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

UPDATING AND DOWNDATING MATRIX FACTORIZATIONS: A CHANGE IN PLANS

By Dianne P. O'Leary

WE SELDOM GET IT RIGHT THE FIRST TIME. WHETHER WE'RE COMPOSING AN EMAIL, SEASONING A STEW, PAINTING A PICTURE, OR PLANNING AN EXPERIMENT, WE almost always make improvements on our original thought. The same is true of engineering design; we draft a plan, but changes are almost always made. Perhaps the customer changes the performance specifications, or perhaps a substitution of building materials leads to a redesign.

In this homework assignment, we consider numerical methods that make it easier to reanalyze the design after small changes are made. For definiteness, we focus on truss design, but the same principles apply to any linear model.

Truss Design

Consider the first truss of Figure 1. Beginning engineering students learn to compute the force acting on each truss member by considering two equations for each node in the truss, ensuring that the sum of the horizontal forces is zero, the sum of the vertical forces is zero, and the moment is zero. They can "march" through the equations, solving for the horizontal and vertical forces by a clever ordering. If the design is changed—by moving a node, for example, as in the second truss in the figure—then the resulting forces are just as easy to determine.

For more complex models—for example, a finite element model of the forces on a wing surface—marching no longer works, and the system of equations must be solved using a method such as Gauss elimination. We would like to have an algorithm that enabled us to easily recompute the forces if the wing's shape changed slightly.

To introduce methods for solving modified models, let's forget the marching trick and return to the familiar truss example. We start by writing a system of equations $\boldsymbol{A}\mathbf{f} = \ell$ for the unknown force on each member; the matrix of the system has one column for each unknown force and two rows per node (for horizontal and vertical forces). The load on node E is put in the right-hand side. For Figure 1's very simple first truss, for example, we have $n = 10$ forces to compute, the solution to the linear system

$$\begin{bmatrix} -1 & 0 & c & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & s & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -c & c & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & s & s & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -c & c & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & s & s & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -c & c & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & s & s & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -c & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & s & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} f_{Ah} \\ f_{Av} \\ f_{AB} \\ f_{BC} \\ f_{CD} \\ f_{DE} \\ f_{BD} \\ f_{AC} \\ f_{CE} \\ F_{Ev} \end{bmatrix} = \begin{bmatrix} \ell_{Ah} \\ \ell_{Av} \\ \ell_{Bh} \\ \ell_{Bv} \\ \ell_{Ch} \\ \ell_{Cv} \\ \ell_{Dh} \\ \ell_{Dv} \\ \ell_{Eh} \\ \ell_{Ev} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -50 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \quad (1)$$

(The subscript Ah, for example, denotes a horizontal force at node A, and the subscript BC refers to the member connecting nodes B and C.) The truss members are all equal length, so $c = \cos(\pi/3)$ and $s = \sin(\pi/3)$.

Several simple kinds of changes to the truss design produce "small" changes in the system of equations. For example,

- If we change the loading on the truss, then we keep the

Handling Small Changes

The problem of efficiently handling small changes in the model matrix arises in many situations other than engineering design. For example,

- Suppose we're solving a system of linear inequalities $\mathbf{Ax} \geq \mathbf{b}$, with $\mathbf{A}$ of dimension $m \times n$ ($m \geq n$), and we think that the first n of them should be active: $\mathbf{a}_i^T\mathbf{x} = b_i$, $i = 1, \ldots, n$. Suppose then that the solution to these equations violates the kth inequality ($k > n$), and we want to add it to our current system of equations and delete the jth equation. Can we solve our new linear system easily? This problem routinely arises in minimization problems when we have linear inequality constraints, and it's the basic computation in the simplex algorithm for solving linear programming problems.
- Suppose that we're solving a linear least-squares problem, $\mathbf{Ax} \approx \mathbf{b}$, and we get some new measurements. This adds rows to $\mathbf{A}$ and $\mathbf{b}$. Can we solve our new least-squares problem easily?
- Suppose we've computed the eigenvalues and eigenvectors of $\mathbf{A}$, and then $\mathbf{A}$ is changed by the addition of a rank-1 matrix $\hat{\mathbf{A}} = \mathbf{A} + cr^T$. What are the eigenvalues of $\hat{\mathbf{A}}$?

