and implement only the specific part we need. We can even define new interfaces as the need arises, which fits into the rest of the framework, provided we follow the naming scheme. This is important in an environment as dynamic as scientific research. Any research project needs its own features, and foreseeing all future needs and implementing them from the beginning would be impossible. It is therefore convenient to be able to postpone implementing and defining the interface until we actually use it.

An important part of our interface development is the emphasis on creating modules that different people can develop independently of each other. To ensure this modularity, we keep each class as weakly coupled to the rest of the structure as possible. This means that all assumptions in the code about objects should be explicit by referring to a suitable specific method. Imagine, for instance, a visualization tool such as a 3D atom plotter. To make sure that it will work with any kind of collection of atoms (for example, `ListOfAtoms`), we must avoid making implicit assumptions about the object, such as in which coordinate system the positions are given. We can avoid this by referring only to well-specified interfaces such as `GetCartesianPositions`, so that any object with the required methods will work equally well. This means that taking full advantage of polymorphism is easier and making extension by adding new classes and features is less painful.

Another advantage is the low barrier of entry for new developers. The step from using the interface to extending it should not be demanding. One step toward achieving this is the modularity discussed earlier, which ensures that new developers have to understand only the part of the interface they want to extend. But even this task can be demanding if the interface is somewhat complex. A complex framework's advantage is that it makes getting a lot of functionality from only writing a few lines of code possible. This leaves us with a dilemma: on one hand, we would like a simple interface that lets new developers get a quick start, but on the other, we would like developers with a more complete understanding of the interface to get the full benefit of a complex framework. Once again, using multiple interfaces can solve our troubles. An inexperienced developer can start by providing simple interfaces to the object. As his or her level of understanding increases, the developer can start working with the more general interfaces and use the framework to its full advantage. In this connection, we can use the suggested naming scheme as a pedagogical tool. The long names for the specific interfaces remind the new developer of the existence of the more general interface with shorter names. So, the developer doesn't stay too long with the simple part of the interface when the complex part would be a greater help.

Many of the problems and guidelines discussed so far are known from other areas of OOP. During our development of the interface, we found it useful to identify which design patterns covered the various objects we needed to implement.[9] A *design pattern* is the part of a problem solution that is reusable across different problems with common features. For instance, we can describe the object adapter pattern (see Figure 2) as a way to make objects with a foreign interface—such as a third-party object—cooperate with the code by wrapping it with a new interface. Later we use this to combine two simulation techniques. Usually one or more patterns fit quite well to a given class, and acknowledging this can ease the discussion and communication of that class's properties.

Multiple codes

When dealing with interface design, it helps to have other back-end codes for comparison. Ensuring that the classes and hierarchy make sense as an interface for another slightly different code assures a useful level of abstraction.

Two of our colleagues at CAMP built another code that deals with the same area of physics, using the effective medium theory (EMT) instead of the DFT to calculate energies.[10] They constructed a Python interface for this code at the same time as the present work; reconciling differences between the two approaches was a great source of inspiration.

The EMT code can handle millions of atoms but with no electronic degrees of freedom; the DFT code applies only to small systems but with high precision and detail in the description. Nevertheless, some systems overlap the two approaches and benefit from both.

Consider an EMT simulation that has found the optimal structure for some collection of

