Creating a GUI for Zori, a Quantum Monte Carlo Program

Rappture is a new GUI development kit that enables developers to build I/O interfaces for specific applications. In this article, the authors describe the Rappture toolkit's use in generating a GUI for the Zori computer code.

In their research laboratories, academic institutions produce some of the most advanced software for scientific applications. However, this software is usually developed only for local applications or for method development. In spite of having the latest advances in a particular field of science and engineering, such software also often lacks adequate documentation and is therefore difficult for anyone to use other than the code developers. Typically, as the codes become more complex, so do the input files and command statements necessary to operate them. To address this, many programs offer the flexibility of performing calculations based on different methods that have their own set of specific variables and options, but certain options can be incompatible with each other. For this reason, users outside the development group might be unaware of how the program runs in detail, thus developers can miss the opportunity to make the software readily available outside the laboratory of origin. This is a long-standing problem in scientific programming.

Rappture—Rapid Application Infrastructure[1]—is a new GUI development kit that lets developers build an I/O interface for a specific application (https://developer.nanohub.org/projects/rappture). This capability enables users to work only with the generated GUI and avoids the problem of needing to teach code details. Furthermore, it explicitly specifies the required variables, reducing input errors.

The Lester group at the University of California, Berkeley,[2] developed Zori, a quantum Monte Carlo (QMC) program that's one of the few free tools available for this field. Like many scientific computer packages, Zori suffers from the problems described earlier: potential users outside the research group have acquired it, but some find the code difficult to use. Furthermore, new Lester group members usually must take considerable time learning all the options the code offers before they can use it successfully. In this article, we describe how the Rappture toolkit eases this problem by generating a GUI called Zopi (Zori Processing Interface) for the Zori computer code.

Zori Basics

Zori implements the QMC method—which simulates quantum systems by solving the many-body Schrödinger equation—for calculating the electronic structure of atoms and molecules. A

1521-9615/09/$25.00 © 2009 IEEE
Copublished by the IEEE CS and the AIP

Roberto Olivares-Amaya, Romelia Salomón-Ferrer, William A. Lester Jr., and Carlos Amador-Bedolla
University of California, Berkeley

detailed description of Zori's structure and capabilities as well as the fundamentals of QMC appear elsewhere (www.zori-code.com).[3–8] Later, we present a general description of the necessary steps to perform a typical QMC calculation to help explain the method's complexity and how the GUI addresses this process.

An external quantum chemistry package typically generates an initial wave function Ψ, which then facilitates a QMC electronic structure calculation. For this purpose, the user optimizes a geometry using the quantum chemistry package, and then the quantum chemistry package, based on the optimized geometry, determines Ψ_T, where the subscript labels the specified geometry's particular wave function. Researchers typically use the GAMESS[9] and ADF benchmarks to determine Ψ_T.[10] These programs then generate the output, and the user transforms them into Zori input format. A QMC calculation stochastically probes configuration space with entities called walkers guided by Ψ_T. Each walker represents all the system's particle positions, and the initial set of walkers is generated at random. Next, the algorithm can proceed along two paths: variational Monte Carlo (VMC) or diffusion Monte Carlo (DMC). Because DMC gives more accurate results, but requires more computer time than VMC, researchers tend to make a VMC run first to relax the initial configurations and then optimize the parameters of Ψ_T, followed by a series of DMC calculations to obtain statistically useful results. A module called Zopt links to the optimization code Opt++ and optimizes Ψ_T in Zori (http://csmr.ca.sandia.gov/opt++).

The way Zori works is reasonably straightforward, but the input files can be confusing. The user inputs information using XML formatted files that contain information about execution time, algorithm type, the number of walkers per processor, and so on. However, at execution, different options useful for code development require specified parameters that aren't pertinent to applications. Moreover, users typically aren't fully aware of all the options or variables that they must determine.

Periodically—that is, for some fixed number of MC iterations—Zori writes a walker file to disk with the most recent walker positions. With either DMC or VMC runs, Zori continuously updates a binary file called `zori-energy.out` that contains the energy list of all the positions the walkers probe. Because Monte Carlo calculations normally run for long periods, it's customary to periodically probe how the energy fluctuates during a run. A tool included with Zori called Zavg takes the list of energies from the file, calculates the mean, variance, and statistical error, and saves these quantities in a text file, which programs such as GNU-Plot (www.gnuplot.info) can easily visualize.

