Part II
Linear Algebra

Lecture 8

Matrices and Matrix Operations in Matlab

You should review the vector operations in Lecture 1.

Matrix operations

Recall how to multiply a matrix A times a vector $\mathbf{v}$:

$$A\mathbf{v} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} -1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \cdot (-1) + 2 \cdot 2 \\ 3 \cdot (-1) + 4 \cdot 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}.$$

This is a special case of matrix multiplication. To multiply two matrices, A and B you proceed as follows:

$$AB = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} -1 & -2 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} -1+4 & -2+2 \\ -3+8 & -6+4 \end{pmatrix} = \begin{pmatrix} 3 & 0 \\ 5 & -2 \end{pmatrix}.$$

Here both A and B are 2×2 matrices. Matrices can be multiplied together in this way provided that the number of columns of A match the number of rows of B. We always list the size of a matrix by rows, then columns, so a 3×5 matrix would have 3 rows and 5 columns. So, if A is $m \times n$ and B is $p \times q$, then we can multiply AB if and only if $n = p$. A column vector can be thought of as a $p \times 1$ matrix and a row vector as a $1 \times q$ matrix. Unless otherwise specified we will assume a vector $\mathbf{v}$ to be a column vector and so $A\mathbf{v}$ makes sense as long as the number of columns of A matches the number of entries in $\mathbf{v}$.

Printing matrices on the screen takes up a lot of space, so you may want to use

```
≫ format compact
```

Enter a matrix into Matlab either as

```
≫ A = [ 1   3 -2 5 ;  -1   -1 5 4 ; 0 1 -9   0]
```

or

```
≫ A = [1,3,-2,5; -1,-1,5,4; 0,1,-9,0]
```

Also enter a vector $\mathbf{u}$:

```
≫ u = [ 1   2   3   4]'
```

To multiply a matrix times a vector $A\mathbf{u}$ use `*`:

```
≫ A*u
```

Since A is 3 by 4 and $\mathbf{u}$ is 4 by 1 this multiplication is valid and the result is a 3 by 1 vector.

Now enter another matrix B using

```
≫ B = [3 2 1; 7 6 5; 4 3 2]
```

You can multiply B times A with

```
≫ B*A
```

but A times B is not defined and

```
≫ A*B
```

will result in an error message.

You can multiply a matrix by a scalar:

```
≫ 2*A
```

Adding matrices $A + A$ will give the same result:

```
≫ A + A
```

You can even add a number to a matrix:

```
≫ A + 3    % add 3 to every entry of A
```

Component-wise operations

Just as for vectors, adding a '`.`' before '`*`', '`/`', or '`^`' produces entry-wise multiplication, division and exponentiation. If you enter

```
≫ B*B
```

the result will be BB, i.e. matrix multiplication of B times itself. But, if you enter

```
≫ B.*B
```

the entries of the resulting matrix will contain the squares of the same entries of B. Similarly if you want B multiplied by itself 3 times then enter

```
≫ B^3
```

but, if you want to cube all the entries of B then enter

```
≫ B.^3
```

Note that `B*B` and `B^3` only make sense if B is square, but `B.*B` and `B.^3` make sense for any size matrix.

The identity matrix and the inverse of a matrix

The $n \times n$ *identity matrix* is a square matrix with ones on the diagonal and zeros everywhere else. It is called the identity because it plays the same role that 1 plays in multiplication, i.e.

$$AI = A, \qquad IA = A, \qquad I\mathbf{v} = \mathbf{v}$$

for any matrix A or vector $\mathbf{v}$ where the sizes match. An identity matrix in MATLAB is produced by the command

```
≫ I = eye(3)
```

A square matrix A can have an *inverse* which is denoted by A^{-1}. The definition of the inverse is that

$$AA^{-1} = I \qquad \text{and} \qquad A^{-1}A = I.$$

In theory an inverse is very important, because if you have an equation

$$A\mathbf{x} = \mathbf{b}$$

where A and $\mathbf{b}$ are known and $\mathbf{x}$ is unknown (as we will see, such problems are very common and important) then the theoretical solution is

$$\mathbf{x} = A^{-1}\mathbf{b}.$$

We will see later that this is not a practical way to solve an equation, and A^{-1} is only important for the purpose of derivations. In MATLAB we can calculate a matrix's inverse very conveniently:

```
≫ C = randn(5,5)
≫ inv(C)
```

However, not all square matrices have inverses:

```
≫ D = ones(5,5)
≫ inv(D)
```

The "Norm" of a matrix

For a vector, the "norm" means the same thing as the length (geometrically, not the number of entries). Another way to think of it is how far the vector is from being the zero vector. We want to measure a matrix in much the same way and the *norm* is such a quantity. The usual definition of the norm of a matrix is

Definition 1 *Suppose A is a $m \times n$ matrix. The norm of A is*

$$\|A\| \equiv \max_{\|\mathbf{v}\|=1} \|A\mathbf{v}\|.$$

The maximum in the definition is taken over all vectors with length 1 (unit vectors), so the definition means the largest factor that the matrix stretches (or shrinks) a unit vector. This definition seems cumbersome at first, but it turns out to be the best one. For example, with this definition we have the following inequality for any vector $\mathbf{v}$:

$$\|A\mathbf{v}\| \leq \|A\|\|\mathbf{v}\|.$$

In MATLAB the norm of a matrix is obtained by the command

```
≫ norm(A)
```

For instance the norm of an identity matrix is 1:

```
≫ norm(eye(100))
```

and the norm of a zero matrix is 0:

```
≫ norm(zeros(50,50))
```

For a matrix the norm defined above and calculated by MATLAB is not the square root of the sum of the square of its entries. That quantity is called the *Froebenius norm*, which is also sometimes useful, but we will not need it.

Some other useful commands

`C = rand(5,5)` ..... random matrix with uniform distribution in $[0,1]$.
`size(C)` ..... gives the dimensions $(m \times n)$ of C.
`det(C)` ..... the determinant of the matrix.
`max(C)` ..... the maximum of each column.
`min(C)` ..... the minimum in each column.
`sum(C)` ..... sums each column.
`mean(C)` ..... the average of each column.
`diag(C)` ..... just the diagonal elements.
`C'` ..... tranpose the matrix.

In addition to `ones`, `eye`, `zeros`, `rand` and `randn`, MATLAB has several other commands that automatically produce special matrices:
`hilb(6)`
`pascal(5)`

Exercises

8.1 Enter the matrix A by

```
≫ A = [3 2 1; 6 5 4; 9 8 7]
```

and also the matrix

$$B = \begin{bmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix}.$$

Find (a) `A*A`, (b) `A^2`, (c) `A.^2`, (d) `A.*B`, (e) `A*B`. Turn in the output.

8.2 By hand, calculate $A\mathbf{v}$, AB, and BA for:

$$A = \begin{pmatrix} 2 & 4 & -1 \\ -2 & 1 & 9 \\ -1 & -1 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & -1 & -1 \\ 1 & 0 & 2 \\ -1 & -2 & 0 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 3 \\ 1 \\ -1 \end{pmatrix}.$$

Check the results using MATLAB. Think about how fast computers are. Turn in your hand work.

8.3 Write a well-commented MATLAB **function** program `myinvcheck` that

- makes a $n \times n$ random matrix (normally distributed, `A = randn(n,n)`),
- calculates its inverse (`B = inv(A)`),
- multiplies the two back together,
- calculates the residual (difference between AB and `eye(n)`), and
- returns the scalar residual (norm of the difference).

Turn in your program.

8.4 Write a well-commented MATLAB **script** program `myinvcheckplot` that calls `myinvcheck` for $n = 10, 20, 40, \ldots, 2^i 10$ for some moderate i, records the results of each trial, and plots the scalar residual versus `n` using a log plot. (See `help loglog`.)

