SEAS Short Course
Programming in MATLAB
Tutorial 2: Numerical Linear Algebra

Kevin Wayne
9/12/07 v3.5

1 Summary

There are three objectives to this tutorial: (a) use MATLAB to process linear equations; (b) learn about MATLAB's numerical linear algebra libraries; (c) learn about numerical analysis; and (d) gain more experience programming in MATLAB.

2 Linear Systems of Equations

Solving a system of *linear equations* is among the most fundamental problems in science and engineering. Given a matrix A and a vector b, the goal is to find a vector x such that $Ax = b$. We will restrict attention to the important case when A is square and non-singular. Here are a few motivating applications.

- Stoichiometric coefficients for chemical equations. The equation governing the combustion of propane is $x_1C_3H_8+x_2O_2 \implies x_3CO_2+x_4H_2O$. Find coefficients x_1 through x_4 subject to conservation of mass and normalization constraints.

$$\begin{array}{rclcll} 3x_1 & = & x_3 & & & \text{(carbon)} \\ 8x_1 & = & 2x_4 & & & \text{(hydrogen)} \\ 2x_2 & = & 2x_3 & + & x_4 & \text{(oxygen)} \\ x_1 & = & 1 & & & \text{(normalize)} \end{array}$$

  The solution is $C_3H_8 + 5O_2 \implies 3CO_2 + 4H_2O$. The observation that stoichiometric coefficients tend to be small integers was among the first hints suggesting the atomic nature of matter.

- Kirchoff's current law for electrical circuits.

- Hooke's law for finite element methods.

- The PageRank algorithm for ranking the importance of web pages by Google.

2.1 Gaussian Elimination

Gaussian elimination is one of the oldest and most widely used algorithms for solving linear systems of equations. The thrust of Gaussian elimination is to transform the original system of equations into an equivalent system that is easy to solve (*upper triangular*) via a sequence of *elementary row operations*. First we will see how to solve upper triangular systems using a simple procedure known as *backward substitution*. Then we will see how to transform our original problem into an upper triangular system using a procedure known as *forward elimination*.

2.2 Elementary row operations

We describe two simple transformations (known as *elementary row operations*) that preserve all solutions to a system of linear equations. We will demonstrate these row operations on the following system of three linear equations in three unknowns.

$$\begin{bmatrix} 0 & 1 & 1 \\ 2 & 4 & -2 \\ 0 & 3 & 15 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 36 \end{bmatrix}$$

```
>> A = [0 1 1; 2 4 -2; 0 3 15];
>> b = [4; 2; 36];
```

- *Row interchange.* Interchange any two equations. For example, we can interchange the first and second rows above to yield the equivalent system.

$$\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 3 & 15 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 36 \end{bmatrix}$$

Note that we must also swap the corresponding entries in b. Swapping rows p and q in MATLAB is especially compact using vector indexing:

```
>> p = 1; q = 2;
>> A([p q], :) = A([q p], :);
>> b([p q], :) = b([q p], :);
```

- *Linear combination.* Subtract a multiple of one equation from another equation. For example, we can subtract three times the second equation from the third to obtain an equivalent system.

$$\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 12 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 24 \end{bmatrix}$$

Again, this computation is easy to express in MATLAB. The following code fragment subtracts α times the pth equation from the qth equation.

```
>> p = 2; q = 3;
>> alpha = 3.0;
>> A(q, :) = A(q, :) - alpha * A(p, :);
>> b(q, :) = b(q, :) - alpha * b(p, :);
```

2.3 Backward substitution

We now describe a straightforward algorithm for solving upper triangular systems like the one below.

$$\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 12 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 24 \end{bmatrix}$$

We directly can solve for x_3 in the last equation $12x_3 = 24$ since it involves only one unknown. Now that we know x_3, we can plug it into the second to last equation $x_2 + x_3 = 4$ and solve for $x_2 = 2$. Finally, since we know x_2 and x_3 we can plug their values into the first equation $2x_1 + 4x_2 - 2x_3 = 2$ and solve for $x_2 = 2$. In general, we solve for the n variables in reverse order using the formula

$$x_i = \frac{1}{a_{ii}} \left[b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right]$$

Implementing back-substitution in MATLAB is exactly as you would expect.

```
[m n] = size(A);
x = zeros(n, 1);
for i = n : -1 : 1    % consider variables in reverse order
    total = 0.0;
    for j = i + 1 : n
        total = total + A(i, j) * x(j);
    end
    x(i) = (b(i) - total) / A(i, i);
end
```

The sum is begging to be vectorized, which we do below. (Don't try to vectorize the outer loop – the order in which we perform the computation is crucial so it is not amenable to vectorization.) The resulting code is more elegant and (slightly) more efficient than our first attempt.

```
[m n] = size(A);
x = zeros(size(b));
for i = n : -1 : 1
    j = i+1 : n;
    x(i, :) = (b(i, :) - A(i, j) * x(j, :)) / A(i, i);
end
```

The variable j is the vector $(i{+}1, i{+}2, \ldots, n)$. The expression `x(j)` is the vector $(x_{i+1}, x_{i+2}, \ldots x_n)$ and `A(i, j)` is the corresponding vector extracted from the ith row of A. The expression `A(i, j) *x(j)` computes the desired sum as a vector inner product.