A Look into Rappture: Wrapping Applications

Rappture is a powerful tool that scientific programmers can use as a wrapping tool if the target program has already been created, or incorporate directly into the application. Because of its special flexibility, Rappture can link to programs written in multiple computer languages. Developers can also write the interface between Rappture and the application program in various languages, including C, Python, Tcl, and Matlab.

Michael McLennan developed Rappture as part of the Network for Computational Nanotechnology (NCN), "a multi-university initiative that was established to create a resource for nanoscience and nanotechnology."[11] However, Rappture's impact has been broader than the nanosciences, with applications that include NEMO 3D, a 3D nanoelectronic modeling tool; an interface for the Spice simulator; a general-purpose circuit simulation program; and more recently, a cyclic peptide ion channel simulator.[12]

Rappture comes with two libraries: one is a unit conversions library that makes huge unit flexibility possible, especially while treating diverse energy and length units; and the other is Rappture Library, which contains the objects necessary to develop the program's GUI. In practice, the programmer must construct two simple files to run Rappture. The first file is `tool.xml`, which contains all the information necessary for developing the GUI. The developer can include several XML elements in the GUI construction, such as strings, meshes, clouds, images, and options. This XML format is extremely simple to use; the Rappture wiki page (https://developer.nanohub.org/projects/rappture) offers extensive information on how to do it. The XML file also contains a link to the second file, called the tool executable, which interprets XML elements and hands them over to the GUI. The executable also takes the user's given information in the GUI and sends it as input to the application.

A general example of an XML tool file looks like this:

```
<?xml version="1.0"?>
<run>
```

```
<tool>
<about> Welcome to ZoPI</about>
<command>python @tool/zopi.py @
   driver</command>
</tool>
<input>
<boolean id = "rn_psi">
<about><label>Yes or No?</label>
<description>Please
choose</description>
</about>
<default>no</default>
</boolean>
</input>
<output>
...(other Rappture XML elements)
</output>
</run>
```

This example demonstrates how the developer should generally structure the `tool.xml` file for a given application, which starts by opening the main element—the `run` instruction—followed by opening the `tool` command. In the command line, the programmer then specifies the tool executable's location—in this case, it's `zopi.py` and is located in the same directory as the `tool.xml` file, which is represented as `@tool/`. Then, all the elements the GUI must display as input are now specified (we've shown a Boolean declaration as an example). The tool executable can be either a script that works as an interface between Rappture and the application "wrapping" it, or it can be immersed in the application itself. The programmer can code this file with several programming languages, which gives Rappture increased flexibility for scientific programmers who are accustomed to a particular language. In the next section, a brief description of the file generated to develop Zopi, the Zori Processing Interface, is presented.

Zopi, the Zori-Processing Interface

Learning how to use Rappture presents minimal difficulty,[15] which is one of its advantages. However, we needed to adapt Rappture's capabilities to Zori's needs. Figure 1 shows a common set of steps that a user would follow to run Zori.

Zopi must have a linear structure composed of separate sections that follow the algorithm in Figure 1. QMC is well known to have the benefit of being embarrassingly parallel, so it's common for users to run applications on more than one processor on either local clusters or large multiprocessor systems. This means that Zopi must have an additional section that controls the necessary options for generating the proper scripts for submitting the job to a particular computer system's queue. The submission of a job to a single processor is likewise desired, so Zopi must also contain this flexibility.

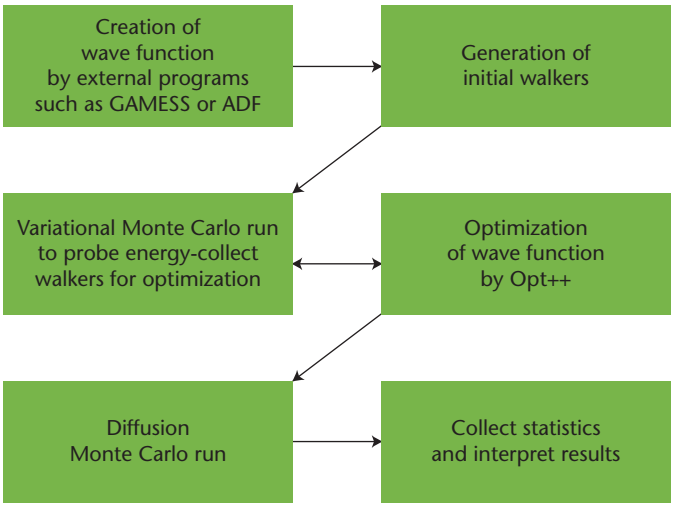

Figure 1. Flow diagram of a Zori run. This figure shows a common set of steps that a user would follow to run Zori.