What happens to the scalar residual as n gets big? Turn in the program, the plot, and a very brief report on the results of your experiments. (Do not print any large random matrices.)

Lecture 9

Introduction to Linear Systems

How linear systems occur

Linear systems of equations naturally occur in many places in engineering, such as structural analysis, dynamics and electric circuits. Computers have made it possible to quickly and accurately solve larger and larger systems of equations. Not only has this allowed engineers to handle more and more complex problems where linear systems naturally occur, but has also prompted engineers to use linear systems to solve problems where they do not naturally occur such as thermodynamics, internal stress-strain analysis, fluids and chemical processes. It has become standard practice in many areas to analyze a problem by transforming it into a linear systems of equations and then solving those equation by computer. In this way, computers have made linear systems of equations the most frequently used tool in modern engineering.

In Figure 9.1 we show a truss with equilateral triangles. In Statics you may use the "method of joints" to write equations for each node of the truss[1]. This set of equations is an example of a linear system. Making the approximation $\sqrt{3}/2 \approx .8660$, the equations for this truss are

$$\begin{aligned}
.5\,T_1 + T_2 &= R_1 = f_1 \\
.866\,T_1 &= -R_2 = -.433\,f_1 - .5\,f_2 \\
-.5\,T_1 + .5\,T_3 + T_4 &= -f_1 \\
.866\,T_1 + .866\,T_3 &= 0 \\
-T_2 - .5\,T_3 + .5\,T_5 + T_6 &= 0 \\
.866\,T_3 + .866\,T_5 &= f_2 \\
-T_4 - .5\,T_5 + .5\,T_7 &= 0,
\end{aligned} \tag{9.1}$$

where T_i represents the tension in the i-th member of the truss.

You could solve this system by hand with a little time and patience; systematically eliminating variables and substituting. Obviously, it would be a lot better to put the equations on a computer and let the computer solve it. In the next few lectures we will learn how to use a computer effectively to solve linear systems. The first key to dealing with linear systems is to realize that they are equivalent to matrices, which contain numbers, not variables.

As we discuss various aspects of matrices, we wish to keep in mind that the matrices that come up in engineering systems are *really large*. It is not unusual in real engineering to use matrices whose dimensions are in the thousands! It is frequently the case that a method that is fine for a 2×2 or 3×3 matrix is totally inappropriate for a 2000×2000 matrix. We thus want to emphasize methods that work for large matrices.

[1] See http://en.wikipedia.org/wiki/Truss or http://en.wikibooks.org/wiki/Statics for reference.

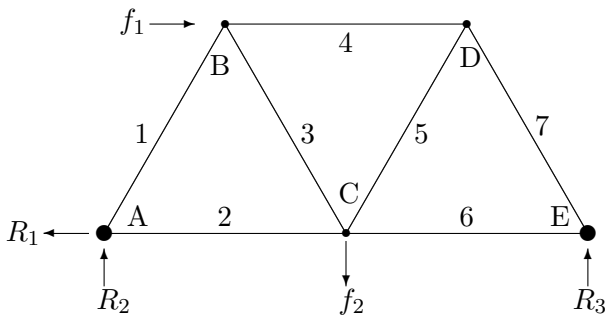

Figure 9.1: An equilateral truss. Joints or nodes are labeled alphabetically, A, B, ... and Members (edges) are labeled numerically: 1, 2, ... The forces f_1 and f_2 are applied loads and R_1, R_2 and R_3 are reaction forces applied by the supports.

Linear systems are equivalent to matrix equations

The system of linear equations

$$\begin{aligned} x_1 - 2x_2 + 3x_3 &= 4 \\ 2x_1 - 5x_2 + 12x_3 &= 15 \\ 2x_2 - 10x_3 &= -10 \end{aligned}$$

is equivalent to the matrix equation

$$\begin{pmatrix} 1 & -2 & 3 \\ 2 & -5 & 12 \\ 0 & 2 & -10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 15 \\ -10 \end{pmatrix},$$

which is equivalent to the *augmented matrix*

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 2 & -5 & 12 & 15 \\ 0 & 2 & -10 & -10 \end{array}\right).$$

The advantage of the augmented matrix, is that it contains only numbers, not variables. The reason this is better is because computers are much better in dealing with numbers than variables. To solve this system, the main steps are called *Gaussian elimination* and *back substitution*.

The augmented matrix for the equilateral truss equations (9.1) is given by

$$\left(\begin{array}{ccccccc|c} .5 & 1 & 0 & 0 & 0 & 0 & 0 & f_1 \\ .866 & 0 & 0 & 0 & 0 & 0 & 0 & -.433f_1 - .5\, f_2 \\ -.5 & 0 & .5 & 1 & 0 & 0 & 0 & -f_1 \\ .866 & 0 & .866 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -.5 & 0 & .5 & 1 & 0 & 0 \\ 0 & 0 & .866 & 0 & .866 & 0 & 0 & f_2 \\ 0 & 0 & 0 & -1 & -.5 & 0 & .5 & 0 \end{array}\right). \tag{9.2}$$

Notice that a lot of the entries are 0. Matrices like this, called *sparse*, are common in applications and there are methods specifically designed to efficiently handle sparse matrices.

Triangular matrices and back substitution

Consider a linear system whose augmented matrix happens to be

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 0 & -1 & 6 & 7 \\ 0 & 0 & 2 & 4 \end{array}\right). \tag{9.3}$$

Recall that each row represents an equation and each column a variable. The last row represents the equation $2x_3 = 4$. The equation is easily solved, i.e. $x_3 = 2$. The second row represents the equation $-x_2 + 6x_3 = 7$, but since we know $x_3 = 2$, this simplifies to: $-x_2 + 12 = 7$. This is easily solved, giving $x_2 = 5$. Finally, since we know x_2 and x_3, the first row simplifies to: $x_1 - 10 + 6 = 4$. Thus we have $x_1 = 8$ and so we know the whole solution vector: $\mathbf{x} = \langle 8, 5, 2\rangle$. The process we just did is called *back substitution*, which is both efficient and easily programmed. The property that made it possible to solve the system so easily is that A in this case is *upper triangular*. In the next section we show an efficient way to transform an augmented matrix into an upper triangular matrix.

Gaussian Elimination

Consider the matrix

$$A = \left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 2 & -5 & 12 & 15 \\ 0 & 2 & -10 & -10 \end{array}\right).$$

The first step of Gaussian elimination is to get rid of the 2 in the (2,1) position by subtracting 2 times the first row from the second row, i.e. (new 2nd = old 2nd - (2) 1st). We can do this because it is essentially the same as adding equations, which is a valid algebraic operation. This leads to

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 0 & -1 & 6 & 7 \\ 0 & 2 & -10 & -10 \end{array}\right).$$

There is already a zero in the lower left corner, so we don't need to eliminate anything there. To eliminate the third row, second column, we need to subtract -2 times the second row from the third row, (new 3rd = old 3rd - (-2) 2nd), to obtain

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 0 & -1 & 6 & 7 \\ 0 & 0 & 2 & 4 \end{array}\right).$$

This is now just exactly the matrix in equation (9.3), which we can now solve by back substitution.

Matlab's matrix solve command

In MATLAB the standard way to solve a system $A\mathbf{x} = \mathbf{b}$ is by the command

```
≫ x = A \ b
```

This syntax is meant to suggest dividing by A from the left as in

$$A\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad A\backslash A\mathbf{x} = A\backslash\mathbf{b} \quad \Leftrightarrow \quad \mathbf{x} = A\backslash\mathbf{b}\,.$$

Such division is not meaningful mathematically, but it helps for remembering the syntax.

