Editor: Dianne P. O'Leary, oleary@cs.umd.edu

COMPUTATIONAL SOFTWARE: WRITING YOUR LEGACY

By Dianne P. O'Leary

IN SCIENTIFIC COMPUTING, WE SOMETIMES BEGIN WITH A CLEAN SLATE; WE'RE GIVEN A NEW PROBLEM TO SOLVE, AND WE WRITE SOFTWARE TO ACCOMPLISH THE TASK. IN THIS situation, we're the inventors. Other times, we work on a problem for which considerable software development has been done, often over a period of many years. The existing software might have had many authors, some of whom have moved on to other positions. In this situation, our job is more akin to detective work. We study the existing code, run examples to see how it behaves, and come to understand both what it does and how it does it.

In this homework assignment, we consider the second situation, in which we're asked to work with a *legacy code*—one that has been in use for a while and now needs maintenance by someone other than its author. We use the following Matlab function as an example:

```
function [r, q] = posted (C)

[m,n] = size(C);
for k = 1:n
  for j=1:m
    x(j) = C(j,k);
  end
  xn = 0;
  for j=1:m,
    xn = xn + x(j)*x(j);
  end
  r(k,k) = sqrt(xn);
  for j=1:m,
    q(j,k) = C(j,k)/r(k,k);
  end
  for j = k+1:n
    r(k,j) = 0;
    for p=1:m
      r(k,j) = r(k,j) + q(p,k)'*C(p,j);
    end
    for p=1:m
      C(p,j) = C(p,j) - q(p,k)*r(k,j);
    end
  end
end
```

Let's consider some principles of documentation and design, and see how they apply to `posted`.

Documentation

Documentation provides you and other potential users of your code with an easy source of information about the software's use and design. Although you completely understand the code you write today, by next year, next month, or even next week, you'll be surprised at how difficult it is to reconstruct your reasoning if you fail to document it.

Ideally, the documentation at the top of the module provides basic information to help a potential user decide whether the software is of interest. It should include

- the code's purpose (*why*: this is certainly the first thing a user wants to know);
- the author's name (*why*: it gives users someone to whom they can report bugs and send questions);
- the date of the original code and a list of later modifications (*why*: it gives information such as whether the code is likely to run under the current computer environment and whether it might include the latest advances);
- a description of each input parameter (*why*: so that a user knows what information must be provided and in what format);
- a description of each output parameter (*why*: so the user knows what information the software will yield); and
- a brief description of the method and references (*why*: to help the user decide whether the method fits his or her needs).

Inline documentation identifies the major sections of the code and provides some detail on the method used. It's im-

Tools

Organizations often have their own standards for documentation of programs. One of the most widely admired and underutilized systems is Donald Knuth's *Literate Programming*,[1] used, for example, in his TeX document typesetting language. Knuth's programming style is also a model of clarity and good design.

Becoming a good programmer requires practice as well as good models. Jon Bentley's book[2] is an excellent source of deceptively simple problems with beautiful solutions.

Mastery of the capabilities of the programming language is essential to writing good software; *Matlab Guide* discusses performance optimization for Matlab.[3]

Considerable research was done in the 1970s to prove program correctness, but it isn't yet a mainstream activity.[4]

References

1. D.E. Knuth, *Literate Programming*, Ctr. for the Study of Language and Information, 1992.
2. J. Bentley, *Programming Pearls*, 2nd ed., Addison-Wesley, 2000.
3. D.J. Higham and N.J. Higham, *Matlab Guide*, SIAM Press, 2000.
4. P. Cousot, "Methods and Logics for Proving Programs," *Handbook of Theoretical Computer Science*, J. van Leeuwen, ed., Elsevier, 1990, pp. 843–993.

portant in specifying the algorithm, identifying bugs, and providing information to someone who might need to modify the software to solve a slightly different problem.

Note that the documentation should be an integral part of the code; in other words, it isn't enough to include it in a separate document because a potential user might not have access to that document.

Problem 1.

Guided by the principles just listed, add documentation to `posted`. Hint: upon completion, `q * r = C`.

Software Design

Software should be designed according to a principle articulated by Albert Einstein: "Make everything as simple as possible, but not simpler."