Each of these problems (and similar ones) can be solved cheaply by using the techniques discussed in this homework assignment.

same matrix but change the right-hand side ℓ.
- If we add a new node along with two new truss members, then our new matrix has two additional rows and columns and contains the old matrix as a submatrix.
- If we remove a set of nodes and their truss members, then we delete the columns of the matrix corresponding to the forces exerted by the members, and we delete the pair of rows corresponding to each removed node.
- If we move a node, then we change the two rows corresponding to the horizontal and vertical forces on that node, and we change the columns corresponding to its members.

For a problem with $n = 10$ unknowns, we could easily recompute the answer after any of these changes, but if $n = 1{,}000{,}000$, we might want to take advantage of our solution to the original problem to more quickly obtain a solution to the modified problem. In this homework assignment, we develop the tools to do this.

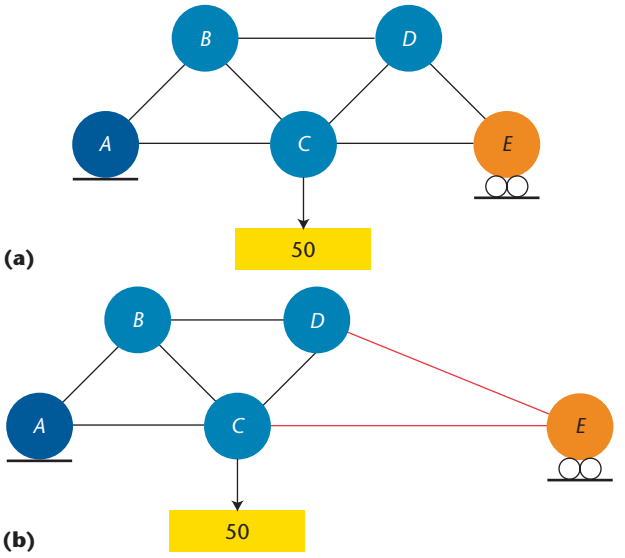

Figure 1. Truss design. (a) A truss with five nodes and seven equal-length members, loaded with a force of 50 Newtons. Node *A* is fixed (supported horizontally and vertically) and node *E* is rolling (supported only vertically). (b) A change in the truss. Member *CE* is now two times the length of the other members.

Changes to the Right-Hand Side

If we need to analyze the truss for several different loadings, then it's a good idea to compute a factorization of the matrix $\mathbf{A}$ once and save it for multiple uses. For example, if we compute the $\mathbf{LU}$ decomposition with partial pivoting

$$\mathbf{PA} = \mathbf{LU}, \tag{2}$$

where $\mathbf{P}$ is a permutation matrix that interchanges rows of $\mathbf{A}$, $\mathbf{L}$ is a lower triangular matrix, and $\mathbf{U}$ is an upper triangular matrix, then each of the loads can be handled by solving

$$\mathbf{Ly} = \mathbf{P}\ell$$

by forward-substitution, and then computing

$$\mathbf{Uf} = \mathbf{y}$$

by back-substitution. If $\mathbf{A}$ were a dense matrix, with very few nonzeros, then the initial $\mathbf{LU}$ decomposition would cost $O(n^3)$ operations, whereas forward- and back-substitution would cost only $O(n^2)$. Taking advantage of the sparsity of $\mathbf{A}$ can reduce the cost of both the $\mathbf{LU}$ decomposition and the substitution (see "Solving Sparse Linear Systems: Taking the Direct Approach," *Computing in Science & Eng.*, vol. 7, no. 5, 2005, pp. 62–67), but substitution will still be significantly less costly than factorization when n is large.

Changes to the Matrix

If we change the shape of the truss, then the matrix changes. We can handle this in two ways: by using the *Sherman-Morrison-Woodbury* formula or by updating a matrix decomposition.

Sherman–Morrison–Woodbury Formula

Sometimes our matrix changes in a rather simple way, and we want to reconsider our problem, making use of the original decomposition without explicitly forming the update.

Suppose we have the factorization from Equation 2 and now want to solve the linear system $(\boldsymbol{A} - \mathbf{z}\mathbf{v}^T)\mathbf{f} = \ell$, where $\mathbf{z}$ and $\mathbf{v}$ are column vectors of length n, so that $\mathbf{z}\mathbf{v}^T$ is an $n \times n$ matrix. For example, if we decide to move node E as in the truss of Figure 1, then we need to change column 6 in our matrix. To do this, we set $\mathbf{v}$ to be the 6th unit vector (1 in position 6 and zeroes elsewhere) and $\mathbf{z}$ to be the difference between the new column and the old one.