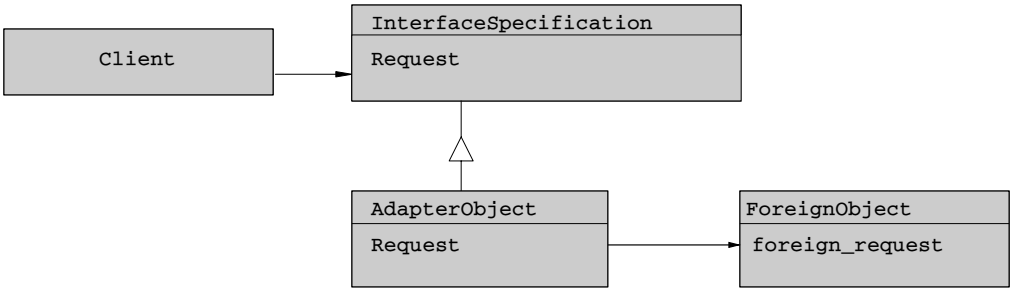

Figure 2. The object adapter pattern. The diagram shows how a `ForeignObject` can adapt to a new environment by letting clients access it through an `AdapterObject` with an interface inherited from an abstract `InterfaceSpecification`. The triangle indicates that the adapter inherits the interface from the class it points to. The arrows indicate that the `AdapterObject` keeps a reference to the `ForeignObject` and is itself referred to by the client.[9]

atoms. To further improve the precision, we would like to continue the calculation by using a DFT energy calculator. In the ideal case, we can simply extend the script needed for running the EMT calculation with a few lines to complete it. To achieve this goal, we need a convenient way of using the configuration from the EMT calculation in the DFT code. If we carefully chose the interfaces, this should be straightforward. After all, both codes should agree on the meaning of such methods as `GetCartesianPosition` and `GetChemicalElement`. Because Python does no type checking, we can use the class of a configuration from one interface code in another setting as long as the method names are the same. So, if we assume that `emtconfig` is a list of atom instances created by the EMT interface with the methods needed to qualify as a `ListOfAtoms`, we can simply use something such as this:

```
...
from Simulations.Dacapo import *
mysim=Simulation()
mysim.dftconfig=ListOfAtoms
   (emtconfig)
mysim.bands=ElectronicBands(9)
mysim.plancut=PlaneWaveCutoff(340)
mysim.Execute()
...
```

These statements are similar to what we use for setting up a regular DFT calculation, except that the structure is derived from the foreign `emtconfig` object. This direct use of one object in another setting is a good example of polymorphism and is a most desired way of bridging between codes with otherwise different internal data structures. This example also illustrates the technique of recasting; we can view `emtconfig` as the EMT version of a `ListOfAtoms`, and using `emtconfig` as an argument to the class constructor, we can create a DFT `ListOfAtoms` with the extra methods needed for the `Simulation` object to use it. Recasting is required if an object does not have all the methods needed in a particular setting. A convenient way of recasting is, as in the previous case, to have the class constructor take a similar object as an argument and work along the lines of the object adapter pattern (see Figure 2).[9]

The interface right now: An example

Recent simulations[11,12] and experiments[13,14] show that in the process of breaking a piece of gold, we can create wires consisting of a single row of gold atoms (see Figure 3). These nanowires possess interesting electrical and mechanical properties. The conductance is, for example, "quantized" and close to the value $2e^2/h$, where e is the electron charge and h denotes Planck's constant.[13]

To study the breaking of such an atomically thin wire, we can perform a series of calculations in which we stretch a nanowire in small steps until it eventually breaks. Figure 4 is a script for performing such a calculation with a small unit cell containing two gold atoms. The unit cell is periodically repeated, so the calculation is really for an infinite string of atoms.

Without explaining the script in detail, we see a few interesting features. The script is made flexible by introducing the variable `numberofatoms`, which we can change to perform calculations with a different number of atoms in the unit cell. We use the `NetCDF.Entry` class to control simple parameters in the simulation. This class

is a generic class for entries in a NetCDF file and is the base class for the `PlaneWaveCutoff` and the `ElectronicBands` classes. Introducing classes for such simple control parameters by inheritance from the `NetCDF.Entry` class is easy.

The Python interface includes many more classes than Figure 1 shows. Many of them are targeted toward visualizing the huge amount of data that results from a simulation. We can easily perform simple curve plotting by using a Python interface to Gnuplot, and we can obtain more advanced data visualization through the Visualization Toolkit's (VTK) Python bindings. To illustrate these possibilities, let's look at a script that analyzes the previous simulation's result (see Figure 5).