- Code should be modular, so that a user can pull out a piece and substitute another when necessary. A minimization code, for instance, should call on a separate function to get a function value, so that it can be used to minimize any function.
- On the other hand, considerable overhead is involved in function calls, so each module should involve a substantial computation to mask this overhead. Don't write a separate function to add five numbers, for example.
- Input parameters should be tested for validity, and clear error messages should be generated for invalid input. If a value defined to be the number of iterations is a negative or complex number, for instance, the user should be informed of this error.
- Data that a function needs should be specified in variables, not constants. A subroutine to solve a linear system, for example, should work for any matrix size, rather than having a size specified.
- *Spaghetti code* should be avoided. In other words, the sequence of instructions should be top-to-bottom (including loops), without a lot of jumps in control.
- The names of variables should be chosen to remind the reader of their purpose. For example, `lambda` is better than `l` as the name of a Lagrange multiplier.
- The code should be reasonably efficient. In particular, it shouldn't take an order of magnitude more time or storage than necessary. If a code to solve a linear system with n variables takes $O(n^4)$ operations or $O(n^3)$ storage, for example, it isn't so useful.
- And, of course, the program should be correct.

Problem 2.

Judge `posted` according to each of the first six design principles just listed. (We'll consider its efficiency and correctness later.)

Validation and Debugging

It would be comforting to have proof that each piece of code we use is correct. Although considerable effort has gone into developing methodologies for proving correctness, there are formidable limitations. If correctness means matching a set of specifications for the code, how do we know the specifications are correct? What does correctness mean when we consider the effects of round-off error? And even if each module is correct, can we ensure that the modules interact with each other correctly?

Rather than a proof of correctness, in most situations we settle for a validation of correctness on a limited set of inputs that we believe span the range of possibilities. Proper design of this testing code is just as important as proper design of the code being tested, and it's a critical part of the debugging process:

- The testing code should be well-documented and easy to read.

- The test should exercise every statement in the target code, include all typical kinds of correct input, and test all conceivable input errors.
- The testing code should compare the output against some trusted result and create a log of the test results.
- The testing code should be archived so that it can be used later in case the target code is modified.

In addition to testing code, a powerful way to debug a code is to write it one day and then read it carefully the next, reasoning through each statement's results to be sure that it performs as intended. Programmers in the 1960s and 1970s had a major incentive to do such desk checks: running the program once often involved waiting until the next morning to see the results, so it was important to get it right the first time. With today's machines, we usually have the luxury of seeing results from our programs much more quickly. It's tempting just to run and modify the program until the answers look good, but this is no substitute for a careful reading.

Let's apply these validation principles to `posted`.

PROBLEM 3.

a. What does `posted` do?

b. Develop a testing code for `posted`.

c. Users have complained that `posted` doesn't seem to behave well when the matrix `C` has more columns than rows. In particular, they've come to expect that `q' * q` is an identity matrix, which is no longer true. Investigate this bug complaint and see what can be done.

Efficiency

Finally, we turn our attention to making `posted` more efficient. The main sources of inefficiency in `posted` arise from

- failing to use the vector capabilities of Matlab (for example, writing a loop to perform `q(:,k)'*C(:,j)`);
- failing to use built-in functions such as `norm`; and
- failing to initialize matrices such as `r` to all zeros, and instead forcing Matlab to allocate new space each time through the loop.

Let's eliminate these inefficiencies.

PROBLEM 4.

The users have run a profiler on their software system to determine timing for each part of their code, and they've found that `posted` takes 22 percent of the total time. Their typical input matrices `C` have roughly 200 rows and 100 columns. Change `posted` to make it run faster. Test the original and modified versions, graphing the time required for problems with 200 rows and 50, 60, …, 200 columns.

After redesign, documentation, and validation, you should have a program that runs 100 times faster than `posted` and provides a much more useful legacy.

Partial Solution to Last Issue's Homework Assignment

Fast Solvers and Sylvester Equations: Both Sides Now

By Dianne P. O'Leary

In the last issue, we used the structure of a problem with n^2 unknowns to reduce the amount of computation from $O(n^6)$ (using the Cholesky decomposition) to $O(n^2 \log_2 n)$ when n is a power of two. Because we're computing n^2 answers, this is close to optimal, and it illustrates the value of exploiting every possible bit of structure in our problems.