This command carries out Gaussian elimination and back substitution. We can do the above computations as follows:

```
≫ A = [1 -2 3 ; 2 -5 12 ; 0 2 -10]
≫ b = [4 15 -10]'
≫ x = A \ b
```

Next, use the MATLAB commands above to solve $A\mathbf{x} = \mathbf{b}$ when the augmented matrix for the system is

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{array}\right),$$

by entering

```
≫ x1 = A \ b
```

Check the result by entering

```
≫ A*x1 - b
```

You will see that the resulting answer satisfies the equation exactly. Next try solving using the inverse of A:

```
≫ x2 = inv(A)*b
```

This answer can be seen to be inaccurate by checking

```
≫ A*x2 - b
```

Thus we see one of the reasons why the inverse is never used for actual computations, only for theory.

Exercises

9.1 Set $f_1 = 1000N$ and $f_2 = 5000N$ in the equations (9.1) for the equailateral truss. Input the coefficient matrix A and the right hand side vector b in (9.2) into MATLAB. Solve the system using the command \ to find the tension in each member of the truss. Save the matrix A as `A_equil_truss` and keep it for later use. (Enter `save A_equil_truss A`.) Print out and turn in A, $\mathbf{b}$ and the solution $\mathbf{x}$.

9.2 Write each system of equations as an augmented matrix, then find the solutions using *Gaussian elimination and back substitution* (i.e. the exact algorithm in this chapter). Check your solutions using MATLAB.

(a)

$$\begin{aligned} x_1 + x_2 &= 2 \\ 4x_1 + 5x_2 &= 10 \end{aligned}$$

(b)

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= -1 \\ 4x_1 + 7x_2 + 14x_3 &= 3 \\ x_1 + 4x_2 + 4x_3 &= 1 \end{aligned}$$

Lecture 10

Some Facts About Linear Systems

Some inconvenient truths

In the last lecture we learned how to solve a linear system using Matlab. Input the following:

```
>> A = ones(4,4)
>> b = randn(4,1)
>> x = A \ b
```

As you will find, there is no solution to the equation $A\mathbf{x} = \mathbf{b}$. This unfortunate circumstance is mostly the fault of the matrix, A, which is too simple, its columns (and rows) are all the same. Now try

```
>> b = ones(4,1)
>> x = [ 1 0 0 0]'
>> A*x
```

So the system $A\mathbf{x} = \mathbf{b}$ does have a solution. Still unfortunately, that is not the only solution. Try

```
>> x = [ 0 1 0 0]'
>> A*x
```

We see that this x is also a solution. Next try

```
>> x = [ -4   5   2.27  -2.27]'
>> A*x
```

This x is a solution! It is not hard to see that there are endless possibilities for solutions of this equation.

Basic theory

The most basic theoretical fact about linear systems is

Theorem 1 *A linear system $A\mathbf{x} = \mathbf{b}$ may have 0, 1, or infinitely many solutions.*

In most (but not all!) engineering applications we would want to have exactly one solution. The following two theorems tell us exactly when we can and cannot expect this.

Theorem 2 *Suppose A is a square $(n \times n)$ matrix. The following are all equivalent:*

1. *The equation* $A\mathbf{x} = \mathbf{b}$ *has exactly one solution for any* $\mathbf{b}$.
2. $det(A) \neq 0$.
3. *A has an inverse.*
4. *The only solution of* $A\mathbf{x} = \mathbf{0}$ *is* $\mathbf{x} = \mathbf{0}$.
5. *The columns of A are linearly independent (as vectors).*
6. *The rows of A are linearly independent.*

If A has these properties then it is called **non-singular**.

On the other hand, a matrix that does not have these properties is called **singular**.

Theorem 3 *Suppose A is a square matrix. The following are all equivalent:*

1. *The equation* $A\mathbf{x} = \mathbf{b}$ *has 0 or* ∞ *many solutions depending on* $\mathbf{b}$.
2. $\det(A) = 0$.
3. *A does not have an inverse.*
4. *The equation* $A\mathbf{x} = \mathbf{0}$ *has solutions other than* $\mathbf{x} = \mathbf{0}$.
5. *The columns of A are linearly dependent as vectors.*
6. *The rows of A are linearly dependent.*

To see how the two theorems work, define two matrices (type in `A1` then scroll up and modify to make `A2`) :

$$\texttt{A1} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \qquad \texttt{A2} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{pmatrix},$$

and two vectors:

$$\texttt{b1} = \begin{pmatrix} 0 \\ 3 \\ 6 \end{pmatrix}, \qquad \texttt{b2} = \begin{pmatrix} 1 \\ 3 \\ 6 \end{pmatrix}.$$

First calculate the determinants of the matrices:

```
≫ det(A1)
≫ det(A2)
```

Then attempt to find the inverses:

```
≫ inv(A1)
≫ inv(A2)
```

Which matrix is singular and which is non-singular? Finally, attempt to solve all the possible equations $A\mathbf{x} = \mathbf{b}$:

```
≫ x = A1 \ b1
≫ x = A1 \ b2
≫ x = A2 \ b1
≫ x = A2 \ b2
```

As you can see, equations involving the non-singular matrix have one and only one solution, but equation involving a singular matrix are more complicated.

The Residual

Recall that the residual for an approximate solution x of an equation $f(x) = 0$ is defined as $r = \|f(x)\|$. It is a measure of how close the equation is to being satisfied. For a linear system of equations we define the residual vector of an approximate solution $\mathbf{x}$ by

$$\mathbf{r} = A\mathbf{x} - \mathbf{b}.$$

If the solution vector $\mathbf{x}$ were exactly correct, then $\mathbf{r}$ would be exactly the zero vector. The size (norm) of $\mathbf{r}$ is an indication of how close we have come to solving $A\mathbf{x} = \mathbf{b}$. We will refer to this number as the **scalar residual** or just the **residual** of the approximation solution:

$$r = \|A\mathbf{x} - \mathbf{b}\|. \tag{10.1}$$

Exercises

10.1 By hand, find all the solutions (if any) of the following linear system using the augmented matrix and Gaussian elimination (following exactly the algorithm in the notes):

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 4, \\ 4x_1 + 5x_2 + 6x_3 &= 10, \\ 7x_1 + 8x_2 + 9x_3 &= 14\,. \end{aligned}$$

Try solving this system in MATLAB using the command `x = A \ b`. What happens? Turn in your hand work.

10.2 (a) Write a well-commented MATLAB **function** program `mysolvecheck` with input a number `n` that makes a random $n \times n$ matrix A and a random vector $\mathbf{b}$, solves the linear system $A\mathbf{x} = \mathbf{b}$, calculates the scalar residual $r = \|A\mathbf{x} - \mathbf{b}\|$, and outputs that number as r.

(b) Write a well-commented MATLAB **script** program that calls `mysolvecheck` 10 times each for $n = 5$, 10, 20, 40, 80, and 160, then records and averages the results and makes a log-log plot of the average r vs. n. Once your program is running correctly, increase the maximum n (by factors of 2) until the program stops running within 5 minutes.

Turn in the plot and the two programs. (Do not print any large random matrices.)

Lecture 11

Accuracy, Condition Numbers and Pivoting

In this lecture we will discuss two separate issues of accuracy in solving linear systems. The first, *pivoting*, is a method that ensures that Gaussian elimination proceeds as accurately as possible. The second, *condition number*, is a measure of how bad a matrix is. We will see that if a matrix has a bad condition number, the solutions are unstable with respect to small changes in data.

The effect of rounding

All computers store numbers as finite strings of binary floating point digits (bits). This limits numbers to a fixed number of significant digits and implies that after even the most basic calculations, rounding must happen.