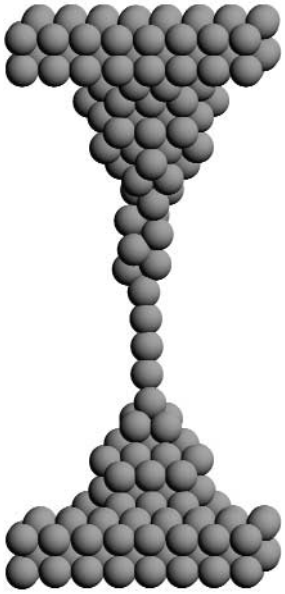

Figure 3. The formation of a chain of gold atoms. The structure is a snapshot from a molecular dynamics simulation using the EMT code.

```
from Simulations.Dacapo import *
from RandomArray import random

numberofatoms=2

sim=Simulation()

# setting up the entry describing how to generate k-points
sim.kpoints=NetCDF.Entry("KpointSetup,"[1,1,16/numberofatoms])
sim.kpoints.gridtype="MonkhorstPack"

# setting up electronic bands
sim.bands = ElectronicBands(11*numberofatoms/2+10)

# setting up the atomic configuration

ucell=BravaisLattice([[11,4,0],[4,11,0],[0,0,2.5*numberofatoms]])
basis=ucell.GetBasis()
sim.atoms=ListOfAtoms(unitcell=ucell)
for nr in range(numberofatoms):
   sim.atoms.append(Atom(Au,Vector([0,0,nr*2.5])))

# various other setup
sim.cutoffenergy=PlaneWaveCutoff(340.145000) # unit is eV
sim.dyn=NetCDF.Entry("Dynamics")

# loop over unitcell size and shake atoms to break symmetry
for i in range(12):
   ucell.SetBasis(basis+
        [[0,0,0],[0,0,0],[0,0,i*.0625*numberofatoms]])
sim.atoms.SetCartesianPositions(
   random((len(sim.atoms),3))*.1+
   sim.atoms.GetCartesianPositions())
sim.Execute("out%i.nc"% i)
```

Figure 4. A script for performing a series of calculations in which a gold nanowire is stretched in small steps.

Figure 5. A script to plot wave functions and energies from the simulation scripted in Figure 4.

```
from Simulations.Dacapo import *
from Simulations.Dacapo.ListOfEigenStates import ListOfEigenStates
from Visualization.Avatars.vtkListOfAtoms import vtkAtoms
import Gnuplot

sim=Simulation()
sim.atoms=ListOfAtoms()

energy=[]

for i in range(12):
   sim.loe=ListOfEigenStates()
   sim.UpdateFromNetCDFFile("out%i.nc"% i)
   # monitor energy
   energy.append([sim.atoms.GetUnitCell().GetBasis()[2,2],
            sim.atoms.GetTotalPotentialEnergy()])
   if i==0:
       waveplot=sim.loe.GetEigenStates(bands=[11],
            kpointnumbers=[0])[0].GetVTKAvatar(
                contourvalues=[2.5])
   waveplot.SetTranslation([-36,-36,0])
   waveplot.SetPeriods([1,1,3])
   atomplot=vtkAtoms(sim.atoms,parent=waveplot)
   atomplot.SetPeriods([1,1,3])
   else:
       waveplot.Update(sim.loe.GetEigenStates(bands=[11],
            kpointnumbers=[0])[0])
   waveplot.Render()
   waveplot.SaveAsBMP("wave%i.bmp" % i)

p=Gnuplot.Gnuplot()
p.plot(Gnuplot.Data(energy,with="linesp"))
p.hardcopy("energy.eps")
```

The script illustrates how we use the interface in connection with other modules to read the data files (using the `UpdateFromNetCDFFile` method) and plot the result. We monitor the energy using Gnuplot and see one of the wave functions close to the Fermi energy. The few lines of code that generate the wave function visualization "hide" a rather elaborate setup of the complex scalar field's VTK plot. Both VTK and Gnuplot are examples of the many useful third-party tools with Python interfaces (see Figure 6).