We can further improve efficiency by not explicitly declaring `j`. This saves the unnecessary memory allocations, and the resulting code is about twice as fast as our original version.

```
[m n] = size(A);
x = zeros(size(b));
for i = n : -1 : 1
    x(i, :) = (b(i, :) - A(i, i+1:n) * x(i+1:n, :)) / A(i, i);
end
```

2.4 Forward elimination

Now we describe how to transform a general system of equations into upper triangular form. A matrix is *upper triangular* if $a_{ij} = 0$ whenever $i > j$. We divide the work into n phases. In phase p, we zero out all entries in the pth column below the main diagonal, as illustrated below (where $*$ denotes a possible nonzero).

$$\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix} \Longrightarrow \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \end{bmatrix} \Longrightarrow \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & * & * & * \\ & & * & * & * \end{bmatrix} \Longrightarrow \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & & * & * \\ & & & * & * \end{bmatrix} \Longrightarrow \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & & * & * \\ & & & & * \end{bmatrix}$$

We accomplish this by subtracting the appropriate multiple α of row p from each row i beneath it. This creates an equivalent system in which each entry below the diagonal in column p is zero. The element that we divide by is called the *pivot*. For each row i below the diagonal, we update

$$\alpha = \frac{a_{ip}}{a_{pp}}, \qquad a_{ij} = a_{ij} - \alpha\, a_{pj} \text{ for each } j, \qquad b_i = b_i - \alpha\, b_p$$

We can translate this directly into MATLAB code.

```
for i = p+1 : n
   alpha = A(i, p) / A(p, p);
   b(i, :) = b(i, :) - alpha * b(p, :);
   A(i, :) = A(i, :) - alpha * A(p, :);
end
```

2.5 Pivoting

The Gaussian elimination procedure described above can fail spectacularly if some pivot element a_{pp} is zero. This can happen even if the matrix is nonsingular. To avoid such cases we use the row interchange operation to replace row p with another row that has a nonzero in column p. A popular strategy, known as *partial pivoting*, is to choose the row that has the biggest (in absolute value) entry in column p among rows below the diagonal. (If *all* entries in column p below the main diagonal are zero, then the matrix is singular.) For reasons that will become apparent in Section 3, we apply this pivoting rule even if a_{pp} is nonzero.

```
q = p;
for i = p+1 : n
   if abs(A(i, p)) > abs(A(q, p))
      q = i;
   end
end
```

We can vectorize this using the `max` command.

```
[val q] = max(abs(A(p:n, p)));
q = q + p - 1;
```

2.6 MATLAB code for Gaussian elimination

Here is the complete code for Gaussian elimination with partial pivoting, packaged in a `.m` file. It solves $Ax = b$ where A is an n-by-n invertible matrix and b is a column vector of length n (or a matrix of such columns).

```
function x = lsolve(A, b)
% LSOLVE  Linear system of equation solver, bare bones version.
%   x = lsolve(A, b) returns the solution to the equation Ax = b,
%   where A is an n-by-n nonsingular matrix and b is a column
%   vector of length n (or a matrix with several such columns).
%   Written by KDW 8/19/2005

% Gaussian elimination with partial pivoting
[m n] = size(A);
for p = 1 : n

   % find index q of largest element below diagonal in column p
   [val q] = max(abs(A(p:n, p)));
   q = q + p - 1;

   % swap rows p and q
   A([p q], :) = A([q p], :);
   b([p q], :) = b([q p], :);

   % zero out entries of A and b using pivot A(p, p)
   for i = p+1 : n
      alpha = A(i, p) / A(p, p);
      b(i, :) = b(i, :) - alpha * b(p, :);
      A(i, :) = A(i, :) - alpha * A(p, :);
   end

end

% back substitution
x = zeros(size(b));
for i = n : -1 : 1
    x(i, :) = (b(i, :) - A(i, i+1:n) * x(i+1:n, :)) / A(i, i);
end
```

Computing the inverse. Given a non-singular matrix A, its *inverse* is the unique matrix B such that $AB = I$. The following code fragment computes the inverse by exploiting `lsolve`'s ability to handle a matrix right hand side.

```
>> B = lsolve(A, eye(size(A)));
```

In practice, we rarely need to compute an inverse. For example, the quantity $\rho = x^T A^{-1} x$ arises frequently in statistics. While it is tempting to compute the inverse explicitly, a more efficient and accurate method is to use `x' * lsolve(A, x)`.

2.7 Solving linear systems in MATLAB

In MATLAB the *backslash operator* solves linear systems of equations $Ax = b$.

```
>> A  = [0 1 1; 2 4 -2; 0 3 15];
>> b = [4; 2; 36];
>> x = A \ b
x =
    -1
     2
     2
```

The backslash operator uses a highly optimized implementation of Gaussian elimination with partial pivoting. It is around 20x faster than our MATLAB implementation. It is also more numerically robust than our implementation: it produces a warning if the system is nearly singular. *Use the built-in commands when available.*

2.8 LU Decomposition

An *LU decomposition* of a matrix A is $A = LU$, where L is a lower triangular matrix, and U is an upper triangular matrix. The algorithm for computing an LU decomposition is Gaussian elimination, with some extra bookkeeping to record L. Given the LU decomposition of a matrix A, we can efficiently solve a system of linear equations $Ax = b$ as follows:

- Solve $Ly = b$ for y via forward-substitution.
- Solve $Ux = y$ for x via back-substiution.