Rappture offers a system that includes a tab interface in which each tab controls a procedure step—the last tab addresses the submission section. It also includes the option of running all the steps—that is, walker generation and VMC and DMC runs sequentially or one at a time.

We've included six tabs in Zopi:

- *Trial Psi* writes the wave function's information for the Zori format to use.
- *Walkers* creates an initial set of walkers.
- *VMC* drives a VMC run.
- *DMC* drives a DMC run.
- *Optimization* calls OPT++ to optimize the correlated wave function.
- *Submit* sends the job to a cluster or runs interactively on a single processor.

The development of tabs through Rappture is specified in the `tool.xml` file. Here's an example of the corresponding instructions:

```
<group id = "WT">
<group id = "PSI">
<about><label>Psi</label>
<description>Creation of the
   wave function and orbitals.xml
   files</description>
</about>
```

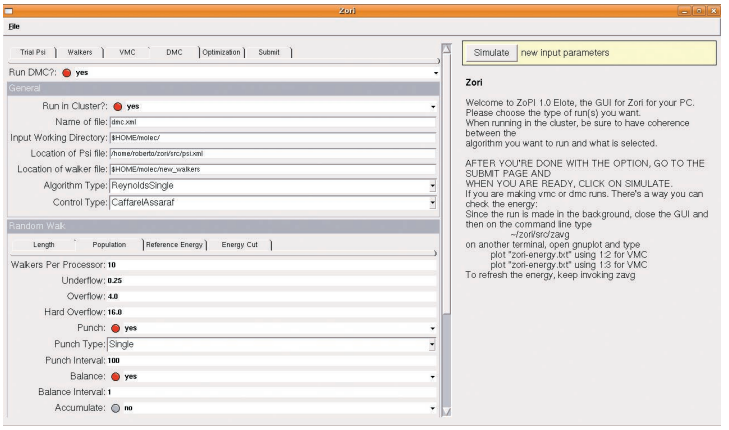

Figure 2. Diffusion Monte Carlo (DMC) tab layout. This screenshot shows three sections: the run instruction, which activates it; the general subsection, which specifies the necessary input files' path and the type of DMC algorithm currently being used; and, finally, a random-walk section.

```
...(Rappture XML elements)
</group>
<group id = "CW">
...
</group>
...
</group>
```

The first group's ID encapsulates all the tabs. The second group's ID is for the first tab, Trial Psi, in which the user prompts the rest of the section's input. The list can continue, and, if so, the tab length automatically changes.

Although similar in structure, it's important to look at other tabs to gain insight into one of Rappture's advantages—namely, compartmentalization. Figure 2 shows a DMC tab section. Other options exist in the random walk section that this figure doesn't show.

Figure 2 shows three sections of the DMC tab: the run instruction, which activates it; the general subsection, which specifies the necessary input files' path and the type of DMC algorithm currently being used; and, finally, a random-walk section. In the random-walk section, more tabs appear that the user must fill out completely with required information. An additional advantage with this approach is that users have a good idea of what they're filling out and therefore can acquire further insight into Zori operations and QMC in general.

The optimization tab enables the user to run the wave function parameters' optimization algorithms. The optimization algorithm, OPT++, improves Zori's performance, as the different parameters improve the wave function's form that Zori needs to use. This tab functions like the others, but it calls the module Zopt as opposed to the main Zori program, which will be evident in the command line itself.

As mentioned, Zopi has an additional tab that provides the option to run QMC calculations on a computer cluster. The user can also specify command-line-specific instructions to run Zori efficiently. This tab's design favors users at the University of California, Berkeley and the Lawrence Berkeley National Laboratory (LBNL) because it explicitly includes typical cluster specifications present at these institutions. However, the program also provides custom cluster script prompts. Furthermore, Rappture's flexibility makes a rapid adaptation of Zopi possible, which lets the user create a customized template for the chosen cluster. As stated earlier, either the user or the developer must edit the XML file and then edit `zopi.py` where the actual command stands. For instance, for the Lester group cluster, we created a function (see Figure 3) that asks the user for the necessary information, such as the number of processors the user will use, the Zori executable's location, and so on.

The Tool Executable

The interface between the core application—namely, Zori—and the GUI is a tool executable called `zopi.py`. This wrapper communicates to the program by transferring the information the user enters into the GUI. Rappture offers the flexibility of hard-coding these instructions into the legacy code or writing them as an external wrapper, which lets the developer *impose* a GUI on an existing legacy code.