Consider the following exaggerated example. Suppose that our computer can only store 2 significant digits and it is asked to do Gaussian elimination on

$$\left(\begin{array}{cc|c} .001 & 1 & 3 \\ 1 & 2 & 5 \end{array}\right).$$

Doing the elimination exactly would produce

$$\left(\begin{array}{cc|c} .001 & 1 & 3 \\ 0 & -998 & -2995 \end{array}\right),$$

but rounding to 2 digits, our computer would store this as

$$\left(\begin{array}{cc|c} .001 & 1 & 3 \\ 0 & -1000 & -3000 \end{array}\right).$$

Backsolving this reduced system gives

$$x_1 = 0 \quad \text{and} \quad x_2 = 3.$$

This seems fine until you realize that backsolving the unrounded system gives

$$x_1 = -1 \quad \text{and} \quad x_2 = 3.001.$$

Row Pivoting

A way to fix the problem is to use pivoting, which means to switch rows of the matrix. Since switching rows of the augmented matrix just corresponds to switching the order of the equations, no harm is done:

$$\left(\begin{array}{cc|c} 1 & 2 & 5 \\ .001 & 1 & 3 \end{array}\right).$$

Exact elimination would produce

$$\left(\begin{array}{cc|c} 1 & 2 & 5 \\ 0 & .998 & 2.995 \end{array}\right).$$

Storing this result with only 2 significant digits gives

$$\left(\begin{array}{cc|c} 1 & 2 & 5 \\ 0 & 1 & 3 \end{array}\right).$$

Now backsolving produces

$$x_1 = -1 \quad \text{and} \quad x_2 = 3,$$

which is the true solution (rounded to 2 significant digits).

The reason this worked is because 1 is bigger than 0.001. To pivot we switch rows so that the largest entry in a column is the one used to eliminate the others. In bigger matrices, after each column is completed, compare the diagonal element of the next column with all the entries below it. Switch it (and the entire row) with the one with greatest absolute value. For example in the following matrix, the first column is finished and before doing the second column, pivoting should occur since $|-2| > |1|$:

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 0 & 1 & 6 & 7 \\ 0 & -2 & -10 & -10 \end{array}\right).$$

Pivoting the 2nd and 3rd rows would produce

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 0 & -2 & -10 & -10 \\ 0 & 1 & 6 & 7 \end{array}\right).$$

Condition number

In some systems, problems occur even without rounding. Consider the following augmented matrices:

$$\left(\begin{array}{cc|c} 1 & 1/2 & 3/2 \\ 1/2 & 1/3 & 1 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{cc|c} 1 & 1/2 & 3/2 \\ 1/2 & 1/3 & 5/6 \end{array}\right).$$

Here we have the same A, but two different input vectors:

$$\mathbf{b}_1 = (3/2, 1)' \quad \text{and} \quad \mathbf{b}_2 = (3/2, 5/6)'$$

which are pretty close to one another. You would expect then that the solutions $\mathbf{x}_1$ and $\mathbf{x}_2$ would also be close. Notice that this matrix does not need pivoting. Eliminating exactly we get

$$\left(\begin{array}{cc|c} 1 & 1/2 & 3/2 \\ 0 & 1/12 & 1/4 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{cc|c} 1 & 1/2 & 3/2 \\ 0 & 1/12 & 1/12 \end{array}\right).$$

Now solving we find

$$\mathbf{x}_1 = (0, 3)' \quad \text{and} \quad \mathbf{x}_2 = (1, 1)'$$

which are *not close at all* despite the fact that we did the calculations exactly. This poses a new problem: some matrices are very sensitive to small changes in input data. The extent of this sensitivity is measured

by the **condition number**. The definition of condition number is: consider all small changes δA and $\delta \mathbf{b}$ in A and $\mathbf{b}$ and the resulting change, $\delta \mathbf{x}$, in the solution $\mathbf{x}$. Then

$$\text{cond}(A) \equiv \max \left(\frac{\|\delta \mathbf{x}\|/\|\mathbf{x}\|}{\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta \mathbf{b}\|}{\|\mathbf{b}\|}} \right) = \max \left(\frac{\text{Relative error of output}}{\text{Relative error of inputs}} \right).$$

Put another way, changes in the input data get multiplied by the condition number to produce changes in the outputs. Thus a high condition number is bad. It implies that small errors in the input can cause large errors in the output. It is not obvious from our definition above, but one can prove that the condition number of a matrix is at least 1.

In MATLAB enter

```
≫ H = hilb(2)
```

which should result in the matrix above. MATLAB produces the condition number of a matrix with the command

```
≫ cond(H)
```

Thus for this matrix small errors in the input can get magnified by 19 in the output. Next try the matrix

```
≫ A = [ 1.2969 0.8648 ; .2161  .1441]
≫ cond(A)
```

For this matrix small errors in the input can get magnified by 2.5×10^8 in the output! (We will see this happen in the exercise.) This is obviously not very good for engineering where all measurements, constants and inputs are approximate.

Is there a solution to the problem of bad condition numbers? Usually, bad condition numbers in engineering contexts result from poor design. So, the engineering solution to bad conditioning is **redesign**.

Finally, find the determinant of the matrix A above:

```
≫ det(A)
```

which will be small. If $\det(A) = 0$ then the matrix is singular, which is bad because it implies there will not be a unique solution. The case here, $\det(A) \approx 0$, is also bad, because it means the matrix is almost singular. Although $\det(A) \approx 0$ generally indicates that the condition number will be large, they are actually independent things. To see this, find the determinant and condition number of the matrix `[1e-10,0;0,1e-10]` and the matrix `[1e+10,0;0,1e-10]`.

Exercises

11.1 Let

$$A = \begin{bmatrix} 1.2969 & .8648 \\ .2161 & .1441 \end{bmatrix}.$$

(a) Find the condition number, determinant and inverse of A (using MATLAB).

(b) Let B be the matrix obtained from A by rounding off to three decimal places ($1.2969 \mapsto 1.297$). Find the inverse of B. How do A^{-1} and B^{-1} differ? Explain how this happened.

(c) Set `b1 = [1.2969; 0.2161]` and do `x = A \ b1` . Repeat the process but with a vector `b2` obtained from `b1` by rounding off to three decimal places. Explain exactly what happened. Why was the first answer so simple? Why do the two answers differ by so much?

11.2 To see how to solve linear systems symbolically, try

```
≫ B = [sin(sym(1)) sin(sym(2)); sin(sym(3)) sin(sym(4))]
≫ c = [1; 2]
≫ x = B \ c
≫ pretty(x)
```

Now input the matrix

$$\mathsf{Cs} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

symbolically as above by wrapping each number in `sym`. Create a numerical version via `Cn = double(Cs)` and define the two vectors `d1 = [4; 8]` and `d2 = [1; 1]`. Solve the systems (as in the third line of code above) `Cs*x = d1`, `Cn*x = d1`, `Cs*x = d2`, and `Cn*x = d2`. Explain the results. Does the symbolic or non-symbolic way give more information?

Lecture 12

LU Decomposition

In many applications where linear systems appear, one needs to solve $A\mathbf{x} = \mathbf{b}$ for many different vectors $\mathbf{b}$. For instance, a structure must be tested under several different loads, not just one. As in the example of a truss (9.2), the loading in such a problem is usually represented by the vector $\mathbf{b}$. Gaussian elimination with pivoting is the most efficient and accurate way to solve a linear system. Most of the work in this method is spent on the matrix A itself. If we need to solve several different systems with the same A, and A is big, then we would like to avoid repeating the steps of Gaussian elimination on A for every different $\mathbf{b}$. This can be accomplished by the *LU decomposition*, which in effect records the steps of Gaussian elimination.

LU decomposition

The main idea of the LU decomposition is to record the steps used in Gaussian elimination on A in the places where the zero is produced. Consider the matrix

$$A = \begin{pmatrix} 1 & -2 & 3 \\ 2 & -5 & 12 \\ 0 & 2 & -10 \end{pmatrix}.$$

The first step of Gaussian elimination is to subtract 2 times the first row from the second row. In order to record what we have done, we will put the multiplier, 2, into the place it was used to make a zero, i.e. the second row, first column. In order to make it clear that it is a record of the step and not an element of A, we will put it in parentheses. This leads to

$$\begin{pmatrix} 1 & -2 & 3 \\ (2) & -1 & 6 \\ 0 & 2 & -10 \end{pmatrix}.$$

There is already a zero in the lower left corner, so we don't need to eliminate anything there. We record this fact with a (0). To eliminate the third row, second column, we need to subtract -2 times the second row from the third row. Recording the -2 in the spot it was used we have

$$\begin{pmatrix} 1 & -2 & 3 \\ (2) & -1 & 6 \\ (0) & (-2) & 2 \end{pmatrix}.$$

Let U be the upper triangular matrix produced, and let L be the lower triangular matrix with the records and ones on the diagonal, i.e.