This process works because $Ax = LUx = Ly = b$, as required. We have traded solving one linear system of equations for solving two, but each of the two new systems is especially easy to solve.

The main reason for computing the LU decomposition is to repeatedly solve $Ax = b$ many times for the same matrix A, but different right-hand side vectors b. Solving $Ax = b$ from via Gaussian elimination takes time proportional to n^3, where n is the number of linear equations. Once we have computed the LU decomposition of A, solving $Ax = b$ takes time proportional to n^2.

```
>> A  = [0 1 1; 2 4 -2; 0 3 15];
>> [L, U] = lu(A);
>> b = [4; 2; 36];
>> y = L \ b;          % backslash uses forward-substitution
>> x = U \ y           % backslash uses back-substitution
x =
    -1
     2
     2
```

Caveat: the matrix L that `lu` returns may be a *permutation* of a lower triangular matrix (and not actually lower triangular). MATLAB calls such matrices "psychologically" lower triangular because, while they are not technically lower triangular, they have the same performance characteristics. By allowing this extra flexibility, we guarantee that the LU decomposition exists for any nonsingular matrix A.

2.9 Cholesky decomposition

A matrix A is *symmetric positive definite* if $A = A^T$ and $x^T Ax > 0$ for all nonzero vectors x. The *Cholesky decomposition* of a matrix A is $A = R^T R$ where R is upper triangular. The main application of a Cholesky decomposition is to solve systems of the form $Ax = b$, where A is symmetric positive definite.

```
>> A = [2 2 18; 2 24 -18; 18 -18 234];
>> b = [65; -62; 819];
>> R = chol(A);
>> y = R' \ b;
>> x = R \ y
x =
   17.0000
   -2.5000
    2.0000
```

The algorithm for computing a Cholesky decomposition is almost identical to the one for computing an LU decomposition except that (i) we don't need to worry about pivoting and (ii) we can save space and time by exploiting symmetry. Computing a Cholesky decomposition is twice as fast as computing an LU decomposition, and uses half as much extra space.

3 Numerical Analysis

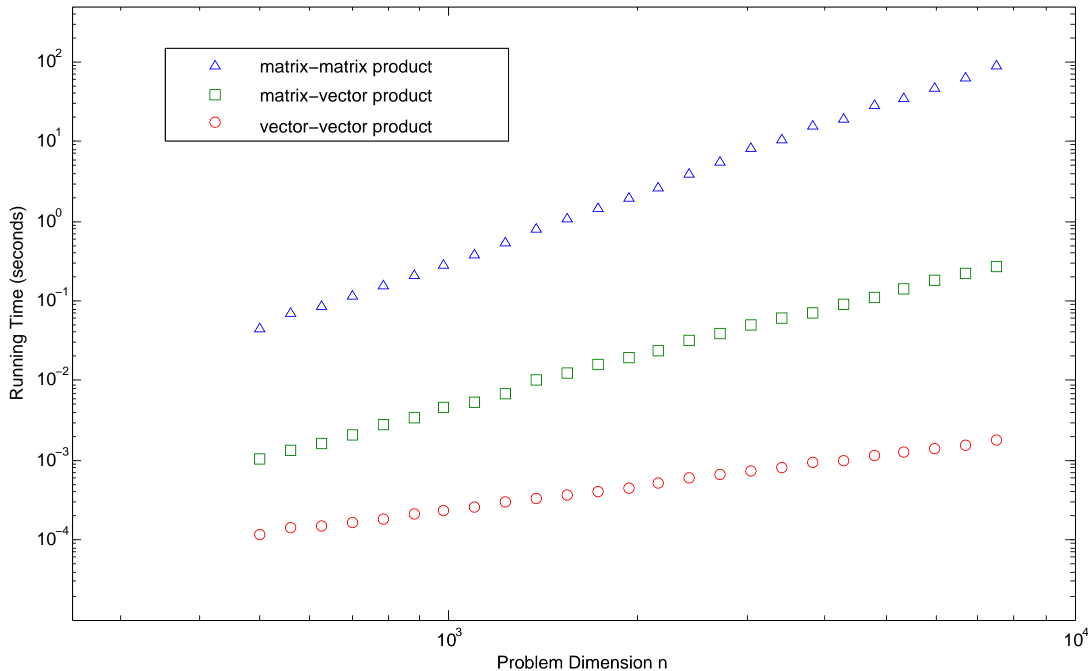
When designing algorithms, we demand that they produce "accurate" solutions; otherwise there is no point in executing them. We also want our algorithms to be fast enough to solve our problems using a reasonable amount of computing resources. Numerical analysis provides a framework for understanding these properties. We begin with running time since it is the easiest to measure.

3.1 Running time

Despite the amazing increase in speed of computers over the past few decades, we always thirst for more speed so that we can solve bigger problems. The script below performs a number of computational experiments to measure the running time of various matrix operations as a function of its input size n. It also plots the running time versus size using a log-log plot. The plots result in approximately straight lines: this leads to the prediction that the running time is of the form an^k for some constants k (the slope of the line) and a (where e^a is the x-intercept).