In the next problem, we see how to apply this principle to more than one set of changes in the matrix.

Problem 1.

a. Suppose we want to change columns 6 and 7 in our matrix $\boldsymbol{A}$. Express the new matrix as $\boldsymbol{A} - \boldsymbol{Z}\boldsymbol{V}^T$, where $\boldsymbol{Z}$ and $\boldsymbol{V}$ have dimension $n \times 2$.

b. Suppose we want to change both column 6 and row 4 of $\boldsymbol{A}$. Find $\boldsymbol{Z}$ and $\boldsymbol{V}$ so that our new matrix is $\boldsymbol{A} - \boldsymbol{Z}\boldsymbol{V}^T$.

In Problem 2, we see how this formulation of our new matrix as a small-rank change in our old matrix leads to an efficient computational algorithm.

Problem 2.

a. Assume that $\boldsymbol{A}$ and $\boldsymbol{A} - \boldsymbol{Z}\boldsymbol{V}^T$ are both nonsingular. Show that

$$(\boldsymbol{A} - \boldsymbol{Z}\boldsymbol{V}^T)^{-1} = \boldsymbol{A}^{-1} + \boldsymbol{A}^{-1}\boldsymbol{Z}(\boldsymbol{I} - \boldsymbol{V}^T\boldsymbol{A}^{-1}\boldsymbol{Z})^{-1}\boldsymbol{V}^T\boldsymbol{A}^{-1}$$

by verifying that the product of this matrix with $\boldsymbol{A} - \boldsymbol{Z}\boldsymbol{V}^T$ is the identity matrix $\boldsymbol{I}$. This is called the Sherman-Morrison-Woodbury formula.

b. Suppose we have an $\boldsymbol{L}\boldsymbol{U}$ decomposition of $\boldsymbol{A}$ as in Equation 2. Assume that $\boldsymbol{Z}$ and $\boldsymbol{V}$ are $n \times k$ and $k << n$. Show that we can use this decomposition and the Sherman-Morrison-Woodbury formula to solve the linear system $(\boldsymbol{A} - \boldsymbol{Z}\boldsymbol{V}^T)\mathbf{f} = \ell$ without forming any matrix inverses. (If $\boldsymbol{A}$ is dense, then we perform $O(n^2)$ operations using Sherman-Morrison-Woodbury, rather than the $O(n^3)$ operations needed to solve the linear system from scratch.) Hint: Remember that we can compute $\boldsymbol{A}^{-1}\mathbf{y}$ by solving a linear system using forward- and back-substitution with the factors $\boldsymbol{L}$ and $\boldsymbol{U}$.

In the following two problems, let's experiment with the Sherman-Morrison-Woodbury algorithm to see when it can be useful.

Problem 3.

Implement the Sherman-Morrison-Woodbury algorithm from Problem 2b. Debug it by factoring the matrix in Equation 1, modeling the first truss in Figure 1, and then changing the model to the second truss.

Problem 4.

For n taken to be various numbers between 10 and 1,000, generate a random $n \times n$ matrix $\boldsymbol{A}$. Find the number of updates k_0 that makes the cost of the Sherman-Morrison-Woodbury method comparable to computing $\boldsymbol{A}\backslash\ell$. Plot k_0 as a function of n.

Updating a Matrix Decomposition

The Sherman-Morrison-Woodbury formula lets us solve modified linear systems without explicitly modifying our matrix factorization. It's very efficient when we need to make only a few changes.

In some problems, though, we must do a long series of updates to the matrix, and it's better to explicitly update the factorization. We could consider updating an $\boldsymbol{L}\boldsymbol{U}$ factorization, but this can be complicated because pivoting is necessary to preserve stability. Instead, let's use a factorization that's stable without pivoting. This also enables us to consider matrices that have more rows than columns, such as those that arise in least-squares problems.

Suppose we've factored

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R},$$

where $\boldsymbol{A}$ is $m \times n$ with $m \geq n$, $\boldsymbol{Q}$ is $m \times m$ and orthogonal

($\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}$), and $\boldsymbol{R}$ is $m \times n$ and has zeros below its main diagonal. For definiteness, we'll let $\boldsymbol{A}$ have dimensions 5×3.