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} 1 & -2 & 3 \\ 0 & -1 & 6 \\ 0 & 0 & 2 \end{pmatrix},$$

then we have the following wonderful property:

$$LU = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix} \begin{pmatrix} 1 & -2 & 3 \\ 0 & -1 & 6 \\ 0 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & -2 & 3 \\ 2 & -5 & 12 \\ 0 & 2 & -10 \end{pmatrix} = A.$$

Thus we see that A is actually the product of L and U. Here L is lower triangular and U is upper triangular. When a matrix can be written as a product of simpler matrices, we call that a *decomposition* of A and this one we call the LU decomposition.

Using LU to solve equations

If we also include pivoting, then an LU decomposition for A consists of three matrices P, L and U such that

$$PA = LU. \tag{12.1}$$

The pivot matrix P is the identity matrix, with the same rows switched as the rows of A are switched in the pivoting. For instance,

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

would be the pivot matrix if the second and third rows of A are switched by pivoting. MATLAB will produce an LU decomposition with pivoting for a matrix A with the command

```
> [L U P] = lu(A)
```

where `P` is the pivot matrix. To use this information to solve $A\mathbf{x} = \mathbf{b}$ we first pivot both sides by multiplying by the pivot matrix:

$$PA\mathbf{x} = P\mathbf{b} \equiv \mathbf{d}.$$

Substituting LU for PA we get

$$LU\mathbf{x} = \mathbf{d}.$$

Then we define $\mathbf{y} = U\mathbf{x}$, which is unknown since $\mathbf{x}$ is unknown. Using forward-substitution, we can (easily) solve

$$L\mathbf{y} = \mathbf{d}$$

for $\mathbf{y}$ and then using back-substitution we can (easily) solve

$$U\mathbf{x} = \mathbf{y}$$

for $\mathbf{x}$. In MATLAB this would work as follows:

```
≫ A = rand(5,5)
≫ [L U P] = lu(A)
≫ b = rand(5,1)
≫ d = P*b
≫ y = L\d
≫ x = U\y
≫ rnorm = norm(A*x - b)  % Check the result
```

We can then solve for any other **b** without redoing the LU step. Repeat the sequence for a new right hand side: `c = randn(5,1)`; you can start at the third line. While this may not seem like a big savings, it would be if A were a large matrix from an actual application.

The LU decomposition is an example of *Matrix Decomposition* which means taking a general matrix A and breaking it down into components with simpler properties. Here L and U are simpler because they are lower and upper triangular. There are many other matrix decompositions that are useful in various contexts. Some of the most useful of these are the QR decomposition, the Singular Value decomposition and Cholesky decomposition. Often a decomposition is associated with an algorithm, e.g., finding the LU decomposition is equivalent to completing Gaussian Elimination.

Exercises

12.1 Solve the systems below by hand using Gaussian elimination and back substitution (exactly as above) on the augmented matrix. As a by-product, give the LU decomposition of A. Pivot wherever appropriate (the number being eliminated should be smaller than the number eliminating it). Check by hand that $LU = PA$ and $A\mathbf{x} = \mathbf{b}$ and compare with MATLAB.

(a) $A = \begin{pmatrix} 0.5 & 4 \\ 2 & 4 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 0 \\ -3 \end{pmatrix}$

(b) $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 1 & 1 \\ 2 & 3 & 9 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$

12.2 Finish the following MATLAB function program:

```
function [x1, r1, x2, r2] = mysolve(A,b)
    % Solves linear systems using the LU decomposition with pivoting
    % and also with the built-in solve function A\b.
    % Inputs: A -- the matrix
    %         b -- the right-hand vector
    % Outputs: x1 -- the solution using the LU method
    %          r1 -- the scalar residual using the LU method
    %          x2 -- the solution using the built-in method
    %          r2 -- the scalar residual using the
    %                built-in method
```

Using `format long`, test the program on both random matrices (`randn(n,n)`) and Hilbert matrices (`hilb(n)`) with n large (as big as you can make it and the program still run). Print your program and summarize your observations. (Do not print any random matrices or vectors.)

Lecture 13

Nonlinear Systems - Newton's Method

An Example

The LORAN (LOng RAnge Navigation) system calculates the position of a boat at sea using signals from fixed transmitters. From the time differences of the incoming signals, the boat obtains differences of distances to the transmitters. This leads to two equations each representing hyperbolas defined by the differences of distance of two points (foci). An example of such equations[1] are

$$\begin{aligned} \frac{x^2}{186^2} - \frac{y^2}{300^2 - 186^2} &= 1 \quad \text{and} \\ \frac{(y-500)^2}{279^2} - \frac{(x-300)^2}{500^2 - 279^2} &= 1\,. \end{aligned} \tag{13.1}$$

Solving two quadratic equations with two unknowns, would require solving a 4 degree polynomial equation. We could do this by hand, but for a navigational system to work well, it must do the calculations automatically and numerically. We note that the Global Positioning System (GPS) works on similar principles and must do similar computations.

Vector Notation

In general, we can usually find solutions to a system of equations when the number of unknowns matches the number of equations. Thus, we wish to find solutions to systems that have the form

$$\begin{aligned} f_1(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ f_3(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, x_3, \ldots, x_n) &= 0. \end{aligned} \tag{13.2}$$

For convenience we can think of $(x_1, x_2, x_3, \ldots, x_n)$ as a vector $\mathbf{x}$ and $(f_1, f_2, \ldots, f_n)$ as a vector-valued function $\mathbf{f}$. With this notation, we can write the system of equations (13.2) simply as

$$\mathbf{f}(\mathbf{x}) = \mathbf{0},$$

i.e. we wish to find a vector that makes the vector function equal to the zero vector.

[1] E. Johnston, J. Mathews, *Calculus*, Addison-Wesley, 2001

As in Newton's method for one variable, we need to start with an initial guess $\mathbf{x}_0$. In theory, the more variables one has, the harder it is to find a good initial guess. In practice, this must be overcome by using physically reasonable assumptions about the possible values of a solution, i.e. take advantage of engineering knowledge of the problem. Once $\mathbf{x}_0$ is chosen, let

$$\Delta\mathbf{x} = \mathbf{x}_1 - \mathbf{x}_0.$$

Linear Approximation for Vector Functions

In the single variable case, Newton's method was derived by considering the linear approximation of the function f at the initial guess $\mathbf{x}_0$. From Calculus, the following is the linear approximation of $\mathbf{f}$ at $\mathbf{x}_0$, for vectors and vector-valued functions:

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + D\mathbf{f}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0).$$

Here $D\mathbf{f}(\mathbf{x}_0)$ is an $n \times n$ matrix whose entries are the various partial derivative of the components of $\mathbf{f}$, evaluated at $\mathbf{x}_0$. Specifically,

$$D\mathbf{f}(\mathbf{x}_0) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}_0) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}_0) & \frac{\partial f_1}{\partial x_3}(\mathbf{x}_0) & \dots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}_0) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}_0) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}_0) & \frac{\partial f_2}{\partial x_3}(\mathbf{x}_0) & \dots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}_0) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(\mathbf{x}_0) & \frac{\partial f_n}{\partial x_2}(\mathbf{x}_0) & \frac{\partial f_n}{\partial x_3}(\mathbf{x}_0) & \dots & \frac{\partial f_n}{\partial x_n}(\mathbf{x}_0) \end{pmatrix}. \tag{13.3}$$