From the plots, we get an idea of what happens electronically as the wire stretches and breaks. For a small stretch, the atoms form a zigzag structure, and the wave function close to the Fermi level follows it. Further stretching makes the wire straighter, and the energy increases. At some point, the chain breaks owing to a Peierls instability. We can see this in the energy curve as an inflexion point and in the wave function as a dimerization. If we had plotted the band structure, we would have seen a band gap appear exactly at this point. This shows how the visualization modules can help us analyze and understand a system's properties. We can use these modules both in scripts as described earlier and in interactive sessions where we can rotate structures and zoom in on interesting parts.

The interface continues to grow because some of the Dacapo DFT code's functionality is migrating to the Python interface. The Dacapo code contains several different possibilities for moving atoms around with the purpose of locating equilibrium structures, finding reaction pathways, or studying time evolution. We found

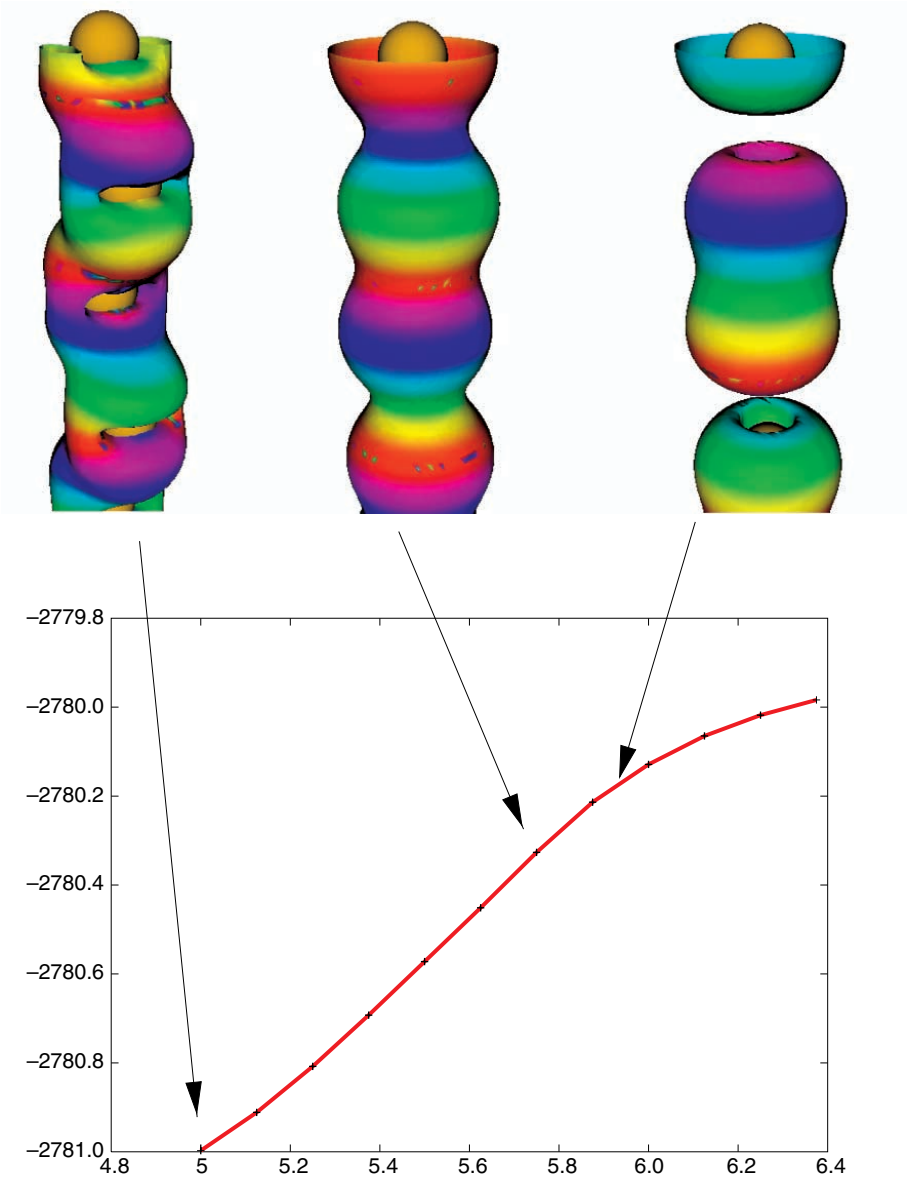

Figure 6. Wave function and energy plot when stretching a chain of atoms. The plot was generated by the script by using VTK and Gnuplot modules. We created the wave-function plot by taking a suitable isosurface of the absolute square of the wave function and coloring it using the complex phase.

it easier to develop new atom movers in Python and to control atomic motion in the Dacapo code externally. This helped us avoid tedious recompilation of the Dacapo code and let us use Python's flexibility to do rapid prototyping of new experimental algorithms. If we use the Numeric package with its Python bindings for numerically efficient array operations, we can make the Python atom movers so efficient that they become attractive even when handling millions of atoms. This is not important for the Dacapo interface, but it gives us the possibility to use the same atom movers to control our classical molecular dynamics code based on the EMT potentials. The Numeric package is also crucial for handling the large amounts of data necessary to describe electronic wave functions or densities.