To measure the running time, we use the pair of commands `tic` (which starts a stopwatch timer) and `toc` (which return the elapsed time in seconds since the last call to `tic`). To setup the plot, we generate a logarithmically spaced interval of points using `logspace`. We use the command `loglog` to produce the log-log plot.

```
% adjust according to the speed of your computer
N = 25;                 % N logarithmically spaced points
MINN = 500;             % between MINN
MAXN = 7500;            % and MAXN

sizes = round(logspace(log10(MINN), log10(MAXN), N));

for i = 1 : N
    n = sizes(i);       % size of ith problem
    A = rand(n);
    b = rand(n, 1);

    tic;
    b' * b;             % linear time computation
    linear(i) = toc;

    tic;
    A * b;              % quadratic time computation
    quadratic(i) = toc;

    tic;
    A * A;              % cubic time computation
    cubic(i) = toc;
end

% plot the results
loglog(sizes, cubic, '^', sizes, quadratic, 's', sizes, linear, 'o');
axis([250 10000 1E-5 500]);
xlabel('Problem Dimension n');
ylabel('Running Time (seconds)');
legend('matrix-matrix', 'matrix-vector', 'vector-vector');
```



The slopes of the three lines are approximately 1, 2, and 3, respectively. Thus, given an n-by-n matrix and an n-by-n vector, we hypothesize that a vector-vector product takes time proportional to n, a matrix-vector product takes n^2 time and a matrix-matrix product takes n^3 time.

3.2 Precision and accuracy

Accuracy is how close an approximate value is to the quantity it is estimating. *Precision* is the number of digits that can be represented. Do not confuse precision with accuracy. 3.133333333 is an estimate of the mathematical constant π that is specified with 10 decimal digits of precision, but it has only two digits of accuracy.

3.3 IEEE binary floating point

MATLAB (and most programming languages) use IEEE double precision binary floating point to represent real numbers. This representation is a binary version of scientific notation. Each number is represented using 64 bits: 1 bit for the sign, 11 bits for the exponent, and 52 for the significand (or mantissa). For example, the number 14.5 is represented as $2^3 \times (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16})$. This handles a large range of real numbers from $\pm 4.94 \times 10^{-324}$ to $\pm 1.80 \times 10^{308}$ with 14 or 15 digits of accuracy. There are only about 2^{64} double precision binary floating point values, including 0, 1, 17, -0.8125, and 14.5 (but not 1/3, π or 1/10).

3.4 Roundoff error

Since the set of floating point values is finite, we do not expect to end up with the exact mathematical answer as we perform calculations. This phenomenon is known as *roundoff error*,

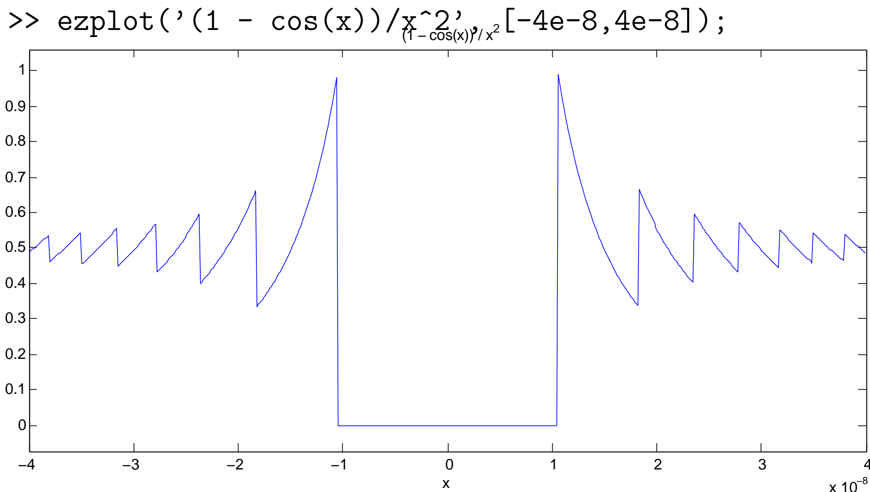
and it can lead to some surprising behavior for the uninitiated programmer.

```
>> if (0.1 + 0.2 == 0.3) fprintf('yes\n'); end      % always false

>> if (0.1 + 0.3 == 0.4) fprintf('yes\n'); end      % always true
yes
```

The reason is that 1/10 is not a floating point value, so `0.1` is rounded to the nearest floating point value. It is usually not a good idea to compare floating point values for exact equality.

Roundoff error may seem irrelevant, but the errors can accumulate in unexpected ways. For example, consider the following MATLAB plot of the function $f(x) == (1 - \cos(x))/x^2$ over the range $-4 \cdot 10^{-8} \le x \le 4 \cdot 10^{-8}$.

```
>> ezplot('(1 - cos(x))/x^2',[-4e-8,4e-8]);
```



The plot is extremely misleading since $f(x)$ is essentially a constant of value 0.5 over this interval. To investigate what has gone wrong, let's consider how `f(x)` is computed when `x = 1.1e-8`. The closest floating point value to `cos(x)` turns out to be

```
0.99999999999999988897769753748434595763683319091796875
```

This agrees with the exact answer to 16 decimal places.

```
0.9999999999999999395...
```

The problem arises when we compute `1 - cos(x)`. Using IEEE arithmetic, we obtain an answer of approximately `1.1102e-16`, which is no longer an accurate estimate of the exact answer `6.0500e-17`. When we divide by `x^2`, we obtain the floating point answer of `0.9175`, which is more than 80% larger than the exact answer (of about 1/2). This devastating loss of accuracy is known as *catastrophic cancellation*: it often arises when a small number is computed from larger ones ($1 - \cos x$ is computed from 1 and $\cos x$), which themselves are subject to errors ($\cos x$ is subject to roundoff error).