Newton's Method

We wish to find $\mathbf{x}$ that makes $\mathbf{f}$ equal to the zero vectors, so let's choose $\mathbf{x}_1$ so that

$$\mathbf{f}(\mathbf{x}_0) + D\mathbf{f}(\mathbf{x}_0)(\mathbf{x}_1 - \mathbf{x}_0) = \mathbf{0}.$$

Since $D\mathbf{f}(\mathbf{x}_0)$ is a square matrix, we can solve this equation by

$$\mathbf{x}_1 = \mathbf{x}_0 - (D\mathbf{f}(\mathbf{x}_0))^{-1}\mathbf{f}(\mathbf{x}_0),$$

provided that the inverse exists. The formula is the vector equivalent of the Newton's method formula we learned before. However, in practice we never use the inverse of a matrix for computations, so we cannot use this formula directly. Rather, we can do the following. First solve the equation

$$D\mathbf{f}(\mathbf{x}_0)\Delta\mathbf{x} = -\mathbf{f}(\mathbf{x}_0). \tag{13.4}$$

Since $D\mathbf{f}(\mathbf{x}_0)$ is a known matrix and $-\mathbf{f}(\mathbf{x}_0)$ is a known vector, this equation is just a system of linear equations, which can be solved efficiently and accurately. Once we have the solution vector $\Delta\mathbf{x}$, we can obtain our improved estimate $\mathbf{x}_1$ by

$$\mathbf{x}_1 = \mathbf{x}_0 + \Delta\mathbf{x}.$$

For subsequent steps, we have the following process:

- Solve $D\mathbf{f}(\mathbf{x}_i)\Delta\mathbf{x} = -\mathbf{f}(\mathbf{x}_i)$ for $\Delta\mathbf{x}$.
- Let $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}$

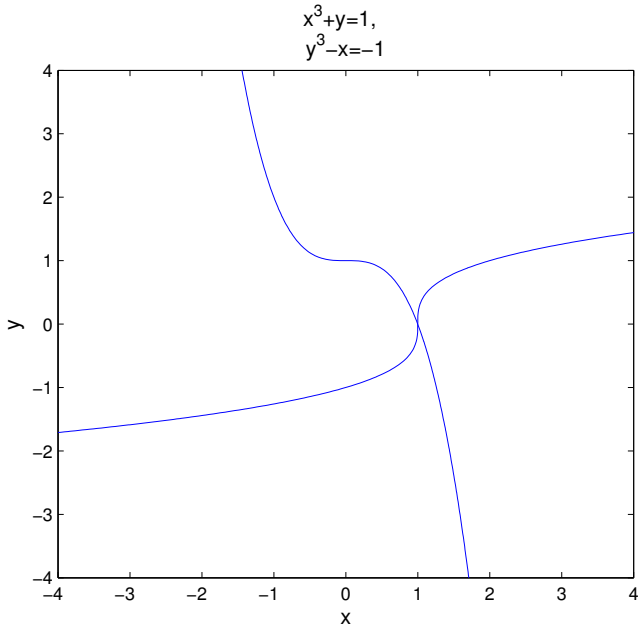

Figure 13.1: Graphs of the equations $x^3 + y = 1$ and $y^3 - x = -1$. There is one and only one intersection; at $(x, y) = (1, 0)$.

An Experiment

We will solve the following set of equations:

$$\begin{aligned} x^3 + y &= 1 \\ y^3 - x &= -1. \end{aligned} \tag{13.5}$$

You can easily check that $(x, y) = (1, 0)$ is a solution of this system. By graphing both of the equations you can also see that $(1, 0)$ is the only solution (Figure 13.1).

We can put these equations into vector-function form (13.2) by letting $x_1 = x$, $x_2 = y$ and

$$\begin{aligned} f_1(x_1, x_2) &= x_1^3 + x_2 - 1 \\ f_2(x_1, x_2) &= x_2^3 - x_1 + 1. \end{aligned}$$

or

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} x_1^3 + x_2 - 1 \\ x_2^3 - x_1 + 1 \end{pmatrix}.$$

Now that we have the equation in vector-function form, write the following script program:

```
format long;
f = @(x)[ x(1)^3+x(2)-1 ; x(2)^3-x(1)+1 ]
x = [.5;.5]
x = fsolve(f,x)
```

Save this program as `myfsolve.m` and run it. You will see that the internal MATLAB solving command `fsolve` approximates the solution, but only to about 7 decimal places. While that would be close enough for most applications, one would expect that we could do better on such a simple problem.

Next we will implement Newton's method for this problem. Modify your `myfsolve` program to:

```
% mymultnewton
format long;
n=8  % set some number of iterations, may need adjusting
f = @(x)[x(1)^3+x(2)-1 ; x(2)^3-x(1)+1] % the vector function
% the matrix of partial derivatives
Df = @(x)[3*x(1)^2,  1 ; -1,  3*x(2)^2]
x = [.5;.5]              % starting guess
for i = 1:n
      Dx = -Df(x)\f(x); % solve for increment
      x = x + Dx        % add on to get new guess
      f(x)              % see if f(x) is really zero
end
```

Save and run this program (as `mymultnewton`) and you will see that it finds the root exactly (to machine precision) in only 6 iterations. Why is this simple program able to do better than MATLAB's built-in program?

Exercises

13.1 (a) Put the LORAN equations (13.1) into the function form (13.2).

(b) Construct the matrix of partial derivatives $D\mathbf{f}$ in (13.3).

(c) Adapt the `mymultnewton` program to find a solution for these equations. By trying different starting vectors, find at least three different solutions. (There are actually four solutions.)

(d) Think of at least one way that the navigational system could determine the correct solution.

Lecture 14

Eigenvalues and Eigenvectors

Suppose that A is a square $(n \times n)$ matrix. We say that a nonzero vector $\mathbf{v}$ is an eigenvector and a number λ is its eigenvalue if

$$A\mathbf{v} = \lambda\mathbf{v}. \tag{14.1}$$

Geometrically this means that $A\mathbf{v}$ is in the same direction as $\mathbf{v}$, since multiplying a vector by a number changes its length, but not its direction.

MATLAB has a built-in routine for finding eigenvalues and eigenvectors:

```
≫ A = pascal(4)
≫ [v e] = eig(A)
```

The results are a matrix `v` that contains eigenvectors as columns and a diagonal matrix `e` that contains eigenvalues on the diagonal. We can check this by

```
≫ v1 = v(:,1)
≫ A*v1
≫ e(1,1)*v1
```

Finding Eigenvalues for 2×2 and 3×3

If A is 2×2 or 3×3 then we can find its eigenvalues and eigenvectors by hand. Notice that Equation (14.1) can be rewritten as

$$A\mathbf{v} - \lambda\mathbf{v} = \mathbf{0}.$$

It would be nice to factor out the $\mathbf{v}$ from the right-hand side of this equation, but we can't because A is a matrix and λ is a number. However, since $I\mathbf{v} = \mathbf{v}$, we can do the following:

$$\begin{aligned} A\mathbf{v} - \lambda\mathbf{v} &= A\mathbf{v} - \lambda I\mathbf{v} \\ &= (A - \lambda I)\mathbf{v} \\ &= \mathbf{0} \end{aligned}$$

If $\mathbf{v}$ is nonzero, then by Theorem 3 in Lecture 10 the matrix $(A - \lambda I)$ must be singular. By the same theorem, we must have

$$\det(A - \lambda I) = 0.$$

This is called the *characteristic equation*.