Hopefully, some of the general considerations we've presented will help others constructing similar interfaces for their codes. In several fields, integrating different methods and codes is becoming a key issue. In materials science, many important problems naturally involve phenomena ranging from the nanometer scale to micrometers or longer. Several research groups are working on the integration of electronic, atomistic, and finite-element descriptions aimed at solving such multiscale ma-

terials problems.[15–17] In molecular biochemistry, this integration of different methods and codes is also an important topic. For example, quantum mechanical approaches combined with more approximate force-field-based molecular modeling techniques can help explain the function of enzymes.[18,19]

The development of multiscale–multimethod approaches should benefit as much as possible from already existing mature codes. The Python scripting interface to our DFT code has certainly made the code more accessible to integration with other codes, especially if they carry similar interfaces. The interface between two codes is easiest if the separate Python interfaces agree on common method names, as is the case with our DFT and EMT codes. However, even without a complete match, it is our experience that communicating information between codes using Python is rather straightforward. OO seems to us a passable way of constructing larger complexes of interacting codes.

Our interface is still under development, and we aim to keep it that way as long as there are users for it. Anyone who is interested in using or even contributing to the interface can download the software from the CAMPOS Web page, www.fysik.dtu.dk/CAMPOS, which also has further information and help.

Acknowledgments

Many thanks to Asbjørn Christensen, Bjørk Hammer, Lars B. Hansen, Jens Jorgen Mortensen, Chris Myers, Ole H. Nielsen, Jakob Schiøtz, and James P. Sethna for their invaluable guidance and helpful discussions. The Danish National Research Foundation sponsors CAMP.