3.5 Conditioning

Informally, a problem is well-conditioned if small perturbations in the input x result in only correspondingly small perturbations in the output $f(x)$. More formally, a problem is *well-conditioned* if $f(x) \approx f(x+\epsilon)$ for *all* small perturbations ϵ. Otherwise it is called *ill-conditioned* or *ill-posed*. Conditioning is a property of the problem instance, not of any particular algorithm for solving it. Many familiar operations and functions are well-conditioned, including $f(x) = x^2$, $f(x) = e^x$, $f(x) = (1 - \cos x)/x^2$, $f(x_1, x_2) = x_1 + x_2$ and $f(x) = \int_0^x e^{-t^2} dt$.

The problem of finding a solution to $Ax = b$ can be either well-conditioned or ill-conditioned, depending on the matrix A. As an example, consider the following 2-by-2 system of equations.

$$\begin{bmatrix} 4.1 & 2.8 \\ 9.7 & 6.6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4.10 \\ 9.70 \end{bmatrix} \quad \Rightarrow \quad x = \begin{bmatrix} 1.00 \\ 0.00 \end{bmatrix}$$

If we perturb the righthand side by a tiny amount, the solution can change by a substantial amount.

$$\begin{bmatrix} 4.1 & 2.8 \\ 9.7 & 6.6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4.11 \\ 9.70 \end{bmatrix} \quad \Rightarrow \quad x = \begin{bmatrix} 0.34 \\ 0.97 \end{bmatrix}$$

The *condition number* $\kappa(A)$ of a matrix A is defined to be the ratio of the largest to smallest *singular value*. It bounds how much the solution x varies as a function of a small perturbation δb in the right hand side b. In particular

$$\frac{\| \delta x \|}{\| x \|} \leq \kappa(A) \frac{\| \delta b \|}{\| b \|}$$

This says that the relative error in the solution is bounded by the relative change in the right hand side, multiplied by the condition number. In the example above, the condition number of A is approximately 1623. This is consistent with the perturbation above: a relative change in the right hand side of bout 0.00095 induces a relative error in the solution of about 1.17, which is about a 1235x blowup (and obeys the condition number bound of a 1623x blowup). In general, if b has t digits of accuracy then x has $t - \log \kappa(A)$ digits of accuracy. If $\kappa(A) \approx 10^{15}$, we are in big trouble!

```
>> A = [4.1 2.8; 9.7 6.6];
>> cond(A)
ans =
   1.6230e+03
```

3.6 Stability

Informally, an algorithm is stable if the output of the algorithm changes by only a small amount when the input data changes by a small amount. More formally, an algorithm `fl(x)` for computing $f(x)$ is *numerically stable* if `fl(x)` $\approx f(x+\epsilon)$ for *some* small perturbation ϵ. In other words, the algorithm produces nearly the right answer to nearly the right problem. The

direct algorithm for computing $f(x) = (1 - \cos x)/x^2$ is not stable for values of x near 0. (See Exercise 2 for a stable way to compute it.)

Gaussian elimination with partial pivoting is a stable algorithm for solving $Ax = b$ (except on certain contrived examples that do not appear to arise in practice). This means that if the condition number is not too large, our code (and the backslash operator) is guaranteed to find an accurate answer. Gaussian elimination without pivoting is stable when the matrix is symmetric positive definite, but it is unstable in general. (See Exercise 3.)

3.7 Summary

Numerical analysis provides a framework for predicting the accuracy of computational methods. Accuracy depends on both stability and conditioning. Accuracy is guaranteed if you apply a stable algorithm to a well-conditioned problem. However, applying an unstable algorithm to a well-conditioned problem or a stable algorithm to an ill-conditioned problem can lead to inaccurate or meaningless solutions. Numerical analysis is the art and science of designing numerically stable algorithms for well-conditioned problems.

4 Other Numerical Linear Algebra Functions

Type `help matfun` to see the extensive list of numerical linear algebra functions supported by MATLAB. We review some of the most important ones below, including eigenvalues, singular values, and linear programming.

4.1 Spectral Decomposition

An *eigenvalue* and *eigenvector* of a square matrix A are a scalar λ and a non-zero vector x such that $Ax = \lambda x$. Solutions to the eigenvalue problem play a fundamental role in many scientific and engineering applications, from string vibrations to population dynamics, from bridge oscillations to oil exploration, from acoustics to materials analysis. The most important special case of the eigenvalue problem is when the matrix A is *symmetric*: $A = A^T$. In this case A has a *spectral decomposition* $A = VDV^T$, where D is the diagonal matrix of eigenvalues and V is the orthogonal matrix of eigenvectors (column j of V corresponds to the eigenvalue on the jth diagonal of D).