For a 2×2 matrix, $A - \lambda I$ is calculated as in the following example:

$$\begin{aligned} A - \lambda I &= \begin{pmatrix} 1 & 4 \\ 3 & 5 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 4 \\ 3 & 5 \end{pmatrix} - \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix} \\ &= \begin{pmatrix} 1-\lambda & 4 \\ 3 & 5-\lambda \end{pmatrix}. \end{aligned}$$

The determinant of $A - \lambda I$ is then

$$\begin{aligned} \det(A - \lambda I) &= (1-\lambda)(5-\lambda) - 4 \cdot 3 \\ &= -7 - 6\lambda + \lambda^2. \end{aligned}$$

The characteristic equation $\det(A - \lambda I) = 0$ is simply a quadratic equation:

$$\lambda^2 - 6\lambda - 7 = 0.$$

The roots of this equation are $\lambda_1 = 7$ and $\lambda_2 = -1$. These are the eigenvalues of the matrix A. Now to find the corresponding eigenvectors we return to the equation $(A - \lambda I)\mathbf{v} = \mathbf{0}$. For $\lambda_1 = 7$, the equation for the eigenvector $(A - \lambda I)\mathbf{v} = \mathbf{0}$ is equivalent to the augmented matrix

$$\left(\begin{array}{cc|c} -6 & 4 & 0 \\ 3 & -2 & 0 \end{array}\right). \tag{14.2}$$

Notice that the first and second rows of this matrix are multiples of one another. Thus Gaussian elimination would produce all zeros on the bottom row. Thus this equation has infinitely many solutions, i.e. infinitely many eigenvectors. Since only the direction of the eigenvector matters, this is okay, we only need to find one of the eigenvectors. Since the second row of the augmented matrix represents the equation

$$3x - 2y = 0,$$

we can let

$$\mathbf{v}_1 = \begin{pmatrix} 2 \\ 3 \end{pmatrix}.$$

This comes from noticing that $(x, y) = (2, 3)$ is a solution of $3x - 2y = 0$.

For $\lambda_2 = -1$, $(A - \lambda I)\mathbf{v} = \mathbf{0}$ is equivalent to the augmented matrix

$$\left(\begin{array}{cc|c} 2 & 4 & 0 \\ 3 & 6 & 0 \end{array}\right).$$

Once again the first and second rows of this matrix are multiples of one another. For simplicity we can let

$$\mathbf{v}_2 = \begin{pmatrix} -2 \\ 1 \end{pmatrix}.$$

One can always check an eigenvector and eigenvalue by multiplying:

$$\begin{aligned} A\mathbf{v}_1 &= \begin{pmatrix} 1 & 4 \\ 3 & 5 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 14 \\ 21 \end{pmatrix} = 7 \begin{pmatrix} 2 \\ 3 \end{pmatrix} = 7\mathbf{v}_1 \quad \text{and} \\ A\mathbf{v}_2 &= \begin{pmatrix} 1 & 4 \\ 3 & 5 \end{pmatrix} \begin{pmatrix} -2 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \end{pmatrix} = -1 \begin{pmatrix} -2 \\ 1 \end{pmatrix} = -1\mathbf{v}_2 . \end{aligned}$$

For a 3×3 matrix we could complete the same process. The $\det(A - \lambda I) = 0$ would be a cubic polynomial and we would expect to usually get 3 roots, which are the eigenvalues.

Larger Matrices

For a $n \times n$ matrix with $n \geq 4$ this process is too long and cumbersome to complete by hand. Further, this process is not well suited even to implementation on a computer program since it involves determinants and solving a n-degree polynomial. For $n \geq 4$ we need more ingenious methods. These methods rely on the geometric meaning of eigenvectors and eigenvalues rather than solving algebraic equations. We will overview these methods in Lecture 16.

Complex Eigenvalues

It turns out that the eigenvalues of some matrices are complex numbers, even when the matrix only contains real numbers. When this happens the complex eigenvalues must occur in conjugate pairs, i.e.

$$\lambda_{1,2} = \alpha \pm i\beta.$$

The corresponding eigenvectors must also come in conjugate pairs:

$$\mathbf{w} = \mathbf{u} \pm i\mathbf{v}.$$

In applications, the imaginary part of the eigenvalue, β, often is related to the frequency of an oscillation. This is because of Euler's formula

$$e^{\alpha+i\beta} = e^{\alpha}(\cos\beta + i\sin\beta).$$

Certain kinds of matrices that arise in applications can only have real eigenvalues and eigenvectors. The most common such type of matrix is the symmetric matrix. A matrix is symmetric if it is equal to its own transpose, i.e. it is symmetric across the diagonal. For example,

$$\begin{pmatrix} 1 & 3 \\ 3 & -5 \end{pmatrix}$$

is symmetric and so we know beforehand that its eigenvalues will be real, not complex.

Exercises

14.1 Find the eigenvalues and eigenvectors of the following matrix by hand:

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

14.2 Find the eigenvalues and eigenvectors of the following matrix by hand:

$$B = \begin{pmatrix} 1 & -3 \\ 3 & 1 \end{pmatrix}.$$

Can you guess the eigenvalues of the matrix

$$C = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}?$$

Lecture 15

An Application of Eigenvectors: Vibrational Modes and Frequencies

One application of eigenvalues and eigenvectors is in the analysis of vibration problems. A simple nontrivial vibration problem is the motion of two objects with equal masses m attached to each other and fixed outer walls by equal springs with spring constants k, as shown in Figure 15.1.

Let x_1 denote the displacement of the first mass and x_2 the displacement of the second, and note the displacement of the walls is zero. Each mass experiences forces from the adjacent springs proportional to the stretch or compression of the spring. Ignoring any friction, Newton's law of motion $ma = F$, leads to

$$\begin{array}{lllllll} m\ddot{x}_1 & = -k(x_1 - 0) & +k(x_2 - x_1) & = & -2kx_1 & +kx_2 & \text{and} \\ m\ddot{x}_2 & = & -k(x_2 - x_1) & +k(0 - x_2) & = & kx_1 & -2kx_2 \quad . \end{array} \tag{15.1}$$

Dividing both sides by m we can write these equations in matrix form

$$\ddot{\mathbf{x}} = -A\mathbf{x}, \tag{15.2}$$

where

$$A = \begin{pmatrix} 2\frac{k}{m} & -1\frac{k}{m} \\ -1\frac{k}{m} & 2\frac{k}{m} \end{pmatrix}. \tag{15.3}$$

For this type of equation, the general solution is

$$\mathbf{x}(t) = c_1\mathbf{v}_1 \sin\left(\sqrt{\lambda_1}\, t + \phi_1\right) + c_2\mathbf{v}_2 \sin\left(\sqrt{\lambda_2}\, t + \phi_2\right) \tag{15.4}$$

where λ_1 and λ_2 are eigenvalues of A with corresponding eigenvectors $\mathbf{v}_1$ and $\mathbf{v}_2$. One can check that this is a solution by substituting it into the equation (15.2).

The eigenvalues of A are the squares of the frequencies of oscillation. Let's set $m = 1$ and $k = 1$ in A. We can find the eigenvalues and eigenvectors of A using Matlab:

```
≫ A = [2 -1 ; -1 2]
≫ [v e] = eig(A)
```

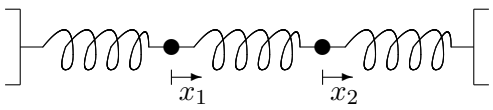

Figure 15.1: Two equal masses attached to each other and fixed walls by equal springs.