Let $n = 5$ and define the $n^2 \times n^2$ matrix $\boldsymbol{A}_x$ by

$$\boldsymbol{A}_x = \frac{1}{h^2}\begin{bmatrix} \boldsymbol{T} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T} \end{bmatrix},$$

where $h = 1/(n+1)$, and let

$$\boldsymbol{T} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

and

$$\boldsymbol{A}_y = \frac{1}{h^2}\begin{bmatrix} 2\boldsymbol{I} & -\boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{I} & 2\boldsymbol{I} & -\boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & -\boldsymbol{I} & 2\boldsymbol{I} & -\boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & -\boldsymbol{I} & 2\boldsymbol{I} & -\boldsymbol{I} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & -\boldsymbol{I} & 2\boldsymbol{I} \end{bmatrix},$$

where $\boldsymbol{I}$ is the identity matrix of dimension $n \times n$. We want to solve the linear system

$$(\boldsymbol{A}_x + \boldsymbol{A}_y)\mathbf{u} = \mathbf{f}, \tag{1}$$

The entries in the vector $\mathbf{u}$ are approximations to values of a function u at points (x, y) in a square grid, and entries in $\mathbf{f}$ contain values of a function f: $\mathbf{u} \approx [u(x_1, y_1), \ldots, u(x_n, y_1), u(x_1, y_2), \ldots, u(x_n, y_2), \ldots u(x_1, y_n), \ldots, u(x_n, y_n)]^T$ and $\mathbf{f} = [f(x_1, y_1), \ldots, f(x_n, y_1), f(x_1, y_2), \ldots, f(x_n, y_2), \ldots f(x_1, y_n), \ldots, f(x_n, y_n)]^T$.

Problem 1.

Show that we can write Equation 1 as

$$(\boldsymbol{B}_y\boldsymbol{U} + \boldsymbol{U}\boldsymbol{B}_x) = \boldsymbol{F}, \tag{2}$$

where the matrix entry u_{jk} is our approximation to $u(x_j, y_k)$, $f_{jk} = f(x_j, y_k)$, and $\boldsymbol{B}_y = \boldsymbol{B}_x = (1/h^2)\boldsymbol{T}$.

Answer:

Equating the (j, k) element on each side of Equation 2, we obtain

$$f(x_j, y_k) = 1/h^2(-u(x_{j-1}, y_k) + 2u(x_j, y_k) - u(x_{j+1}, y_k) - u(x_j, y_{k-1}) + 2u(x_j, y_k) - u(x_j, y_{k+1})),$$

which is the same as equation $(k-1)n + j$ of Equation 1.

Problem 2.

a. Consider the Sylvester equation $\boldsymbol{L}\boldsymbol{U} + \boldsymbol{U}\boldsymbol{R} = \boldsymbol{C}$, where $\boldsymbol{L}$ is lower triangular and $\boldsymbol{R}$ is upper triangular. Show that we can easily determine the elements of the matrix $\boldsymbol{U}$ either row-by-row or column-by-column. How many arithmetic operations does this algorithm require?

b. By examining your algorithm, determine necessary and sufficient conditions on the main diagonal elements of $\boldsymbol{L}$ and $\boldsymbol{R}$ (that is, their eigenvalues) to ensure that a solution to the Sylvester equation exists.

c. Suppose we want to solve the Sylvester equation $\boldsymbol{A}\boldsymbol{U} + \boldsymbol{U}\boldsymbol{B} = \boldsymbol{C}$, where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ of dimension $n \times n$ are given. ($\boldsymbol{A}$ and $\boldsymbol{B}$ are unrelated to the previously described matrices.) Let $\boldsymbol{A} = \boldsymbol{W}\boldsymbol{L}\boldsymbol{W}^*$ and $\boldsymbol{B} = \boldsymbol{Y}\boldsymbol{R}\boldsymbol{Y}^*$, where $\boldsymbol{W}\boldsymbol{W}^* = \boldsymbol{W}^*\boldsymbol{W} = \boldsymbol{I}$, $\boldsymbol{Y}\boldsymbol{Y}^* = \boldsymbol{Y}^*\boldsymbol{Y} = \boldsymbol{I}$; $\boldsymbol{L}$ is lower triangular; and $\boldsymbol{R}$ is upper triangular. (This is called a *Schur decomposition* of the two matrices.) Show that we can solve the Sylvester equation by applying the algorithm derived in part (a) to the equation $\boldsymbol{L}\hat{\boldsymbol{U}} + \hat{\boldsymbol{U}}\boldsymbol{R} = \hat{\boldsymbol{C}}$, where $\hat{\boldsymbol{U}} = \boldsymbol{W}^*\boldsymbol{U}\boldsymbol{Y}$ and $\hat{\boldsymbol{C}} = \boldsymbol{W}^*\boldsymbol{C}\boldsymbol{Y}$.