Rappture makes it easy to write tool executables because it includes libraries of different programming languages. Although Zori is mainly written in C, the present tool executable is a Python script, which offers usability and helpful XML-related libraries. The following lines of code show an example of how `zori.py` extracts a value out of a particular GUI entry and stores it in its corresponding variable:

```
# Creation of .xml file for running
VMC in Zori
import sys
import Rappture
driver = Rappture.library(sys.argv[1])
rnVMC = driver.get('input.group.(VMC).
(rnVMC).current')
```

```
if rnVMC ='yes':
...
wppv = driver.get('input.group.(VMC).
(rdmwalk).(vls).(wppv).current')
punchv = driver.get('input.group.
(VMC).(rdmwalk).(vls).(punch).
current')
...
```

Looking at the variable `wppv` (walkers per processor for a VMC calculation), the developer can appreciate how the code obtains the current value in the slot labeled `wppv`—that is, within the subsection `vls`, which in turn is in the subsection `rdmwalk` that belongs to the tab VMC.

As mentioned earlier, Zori uses a file in XML format as input. After we obtain the variable's values that we'll use from the GUI, we use the library `libxml2` as an XML parser to generate the necessary input files:[15]

```
docv = libxml2.newDoc("1.0")
popv = docv.newChild(None,
"Population", "\n")
popv.setProp("WalkersPerProcessor",
   wppv)
popv.setProp("Punch",punchv)
docv.saveFile(namev)
docv.freeDoc()
```

This example illustrates how to start writing an XML file with `libxml2` and the command `libxml2.newDoc`. The command `newChild` indicates the creation of a new branch or section in the XML tree, which would correspond to the following resulting input file's statement:

```
<Population></Population>
```

After creating the new section, the programmer specifies the necessary properties using the command `setProp`. In the example, the properties `WalkersPerProcesor` and `Punch` are defined with the values `read` from the GUI, `wppv` and `punchv`, respectively. If the corresponding values entered in the GUI for these two variables are `10` and `True`, then `libxml2` parses the following line into the following input file:

```
WalkersPerProcessor="10", Punch="True"

Overall, this example should produce
an input file that reads as follows:

<?xml version="1.0" ?>
```

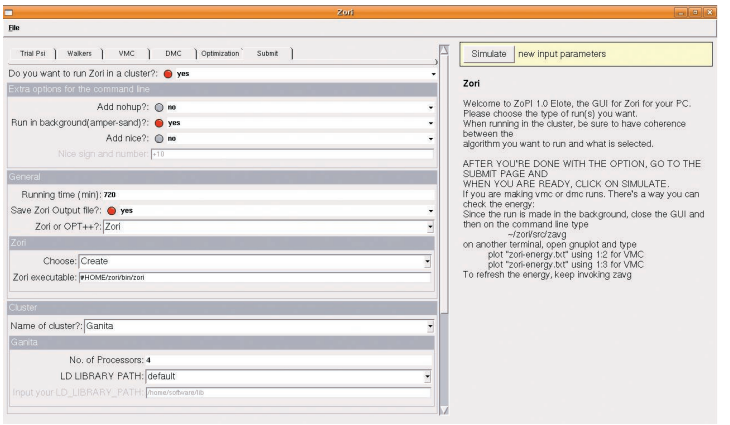

Figure 3. Submit input tab. The user sets information concerning the cluster, what to run, and Zori specifications here.

```
<Population> WalkersPerProcessor="10"
Punch="True"</Population>
```

Once the user generates all the necessary input files, he or she can initialize the QMC calculation. These calculations usually require a lot of computer power, but QMC is quite parallel, so the calculations typically run in multiprocessor systems or clusters. We implemented a simple window within the GUI to input all the parameters to run in the cluster. Here, we present only a pseudocode that highlights the main variables to be specified because the real code will be specific to the architecture used:

```
function run_GANITA(nodes, run,
command, extra)
  write script ganita.scr
    #include all specifications
  if nodes = 1
    #run single-node script
  else if nodes > 1
    #run in parallel
  end
```

The function uses the necessary information to generate a script called `ganita.scr`, which the GUI then sends to the cluster's queue system. Zopi prompts all the variables that the function requires. The user can write a simple function to create a template for his or her chosen cluster.