As examples, let's consider two kinds of common changes:

- *Adding a row*. In least squares, this happens when new data comes in; in our truss problem it means that we have a new node.
- *Deleting a column*. In least squares, this happens when we decide to reduce the number of parameters in the model; in our truss problem, it could result from removing a member.

Let's look at both in more detail.

Adding a Row

Denote the new matrix as $\hat{\boldsymbol{A}}$. Our decomposition can be written as

$$\hat{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{Q} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ a_{61} & a_{62} & a_{63} \end{bmatrix}. \quad (3)$$

To complete the decomposition, we need to reduce the a's to zeros. We can do this by using n Givens rotations—much cheaper than recomputing the entire decomposition.

We'll write the Givens matrix as

$$\boldsymbol{G} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix},$$

where $c^2 + s^2 = 1$. Thus, c and s have the geometric interpretation of the cosine and sine of angle θ; multiplying a vector by this matrix rotates the vector by an angle θ.

Let's see how we can use Givens rotation matrices.

Problem 5.

a. Given a vector $\mathbf{z} \neq \mathbf{0}$ of dimension 2×1, find $\boldsymbol{G}$ so that $\boldsymbol{G}\mathbf{z} = x\mathbf{e}_1$, where $x = \|\mathbf{z}\|$ and $\mathbf{e}_1$ is the vector with a 1 in the first position and zeros elsewhere.

b. We will use the notation $\boldsymbol{G}_{ij}$ to denote an $m \times m$ identity matrix with its ith and jth rows modified to include the Givens rotation: for example, if $m = 6$, then

$$\boldsymbol{G}_{25} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & c & 0 & 0 & s & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & s & 0 & 0 & -c & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Multiplication of a vector by this matrix leaves all but rows 2 and 5 of the vector unchanged. Show that we can finish our $\boldsymbol{QR}$ decomposition by (left) multiplying the $\boldsymbol{R}$ matrix in Equation 3 first by $\boldsymbol{G}_{16}$, then by $\boldsymbol{G}_{26}$, and finally by $\boldsymbol{G}_{36}$, where the angle defining each of these matrices is suitably chosen. To preserve the equality, we multiply the $\boldsymbol{Q}$-matrix by $\boldsymbol{G}_{16}^T\boldsymbol{G}_{26}^T\boldsymbol{G}_{36}^T$ on the right, and we have the updated factorization.

Deleting a Column

If we delete column 1 from $\boldsymbol{A}$, for example, we can write the decomposition as

$$\hat{\boldsymbol{A}} = \boldsymbol{Q} \begin{bmatrix} r_{12} & r_{13} \\ r_{22} & r_{23} \\ 0 & r_{33} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

The resulting $\boldsymbol{R}$ is almost upper triangular; we just need rotations to reduce the elements labeled r_{22} and r_{33} to zero. In general, we need $n - k$ rotations when column k is deleted.

Algorithms for deleting a row and adding a column are similar to those for adding a row and deleting a column. In the next problem, we construct algorithms for changing a column.

Problem 6.

Write a Matlab function that updates a $\boldsymbol{QR}$ decomposition of a matrix $\boldsymbol{A}$ when a single column is changed. Apply it to the truss examples in Problem 3.

The Point of Updating

It might seem silly to worry so much about whether to update

Tools

Standard textbooks on scientific computing, such as the one by Charles van Loan, discuss the *QR* factorization and the use of Givens rotations.[1]

Other stable methods for modifying matrix factorizations are considered by P.E. Gill, Gene Golub, Walter Murray, and Michael Saunders.[2] Gene Golub also discusses the solution of eigenvalue problems when the matrix is modified.[3]

References

1. C.F. van Loan, *Introduction to Scientific Computing*, Prentice Hall, 2000, section 7.2.
2. P.E. Gill et al., "Methods for Modifying Matrix Factorizations," *Mathematics of Computation*, vol. 28, no. 126, 1974, pp. 505–535.
3. G.H. Golub, "Some Modified Matrix Eigenvalue Problems," *SIAM Rev.*, vol. 15, 1973, pp. 318–335.

or recompute; computers are fast, and if we make one change to the matrix, it really doesn't matter which we do.

But when we need to do the task over and over again, perhaps in a loop that solves a more complicated problem, it's essential to use appropriate updating techniques to reduce the cost.

Unfortunately, the literature contains many unstable updating algorithms, so it's important to use stable and trusted algorithms such as those given here and in the references in the "Tools" sidebar.