Answer:
a. Using Matlab notation for subvectors, the algorithm is the following:

```
for i = 1 : n,
 for j = 1 : n,
  U(i,j) = (C(i,j) – L(i, 1 : i – 1) * U(1 : i – 1, j)
   –U(i, 1 : j – 1) * R(1 : j – 1, j))/(L(i, i) + R(j, j))
 end
end
```

The number of multiplications is

$$\sum_{i=1}^{n}\sum_{j=1}^{n}(i-1+j-1)=n^2(n-1),$$

and the other operations are also easy to count.

b. The algorithm fails if $L(i, i) + R(j, j) = 0$ for some value of i and j. The main diagonal elements of triangular matrices are the matrix's eigenvalues, so it's necessary and sufficient that L and $-R$ have no common eigenvalues.

c. If $\boldsymbol{AU} + \boldsymbol{UB} = \boldsymbol{C}$, then $\boldsymbol{WLW}^*\boldsymbol{U} + \boldsymbol{UYRY}^* = \boldsymbol{C}$.

Multiplying on the left by $\boldsymbol{W}^*$ and on the right by $\boldsymbol{Y}$, we obtain $\boldsymbol{L}\hat{\boldsymbol{U}} + \hat{\boldsymbol{U}}\boldsymbol{R} = \hat{\boldsymbol{C}}$.

PROBLEM 3.

Determine a way to solve the equation $\boldsymbol{\Lambda}_x\boldsymbol{Y} + \boldsymbol{Y}\boldsymbol{\Lambda}_y = \boldsymbol{F}$, where $\boldsymbol{\Lambda}_x$ and $\boldsymbol{\Lambda}_y$ are diagonal, using only $O(n^2)$ arithmetic operations.

Answer:
The algorithm of Problem 2a reduces to $U(i, j) = F(i, j)/(L(i, i) + R(j, j))$ for $i, j = 1, \ldots, n$, which requires n^2 additions and divisions.

PROBLEM 4.

a. We know the eigenvalues and eigenvectors of $\boldsymbol{B}_x$. Denote the elements of the vector $\mathbf{v}_j$ by

$$v_{kj} = \alpha_j \sin\frac{kj\pi}{n+1},$$

where we choose α_j so that $\|\mathbf{v}_j\| = 1$. Show that $\boldsymbol{B}_x\mathbf{v}_j = \lambda_j\mathbf{v}_j$, where

$$\lambda_j = \frac{\left(2-2\cos\frac{j\pi}{n+1}\right)}{h^2} \quad \text{for } j = 1, 2, \ldots, n.$$

b. Show that we can multiply a vector by the matrix $\boldsymbol{V}$ or $\boldsymbol{V}^T$ via a discrete Fourier (sine) transform or inverse Fourier (sine) transform of length n. The discrete sine transform of a vector $\mathbf{x}$ is

$$y_k = \sum_{j=1}^{n} x_j \sin(jk\pi/(n+1)).$$

Answer:
a. Recall the identities

$$\sin(a \pm b) = \sin a \cos b \pm \cos a \sin b.$$

If we form $\boldsymbol{B}_x$ times the jth column of $\boldsymbol{V}$, then the kth element is

$$\frac{-v_{k-1,j} + 2v_{k,j} - v_{k+1,j}}{h^2}$$

$$= \frac{\alpha_j}{h^2}\left(-\sin\frac{(k-1)j\pi}{n+1} + 2\sin\frac{kj\pi}{n+1} - \sin\frac{(k+1)j\pi}{n+1}\right)$$

$$= \frac{\alpha_j}{h^2}\left(-\sin\frac{kj\pi}{n+1}\cos\frac{j\pi}{n+1} + \cos\frac{kj\pi}{n+1}\sin\frac{j\pi}{n+1}\right.$$
$$\left. + 2\sin\frac{kj\pi}{n+1} - \sin\frac{kj\pi}{n+1}\cos\frac{j\pi}{n+1} - \cos\frac{kj\pi}{n+1}\sin\frac{j\pi}{n+1}\right)$$

$$= \frac{\alpha_j}{h^2}\left(2-2\cos\frac{j\pi}{n+1}\right)\sin\frac{kj\pi}{n+1}$$

$$= \frac{1}{h^2}\left(2-2\cos\frac{j\pi}{n+1}\right)v_{k,j}$$

$$= \lambda_j v_{k,j}.$$

An error appears in the last line of Problem 4 in "Fast Solvers and Sylvester Equations" (last issue's homework assignment); it is corrected here, on page 82. We regret any confusion this error might have caused. —*Eds.*

Stacking these elements, we obtain $\boldsymbol{B}_x\mathbf{v}_j = \lambda_j\mathbf{v}_j$.

b. This follows by writing the kth component of $\boldsymbol{V}\mathbf{y}$.