Running Zopi

With the GUI's aid, the user can prepare and execute a calculation using Zori with little or no difficulty. He or she needs only to click on the *simulate* button located on the interface's right side after

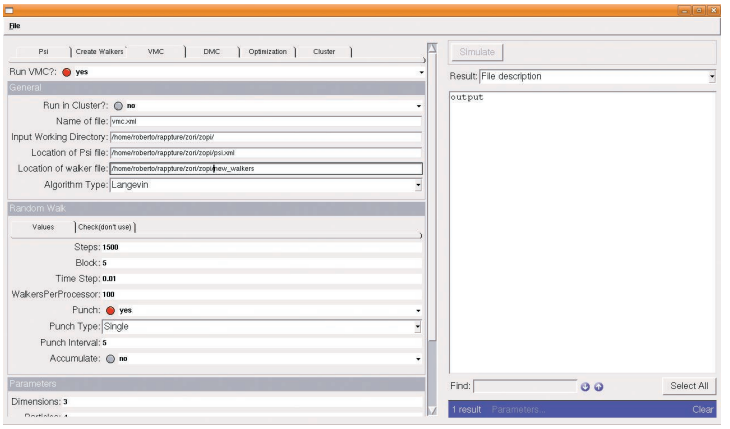

Figure 4. Virtual Monte Carlo (VMC) run. The input appears on the left side of the window, and the information regarding the run appears on the right side.

completing the requested information on the left side to launch the calculation.

Internally, the GUI saves the input values, and Rappture interprets them based on the tool executable's information. The Rappture interface then builds the XML file that the program needs to run Zori, and the interface builds the appropriate command line to run Zori with all the user's specifications. For instance, a typical Zori command line for a DMC calculation would look as follows:

```
./zori -i dmc.xml -p psi.xml -r new_
   walkers -t 720
```

Here, we assume that the executable for Zori as well as the necessary XML input files are located in the directory from which we send the job. This command line asks Zori to run by using the file `dmc.xml` as the main input file and the `psi.xml` file for the wave function; the walker files come from the `new_walkers` prefix. Zori will run for a maximum of 720 minutes on the specified machine. This input procedure is now encapsulated within Zopi, where all these options are inputs that the user can enter into the proper box. The tool executable makes the proper arrangements to parse and run the program with all the correct specifications.

The optimization program, Opt++, has a slightly different command line. It invokes the module `zopt`:

```
./zopt -i walkers.xml -p psi.xml -r
   new_walkers
```

The GUI performs all the previous steps and hides them from the user. As an example of how the user would use the GUI, suppose he or she has a valid wave function file that Zori can understand—`psi.xml`—and wants to run a VMC calculation. The user would only have to go to the GUI's VMC section and activate this calculation by selecting *yes* in response to the `run VMC` command. This will change the GUI's look and display the information fields needed to specify a VMC calculation. After providing this information, the user just has to go to the GUI's *submit* section to specify whether the calculation should run interactively on one processor or on the cluster's queue system where the GUI runs. After entering this information, the user clicks on the *simulate* button to perform the calculation. If he or she chooses to perform the run interactively, the right panel would then show the Zori's output. Figure 4 shows a GUI screenshot during a VMC calculation. At this point, the user can choose to close the GUI and let the calculation continue running in the background or continue to monitor the run's progress by calling `zavg`.

Although Rappture offers a wide array of output implementations (graphs, meshes, clouds, and, more simply, logs), we decided to implement Zopi with only the terminal screen output—this is a matter of convenience. QMC calculations aren't usually run for long, thus having a GUI running the whole time presents no extra benefit. On the other hand, Zori comes with a tool named `zavg`, which the user can call at any time to determine the calculation convergence. Therefore, instead of taking advantage of Rappture's wide array of information, we've chosen to observe if the calculation itself is running smoothly. Another option would be to have Rappture call `zavg` periodically to analyze and display the run's updated statistical analysis. This capability, however, is left for future implementations.

When the program finishes the calculation, Zopi outputs a simple indication of which runs have taken place. As stated earlier, the user can choose to close the GUI after submitting the job. This message will only appear if the GUI is still active when the run ends.

A simple and understandable way of running a given computer package is essential to making software accessible to the general user. In scientific programming, developerse don't often give this capability a high priority. Increasingly, production-package

developers are turning to GUI development to make their application codes user friendly.

With the development of a simple GUI for Zori called Zopi, we hope to broaden the use of QMC and show the scientific community the ease and benefits of generating such tools, which also offer instructional advantages for students in higher education.