An example. Rigid bodies have preferred axes of rotation (e.g., spiraling football). The *inertia tensor* of a rigid body is a 3-by-3 symmetric matrix A determined from the distribution of mass of the body. The eigenvalues of the inertia tensor are the *moments of inertia* (resistance to angular acceleration) and the eigenvectors are the *principal axes of inertia* (angular velocities). If the following matrix represents the inertia tensor,

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

then its moments of inertia are 2, $2+\sqrt{2}$ and $2-\sqrt{2}$. The code fragment below computes the eignvalues and eigenvectors of this 3-by-3 matrix using the `eig` command.

```
>> A = [2 -1 0; -1 2 -1; 0 -1 2]
>> [V, D] = eig(A)
V =
    0.5000   -0.7071   -0.5000
    0.7071    0.0000    0.7071
    0.5000    0.7071   -0.5000

D =
    0.5858         0         0
         0    2.0000         0
         0         0    3.4142
```

4.2 Singular value decomposition

The singular value decomposition (SVD) is a central problems in numerical linear algebra which has numerous applications ranging from statistic to computer graphics, from signal processing to computational tomography, and from seismology to gene expression analysis. It is also used to compute the numerical rank and condition number of a matrix. The singular value decomposition of a real n-by$n$ matrix A is $A = USV^T$, where U and V are n-by-n orthogonal matrices and S is an n-by-n diagonal matrix (with the singular values in descending order), and V is an n-by-n orthogonal matrix. The command `svd(A)` returns the singular values of `A`.

```
>> A  = [0 1 1; 2 4 -2; 0 3 15];
>> svd(A)
ans =
   15.3908
    4.7978
    0.3250
```

Principal component analysis. Principal component analysis (PCA) approximates a matrix by a lower rank matrix. One of the SVDs most important properties is that the truncated SVD $A_r = U_r S_r V_r^T$ is the best rank r approximation to A, where U_r and V_r denote the first r columns of U and V, and S_r denotes the first r rows and columns of S. Here "best" means that $\| A_r - A \|_2$ is minimized over all rank r matrices. PCA is widely used in psychometrics, marketing, management, operations research, and other social and applied sciences that deal with large quantities of data.

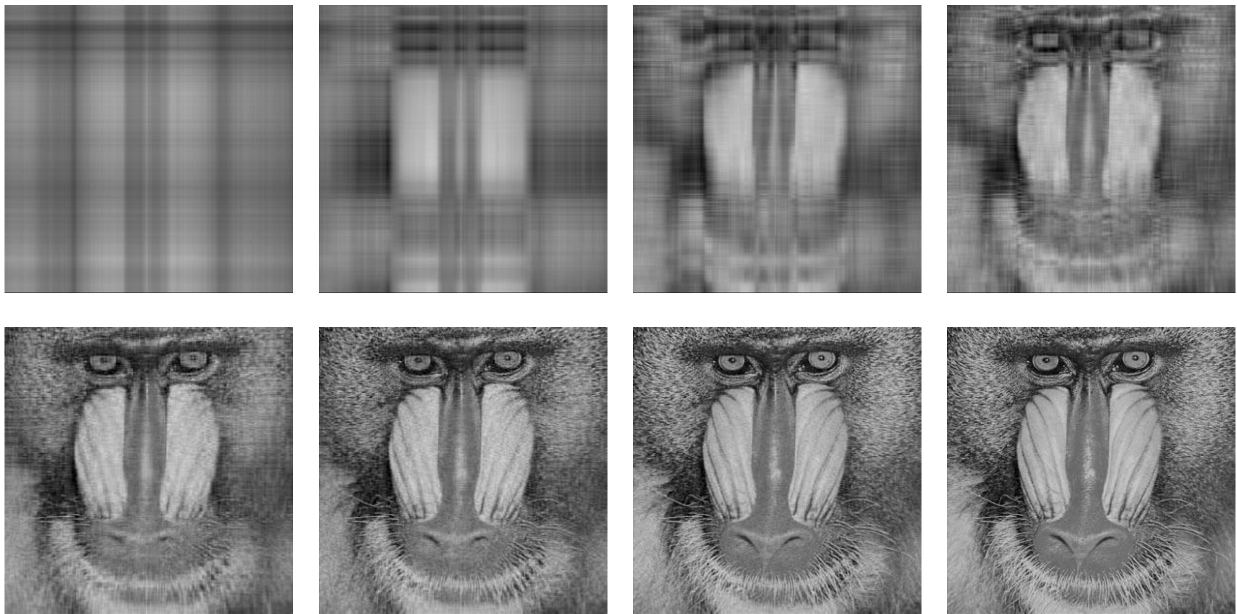
We now apply PCA to image processing. We treat an n-by-n grayscale image as an n-by-n matrix of grayscale intensities. The following code fragment reads in a JPEG image from a file, converts the image to grayscale, and then converts the image to a matrix (where each entry is between 0 and 255).

```
>> A = imread('baboon.jpg');       % read image from a file
>> A = rgb2gray(A);                % convert from color to grayscale
>> A = im2double(A);               % convert to double precision matrix
>> imshow(A);                      % display the image in a window
```

The next code fragment computes the SVD, use it to compute the best rank r approximation for various values of r, displays the resulting image, waits for the user to type any key, and saves it to a file of the form `baboon-5.jpg`.

```
[U, S, V] = svd(A);
for r = [1 2 5 10 25 50 100 298]
    Ar = U(:, 1:r) * S(1:r, 1:r) * V(:, 1:r)';
    imshow(Ar);
    pause;
    imwrite(Ar, sprintf('baboon-%d.jpg', r));
end
```

Below are the resulting pictures for r = 1, 2, 5, 10, 25, 50, 100 and 298. The last one is the original image. Even when the rank is relatively small, we get quite accurate descriptions of the image. This makes PCA useful for image compression and face recognition.