PROBLEM 5.

Write a well-documented program to solve the discretization of the differential equation using Problem 2's Schur-based algorithm and the algorithm developed in Problem 4. (Because the matrices are symmetric, the lower and upper triangular matrices are actually diagonal.) Test your algorithms for $n = 2^p$, with $p = 2, \ldots, 9$, and choose the true solution matrix $\boldsymbol{U}$ randomly. Compare the results of the two algorithms with backslash for accuracy and time.

Answer:

See the Web site (www.computer.org/cise/homework/) for the programs. Figures A and B show the results. All the algorithms give accurate results, but as n gets large, the efficiency of Problem 4's fast algorithm becomes more apparent.

Dianne P. O'Leary is a professor of computer science and a faculty member in the Institute for Advanced Computer Studies and the Applied Mathematics Program at the University of Maryland. She has a BS in mathematics from Purdue University and a PhD in computer science from Stanford. O'Leary is a member of SIAM, the ACM, and AWM. Contact her at oleary@cs.umd.edu; www.cs.umd.edu/users/oleary/.

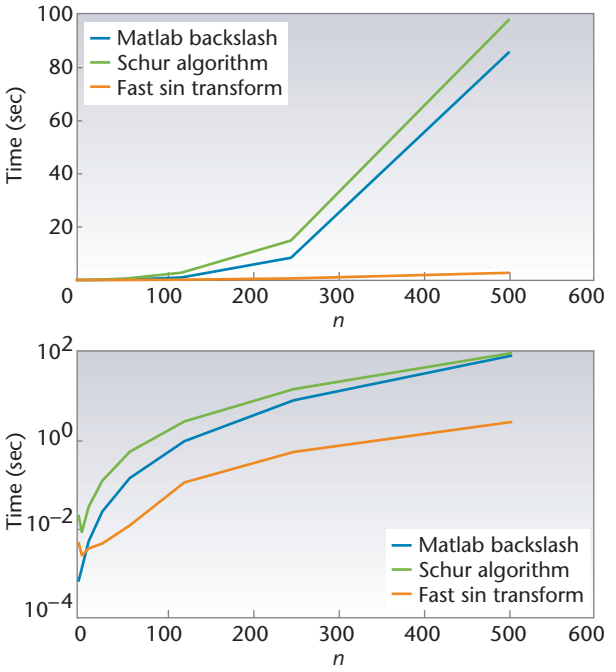

Figure A. The time (in seconds on a Sun UltraSPARC-III with clock speed 750 MHz running Matlab 6) taken by the three algorithms as a function of *n*. The bottom plot uses logscale to better display the times for the fast sin transform.

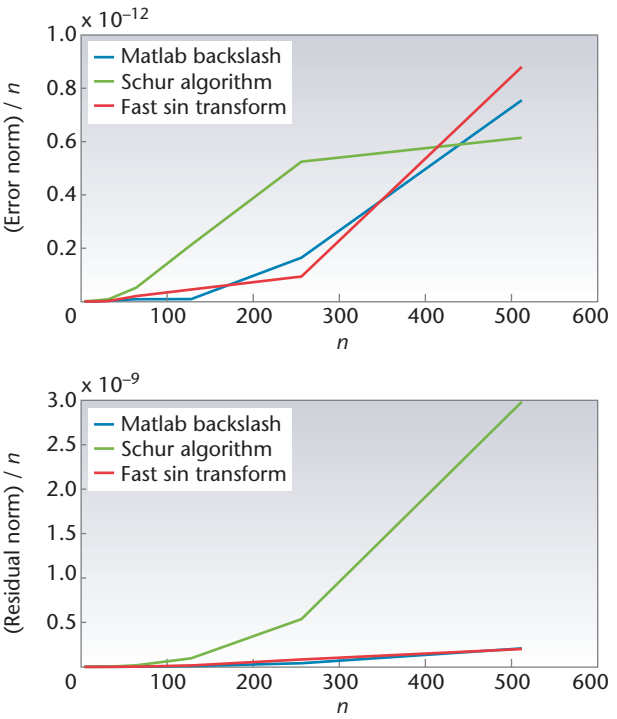

Figure B. The accuracy of the three algorithms as a function of *n*.