Acknowledgments

Roberto Olivares-Amaya and Carlos Amador-Bedolla thank DGAPA-UNAM for financial support during the semester at Berkeley. Romelia Salomón-Ferrer and William A. Lester Jr. were supported by the US Department of Energy under contract number DE-AC03-76SF00098. We give special thanks to Rapture mailing list members Michael McLennan and Alan Aspuru-Guzik for invaluable help during the development of this project.

References

1. M. Lundstrom and G. Klimeck, "The NCN: Science, Simulation, and Cyber Services," *Proc. 2006 IEEE Conf. Emerging Technologies – Nanoelectronics*, IEEE CS Press, 2006, pp. 496–500.
2. A. Aspuru-Guzik et al., "Zori 1.0: A Parallel Quantum Monte Carlo Electronic Structure Package" *J. Computational Chemistry*, vol. 26, no. 8, 2005, pp. 856–862.
3. B.L. Hammond, W.A. Lester Jr., and P.J. Reynolds, *Monte Carlo Methods in Ab Initio Quantum Chemistry*, World Scientific, 1994.
4. J.B. Anderson, "Quantum Monte Carlo: Atoms, Molecules, Clusters, Liquids and Solids," *Reviews in Computational Chemistry*, vol. 13, K.B. Lipkowitz and D.B. Boyd, eds., Wiley-VCH, 1999, pp. 132–182.
5. W.M.C. Foulkes et al., "Quantum Monte-Carlo Simulations of Solids," *Rev. Modern Physics*, vol. 73, no. 1, 2001, pp. 33-83.
6. A. Aspuru-Guzik and W.A. Lester Jr., "Quantum Monte Carlo Methods for the Solution of the Schrödinger Equation for Molecular Systems in Computational Chemistry," *Handbook of Numerical Analysis: Special Volume, Computational Chemistry*, vol. 10, C. Le Bris, ed., Elsevier, 2003, pp. 485–535.
7. A. Aspuru-Guzik and W.A. Lester, Jr., "Quantum Monte Carlo: Theory and Application to Molecular Systems," *Advances in Quantum Chemistry*, vol. 49, Elsevier, 2005, pp. 209–226.
8. P.J. Reynolds et al., "Fixed-Node Quantum Monte Carlo for Molecules," *J. Chemical Physics*, vol. 77, no. 11, 1982, pp. 5593–5603.
9. M. Schmidt et al., "General Atomic and Molecular Electronic Structure System," *J. Computational Chemistry*, vol. 14, no. 11, 1993, pp. 1347–1363.
10. G. Velde et al., "Chemistry with ADF," *J. Computational Chemistry*, vol. 22, no. 9, 2001, pp. 931–967.
11. G. Klimeck et al., "NEMO 3-D and nanoHUB: Bridging Research and Education," *Proc. 6th IEEE Conf.*, 2006, IEEE CS Press, pp. 441–444.
12. B. Radak, H. Hwang, and G.C. Schatz, "Modeling Ion Channels Using Poisson-Nernst-Planck Theory as an Integrated Approach to Introducing Nanotechnology Concepts: The PNP Cyclic Peptide Ion Channel Model," *J. Chemical Education*, vol. 85, no. 5, 2008, pp. 744–748.

Roberto Olivares-Amaya *is a PhD student in the Department of Chemistry and Chemical Biology at Harvard University. His research includes the study of structure, response, and reactivity of molecular systems studied by first-principles electronic structure methods. Olivares-Amaya has a BS in chemistry from UNAM and was an exchange student at the University of California, Berkeley, when this work was performed. Contact him at olivares@chemistry.harvard.edu.*

Romelia Salomón-Ferrer *is a postdoctoral researcher at the California Institute of Technology. Her research interests include energy and electron and proton transfer between molecules. Romelia has a PhD in chemistry from the University of California, Berkeley, where she worked in Monte Carlo method development and application to molecular systems of biological interest. Contact her at romelia@caltech.edu.*

William A. Lester Jr. *is a chemistry professor at the University of California, Berkeley, and a principal investigator in the Chemical Sciences Division, Lawrence Berkeley National Laboratory. His research interests are the electronic structure of atoms and molecules, including quantum Monte Carlo method development and application. Lester has a PhD in chemistry from the Catholic University of America. He is a fellow of the American Physical Society, the American Association for the Advancement of Science, and the California Academy of Sciences. He is also a member of the American Chemical Society. Contact him at walester@lbl.gov.*

Carlos Amador-Bedolla *is a chemistry professor at the National Autonomous University of Mexico (UNAM). His research interests are in first-principles electronic structure methods. Amador-Bedolla has a PhD in chemistry from UNAM. Contact him at amador@cbsj.org.*