Acknowledgments

I'm grateful to Brendan O'Leary for helpful discussions regarding trusses.

Partial Solution to Last Issue's Homework Assignment

Computational Software: Writing Your Legacy

By Dianne P. O'Leary

IN THE JANUARY/FEBRUARY ISSUE, WE DISCUSSED SITUATIONS IN WHICH WE'RE ASKED TO WORK WITH *LEGACY CODE*—CODE THAT HAS BEEN IN USE FOR A WHILE AND NOW needs maintenance by someone other than its author. We used a sample Matlab code for all our examples.

Problem 1.

Add documentation to `mgs`.

Answer:

See the program on the Web site (www.computer.org/cise/homework).

Problem 2.

Judge `mgs` according to each of the first six design principles.

Answer:

- Data that a function needs should be specified in variables, not constants. This is fine; `C` is a variable.
- Code should be modular, so that a user can pull out one piece and substitute another when necessary. The program `mgs` factors a matrix into the product of two other matrices, and it would be easy to substitute a different factorization algorithm.
- On the other hand, considerable overhead is involved in function calls, so each module should involve a substantial computation to mask this overhead. This is also satisfied; `mgs` performs a significant computation ($O(mn^2)$ operations).
- Input parameters should be tested for validity, and clear error messages should be generated for invalid input. The factorization can be performed for any matrix or scalar, so input should be tested to be sure it isn't a string, cell variable, and so on.
- "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 `mgs` program is fine in this regard, although there is a lot of nesting of loops.
- The names of variables should be chosen to remind the

reader of their purpose. The letter `q` is often used for an orthogonal matrix, and `r` is often used for an upper triangular one, but it would probably be better practice to use uppercase names for these matrices.

Problem 3.

a. What does `mgs` do?
b. Develop a testing code for `mgs`.
c. Users have complained that `mgs` 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.

Answer:

a. This program computes a ***QR*** factorization of the matrix `C` using the modified Gram-Schmidt algorithm.

b. See the Web site (www.computer.org/cise/homework/).

c. This is corrected in `mgsfact.m` on the Web site. The columns of `q` should be mutually orthogonal, but the number of columns in `q` should be the minimum of the row and column dimensions of `C`. Nonzero columns after that are just the result of rounding errors.

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 `mgs` takes 22 percent of the total time. Their typical input matrices `C` have roughly 200 rows and 100 columns. Change `mgs` 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.

Answer:

The resulting program is on the Web site (www.computer.org/cise/homework/), and Figure A shows the timing results. The program `mgs` has been modified in `mgsfact` to use vector operations and internal functions like `norm` when possible, and preallocate storage for `q` and `r`. The `mgsfact` function runs 150 to 200 times faster than `mgs` on matrices with 200 rows, using a Sun UltraSPARC-III with clock speed 750 MHz running Matlab 6. It's an interesting exercise to determine the relative importance of the three changes.

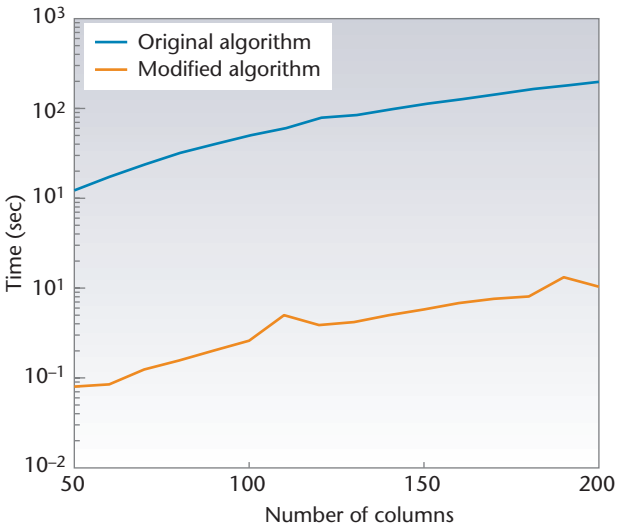

Figure A. Time taken by the two algorithms for matrices with 200 rows. The modified algorithm runs 150 to 200 times faster than the original on a Sun UltraSPARC-III.

You also might think about how an efficient implementation in your favorite programming language might differ from this one.

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 also a member of SIAM, the ACM, and the Association for Women in Mathematics. Contact her at oleary@cs.umd.edu; www.cs.umd.edu/users/oleary/.