References

1. P. Dubois, "Making Applications Programmable," *Computers in Physics*, vol. 8, no. 1, 1994, pp. 70–73.
2. D.M. Beazley and P.S. Lomdahl, "Controlling the Data Glut in Large-Scale Molecular Dynamics Simulations," *Computers in Physics*, vol. 11, no. 3, 1997, pp. 230–238.
3. D. Maier and J.B. Cushing, "Treating Programs as Object: The Computational Proxy Experience," *Deductive and Object-Oriented Databases*, Springer-Verlag, Berlin, 1993, pp. 1–12.
4. B. Hammer and J.K. Nørskov, "Why Gold Is the Noblest of all the Metals," *Nature*, vol. 376, no. 6537, 1995, pp. 238–240.
5. S. Horch et al., "Enhancement of Surface Self-Diffusion of Platinum Atoms by Adsorbed Hydrogen," *Nature*, vol. 398, no. 6723, 1999, pp. 134–136.
6. T.H. Rod and J.K. Nørskov, "Modeling the Nitrogenase FeMo Cofactor," *J. Am. Chemical Soc.*, vol. 122, no. 51, 2000, pp. 12751–12763.
7. P. Hohenberg and W. Kohn, "Inhomogeneous Electron Gas," *Physical Rev.*, vol. 136, no. 38, Nov. 1964, pp. B864–B871.
8. C. Bendtsen, O.H. Nielsen, and L.B. Hansen, "Solving Large Nonlinear Generalized Eigenvalue Problems from Density Functional Theory Calculations in Parallel," *Applied Numerical Mathematics*, vol. 37, nos. 1–2, 2001, p. 189.
9. R. Johnson et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, Reading, Mass., 1995.
10. K.W. Jacobsen, J.K. Nørskov, and M.J. Puska, "Interatomic Interactions in the Effective-Medium Theory," *Physical Rev. B*, vol. 35, no. 14, May 1987, pp. 7423–7442.
11. M.R. Sørensen, M. Brandbyge, and K.W. Jacobsen, "Mechanical Deformation of Atomic-Scale Metallic Contacts: Structure and Mechanisms," *Physical Rev. B*, vol. 57, no. 6, Feb. 1998, pp. 3283–3294.
12. G. Rubio-Bollinger et al., "Mechanical Properties and Formation Mechanisms of a Wire of Single Gold Atoms," *Physical Rev. Letters*, vol. 87, no. 2, July 2001, pp. 26101–26104.
13. A.I. Yanson et al., "Formation and Manipulation of a Metallic Wire of Single Gold Atoms," *Nature*, vol. 395, no. 6704, 1998, p. 783.
14. H. Ohnishi, Y. Kondo, and K. Takayanagi, "Quantized Conductance through Individual Rows of Suspended Gold Atoms," *Nature*, vol. 395, no. 6704, 1998, pp. 780–783.
15. A. Nakano et al., "Multiscale Simulation of Nanosystems," *Computing in Science & Eng.*, vol. 3, no. 4, July/Aug. 2001, pp. 56–66.
16. G.S. Smith, E.B. Tadmor, and E. Kaxiras, "Multiscale Simulation of Loading and Electrical Resistance in Silicon Nanoindentation," *Physical Rev. Letters*, vol. 84, no. 6, Feb. 2000, pp. 1260–1263.
17. C.R. Myers et al., "Digital Material: A Framework for Multiscale Modeling of Defects in Solids," *Material Research Soc. Symp. Proc.*, Materials Research Soc., Boston, 1999, pp. 509–538.
18. P. Amara and M.J. Field, "Hybrid Methods for Large Molecular Systems," *Computational Molecular Biology*, Elsevier Science, Netherlands, 1999.
19. K. Hinsen, "The Molecular Modeling Toolkit: A New Approach to Molecular Simulations," *J. Computational Chemistry*, vol. 21, no. 2, 2000, pp. 79–85.

Sune R. Bahn received a PhD in physics from the Technical University of Denmark in Copenhagen, Denmark. His research interests include the properties of metallic nanowires using quantum mechanical methods and electronic structure codes. Contact him at CAMP, Dept. of Physics, DTU, Bldg. 307, DK-2800 Kongens Lyngby, Denmark; bahn@fysik.dtu.dk.

Karsten W. Jacobsen is a professor of condensed matter physics in the Center for Atomic-Scale Materials Physics at the Technical University of Denmark in Copenhagen. His research interests include the theoretical description of mechanical, electrical, and chemical properties of nanostructures and nanostructured materials. He received a PhD in physics from the University of Copenhagen. Contact him at CAMP, Dept. of Physics, DTU, Bldg. 307, DK-2800 Kongens Lyngby, Denmark; kwj@fysik.dtu.dk.