4.3 Linear Programming

Linear programming is a central problem in operations research: Given a matrix A, a vector b, and a vector c the goal problem is to find a vector x that maximizes $c^T x$ subject to $Ax \leq b$

and $x \geq 0$. It is the natural generalization of the $Ax = b$ problem to linear inequalities. Here is a small example.

$$\begin{array}{lrcrcr} \text{maximize} & 13x_1 & + & 23x_2 & & \\ \text{subject to:} & 5x_1 & + & 15x_2 & \leq & 480 \\ & 4x_1 & + & 4x_2 & \leq & 160 \\ & 35x_1 & + & 20x_2 & \leq & 1190 \\ & x_1 & , & x_2 & \geq & 0 \end{array}$$

The following code fragment solves this linear program. We use `-c` for the objective function parameter since the `linprog` command seeks to minimize $c^T x$, but we want to maximize. We leave the fourth and fifth parameters empty since these are to enforce equality constraints (which we don't have). The final two parameters are lower and upper bounds on the variables – we constrain x to be nonnegative.

```
>> A = [5 15; 4 4; 35 20];
>> b = [480; 160; 1190];
>> c = [13; 23];
>> lb = [0; 0];
>> ub = [inf; inf];
>> x = linprog(-c, A, b, [], [], lb, ub)
x =
   12.0000
   28.0000
```

4.4 Sparse Matrices

An n-by-n matrix is *sparse* if the number of nonzeros is proporitional to n. This includes matrices that are diagonal, tridiagonal, or banded, as well as sparse matrices with no particular pattern of nonzeros. Most large matrices that arise in practice are sparse. MATLAB provides specialized matrix routines for dealing with sparse matrices. These routines are significantly more efficient (both in terms of time and space) than their dense counterparts. For example, the amount of space required to store a sparse matrix is proportional to n instead of n^2; the amount of time to do a matrix-vector product is proportional to n instead of n^2. This can make the difference between solving a problem and not solving it. Type `help sparfun` to get information on using sparse matrices in MATLAB.

4.5 Rectangular Matrices

Although we have mainly focused on square matrices in this tutorial, most of MATLAB's matrix functions extend to non-square matrices, including the backslash operator, the LU decomposition, the singular value decomposition, and sparse matrices. For example when A is an over-determined system, `A \ b` returns a vector x that minimizes $\| Ax - b \|_2$, a least squares solution.

5 Exercises

1. *Google's PageRank algorithm.* Consider a web surfer who surfs from one web page to another by selecting a page at random. Suppose that if the surfer is currently visiting page j, the surfer will choose page i with probability $p_{ij} > 0$, where $\sum_i p_{ij} = 1$, and each choice is made indepenent of past choices. This model is known as a *Markov chain.* The n-by-n matrix P is called the *transition matrix.* The following is a sample transition matrix for a 3 page Web.

$$P = \begin{bmatrix} 0.3 & 0.4 & 0.5 \\ 0.3 & 0.4 & 0.3 \\ 0.4 & 0.2 & 0.2 \end{bmatrix}$$

   (a) Given a transition matrix P, verify that all column sums equal 1.

   (b) Create a MATLAB function `unit.m` so that `unit(i, n)` creates a column vector e_i of length n whose ith entry is 1 and whose other entries are 0.

   (c) Given that you start in state i, write a MATLAB code fragment (type commands directly into MATLAB) to compute the distribution of states that you will end up in after one step. Mathematically, the answer is Pe_i.

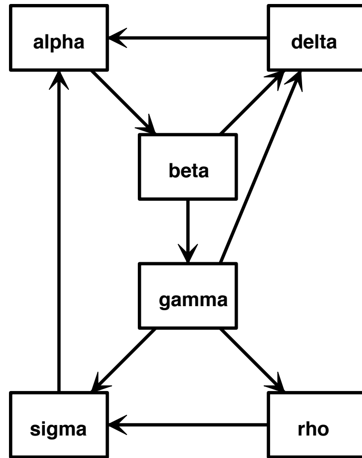
   (d) Given that you start in state i, write a MATLAB code fragment to compute the distribution of states that you will end up in after k steps. Do this by initializing $x = e_i$ and then iterating $x = Px$, k times. (This is equivalent to computing $P^k e_i$.) What do you observe? How does the answer depend on the starting state i? Now, assume you start in each state with probability $1/n$, i.e., initialize x to $(1/n, 1/n, \ldots, 1/n)$. What is the distribution of states you will end up in after k steps?

   (e) What fraction of the time will the random surfer spend in each state? Under general conditions, this distribution is unique and is known as the *stationary distribution.* It is the vector x that satisfies $Px = x$, or equivalently $(P - I)x = 0$, subject to the normalization constraint $\sum_i x_i = 1$. The matrix $(P-I)$ has one redundant equation, so if you replace (any) equation with $\sum_i x_i = 1$, you will have a linear system of n equations in n unknowns that can be solved by Gaussian elimination. Write a MATLAB function `stationary.m` so that `stationary(P)` return the stationary distribution of the transition matrix P.

   (f) Imagine now that the surfer goes from one page to another page by randomly choosing either (i) an outgoing hyperlink from the current page with probability 0.85 or (ii) a completely random page with probability 0.15. The fraction of time a surfer spends on a particular page is its PageRank. A page is deemed important (by Google) if it has a high PageRank. Note that a page may have high PageRank even it has few incoming hyperlinks, provided those hyperlinks have high PageRank.

   One way to represent the hyperlink structure of the Web is by its *connectivity matrix* G, a matrix with $g_{ij} = 1$ if there is a hyperlink from page j to i and 0 otherwise.

Given a connectivity matrix G, write a MATLAB function `pagerank.m` so that `pagerank(G)` computes the PAGERANK of each web page. (For simplicity, you may assume each website has at least one outgoing hyperlink.)



$$G = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

```
>> % the connectivity matrix
>> G = [ 0, 0, 0, 1, 0, 1;
         1, 0, 0, 0, 0, 0;
         0, 1, 0, 0, 0, 0;
         0, 1, 1, 0, 0, 0;
         0, 0, 1, 0, 0, 0;
         0, 0, 1, 0, 1, 0 ];

>> % a cell array whose entries are strings
>> urls = {'http://alpha.com',
           'http://beta.com',
           'http://gamma.com',
           'http://delta.com',
           'http://rho.com',
           'http://sigma.com' };

>> x = pagerank(G)
x =
    0.2675
    0.2524
    0.1323
    0.1697
    0.0625
    0.1156
```

(g) Download `princeton500.mat` and `princeton1000.mat` from either Blackboard or

`http://www.cs.princeton.edu/~wayne/princeton500.mat`

```
http://www.cs.princeton.edu/~wayne/princeton1000.mat
```

and put them into your working directory. The file `princeton500.mat` contains two variables: a column vector of strings containing the names of 500 web pages and a 500-by-500 matrix containing the connectivity matrix of those web pages. The file `princeton1000.mat` is a larger example, comprised of 1,000 web pages. To load the two variables in your workspace and display their properties, type:

```
>> load princeton500.mat;
>> whos;
```

Now, compute the PageRank of this data using your function `pagerank.m`.

(h) Print out the URLs in descending order of PageRank. The following code fragment prints out the PageRanks and URLs in the original order. The `fprintf` command is useful for printing formatted string output. We need to use `urls{i}` instead of `urls(i)` since `urls` is a *cell array*.

```
for i = 1 : n
   fprintf('%8.4f %s\n', x(i), urls{i});
end
```

Use the MATLAB command `sort(x)` to sort the entries of `x` in ascending order. The challenge here is that you need to rearrange the URLs too. Type `help sort` to learn about applying the command to this task. *Hint*: you will need to use the second return argument.

(i) An *eigenvalue* and *eigenvector* of a square matrix A are a scalar λ and a non-zero vector x such that $Ax = \lambda x$. Observe that the stationary distribution of a transition matrix is an eigenvector corresponding to the eigenvalue $\lambda = 1$, scaled so that its components sum to 1. In fact, $\lambda = 1$ is guaranteed to be the *largest eigenvalue* and the corresponding eigenvector is *unique*. In MATLAB, the statement `[x lambda] = eigs(A, 1)` returns the largest (in magnitude) eigenvalue `lambda` and a corresponding eigenvector `x` of a square matrix `A`. Use the command `eigs` to compute the stationary distribution of a transition matrix `P`.

Remark: one classic algorithm for finding the principal eigenvector is known as the *power method*. In our application, the power method boils down to starting with any unit vector x and repeatedly replacing it with Ax. This explains why the sequence of iterates in (d) converges to the stationary distribution. Iterative approaches like this are especially useful when the matrix A is large and sparse (or has special structure). In Google's case, A is a 4 billion-by-4 billion matrix, so exploiting special structure is essential.

2. Devise a stable algorithm for computing $f(x) = (1-\cos x)/x^2$. *Hint:* the straightforward algorithm is stable, except when x is close to 0 (or a multiple of 2π). In such cases, use the Taylor series approximation $\cos x \approx 1 - x^2/2 + x^4/24 - x^6/720$, and estimate $f(x) \approx 1/2 - x^2/24 + x^4/720$.

3. Consider the following linear system of two equations in two unknowns.

$$\begin{aligned} \epsilon x_1 \quad &+x_2 &= 1 \\ x_1 \quad &+2x_2 &= 3 \end{aligned}$$

   Suppose $\epsilon = 10^{-17}$. Verify that if you apply Gaussian elimination with no pivoting, you obtain the solution $x = [0.0\ 1.0]$, but the exact answer is approximately [1.0 1.0], so we failed to achieve even one digit of accuracy for x_1. Show that this is as a result of an unstable algorithm by verifying that the problem is well-conditioned (condition number around 5.8).

References

The following is an excellent introduction to numerical computing using MATLAB. The primary source of material in this tutorial is Chapters 1 and 2.

[1] Moler, Cleve. *Numerical Computing.* Society of Industrial and Applied Mathematics, 2004. `http://www.mathworks.com/moler/chapters.html`

For an excellent introduction to linear algebra, we recommend:

[2] Strang, Gilbert. *Linear Algebra and Its Applications.* Brooks Cole, 2005.

The "bible" of numerical linear algebra is:

[3] Golub, Gene H. and Van Loan, Charles F. *Matrix Computations.* The Johns Hopkins University Press, 1989.

For a wealth of information on Google's PageRank algorithm, see:

[4] Langville, Amy N. and Meyer, Carl D. *Google's PageRank and Beyond: The Science of Search Engine Rankings.* Princeton University Press, 2006.