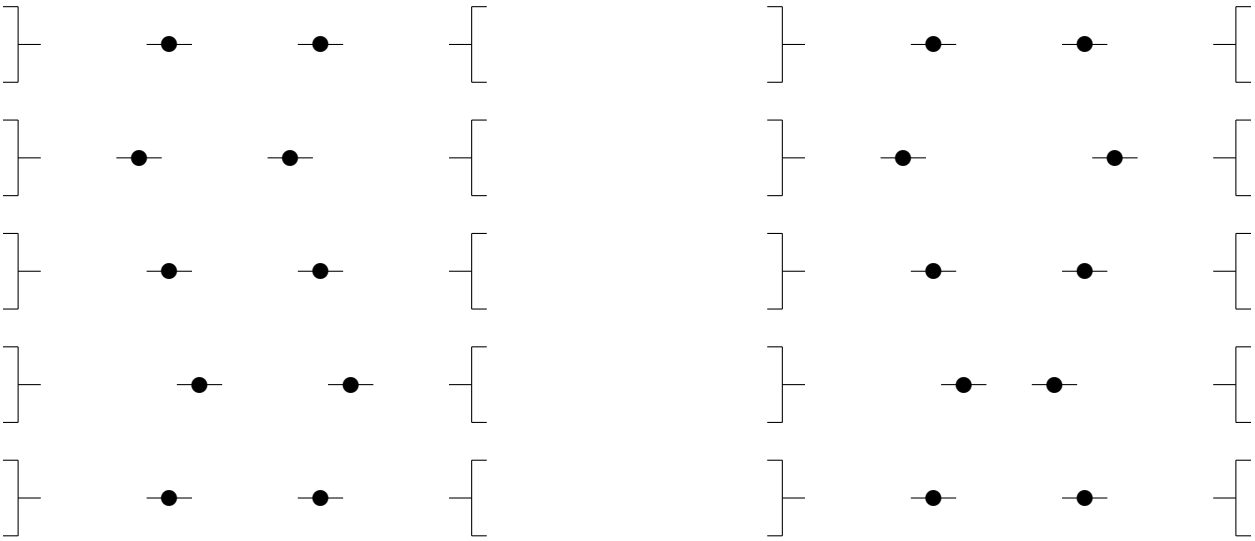

Figure 15.2: Two vibrational modes of a simple oscillating system. In the left mode the weights move together and in the right mode they move opposite. Note that the two modes actually move at different speeds.

This should produce a matrix v whose columns are the eigenvectors of A and a diagonal matrix e whose entries are the eigenvalues of A. In the first eigenvector, $\mathbf{v}_1$, the two entries are equal. This represents the mode of oscillation where the two masses move in sync with each other. The second eigenvector, $\mathbf{v}_2$, has the same entries but opposite signs. This represents the mode where the two masses oscillate in anti-synchronization. Notice that the frequency for anti-sync motion is $\sqrt{3}$ times that of synchronous motion.

Which of the two modes is the most dangerous for a structure or machine? It is the one with the *lowest frequency* because that mode can have the largest displacement. Sometimes this mode is called the *fundamental mode*.

We can do the same for three equal masses. With $m = 1$, $k = 1$ the corresponding matrix A would be

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

Find the eigenvectors and eigenvalues as above. There are three different modes. Interpret them from the eigenvectors.

Exercises

15.1 Find the modes and their frequencies for 4 equal masses with $m = 3$ kg and equal springs with $k = 10$ N/m. Describe the modes (a sketch will suffice).

15.2 Find the modes and their frequencies for three unequal masses $m_1 = 2$ kg, $m_2 = 3$ kg and $m_3 = 4$ kg connected by 4 equal springs with $k = 5$ N/m. How do unequal masses affect the modes? (You must start with the equations of motion to do this correctly.)

Lecture 16

Numerical Methods for Eigenvalues

As mentioned above, the eigenvalues and eigenvectors of an $n \times n$ matrix where $n \geq 4$ must be found numerically instead of by hand. The numerical methods that are used in practice depend on the geometric meaning of eigenvalues and eigenvectors which is equation (14.1). The essence of all these methods is captured in the Power method, which we now introduce.

The Power Method

In the command window of MATLAB enter

```
≫ A = hilb(5)
≫ x = ones(5,1)
≫ x = A*x
≫ el = max(x)
≫ x = x/el
```

Compare the new value of `x` with the original. Repeat the last three lines (you can use the scroll up button). Compare the newest value of `x` with the previous one and the original. Notice that there is less change between the second two. Repeat the last three commands over and over until the values stop changing. You have completed what is known as the *Power Method*. Now try the command

```
≫ [v e] = eig(A)
```

The last entry in `e` should be the final `el` we computed. The last column in `v` is `x/norm(x)`. Below we explain why our commands gave this eigenvalue and eigenvector.

For illustration consider a 2×2 matrix whose eigenvalues are $1/3$ and -2 and whose corresponding eigenvectors are $\mathbf{v}_1$ and $\mathbf{v}_2$. Let $\mathbf{x}_0$ be any vector which is a combination of $\mathbf{v}_1$ and $\mathbf{v}_2$, e.g.,

$$\mathbf{x}_0 = \mathbf{v}_1 + \mathbf{v}_2.$$

Now let $\mathbf{x}_1$ be A times $\mathbf{x}_0$. It follows from (14.1) that

$$\begin{aligned} \mathbf{x}_1 &= A\mathbf{v}_1 + A\mathbf{v}_2 \\ &= \frac{1}{3}\mathbf{v}_1 + -2\mathbf{v}_2. \end{aligned} \tag{16.1}$$

Thus the $\mathbf{v}_1$ part is shrunk while the $\mathbf{v}_2$ is stretched. If we repeat this process k times then

$$\begin{aligned} \mathbf{x}_k &= A\mathbf{x}_{k-1} \\ &= A^k\mathbf{x}_0 \\ &= \left(\frac{1}{3}\right)^k \mathbf{v}_1 + (-2)^k\mathbf{v}_2. \end{aligned} \tag{16.2}$$

Clearly, $\mathbf{x}_k$ grows in the direction of $\mathbf{v}_2$ and shrinks in the direction of $\mathbf{v}_1$. This is the principle of the Power Method, vectors multiplied by A are stretched most in the direction of the eigenvector whose eigenvalue has the largest absolute value.

The eigenvalue with the largest absolute value is called the *dominant* eigenvalue. In many applications this quantity will necessarily be positive for physical reasons. When this is the case, the MATLAB code above will work since `max(v)` will be the element with the largest absolute value. In applications where the dominant eigenvalue may be negative, the program must use flow control to determine the correct number.

Summarizing the Power Method:

- Repeatedly multiply $\mathbf{x}$ by A and divide by the element with the largest absolute value.
- The element of largest absolute value converges to largest absolute eigenvalue.
- The vector converges to the corresponding eigenvector.

Note that this logic only works when the eigenvalue largest in magnitude is real. If the matrix and starting vector are real then the power method can never give a result with an imaginary part. Eigenvalues with imaginary part mean the matrix has a rotational component, so the eigenvector would not settle down either.

Try

```
≫ A = randn(15,15);
≫ e = eig(A)
```

You can see that for a random square matrix, many of the eigenvalues are complex.

However, matrices in applications are not just random. They have structure, and this can lead to real eigenvalues as seen in the next section.

Symmetric, Positive-Definite Matrices

As noted in the previous paragraph, the power method can fail if A has complex eigenvalues. One class of matrices that appear often in applications and for which the eigenvalues are always real are called the symmetric matrices. A matrix is *symmetric* if

$$A' = A,$$

i.e. A is symmetric with respect to reflections about its diagonal.

Try

```
>> A = rand(5,5)
>> C = A'*A
>> e = eig(C)
```

You can see that the eigenvalues of these symmetric matrices are real.

Next we consider an even more specialized class for which the eigenvalues are not only real, but positive. A symmetric matrix is called *positive definite* if for all vectors $\mathbf{v} \neq \mathbf{0}$ the following holds:

$$A\mathbf{v} \cdot \mathbf{v} > 0.$$

Geometrically, A does not rotate any vector by more than $\pi/2$. In summary:

- If A is symmetric then its eigenvalues are real.
- If A is symmetric positive definite, then its eigenvalues are positive numbers.

Notice that the B matrices in the previous section were symmetric and the eigenvalues were all real. Notice that the Hilbert and Pascal matrices are symmetric.

The residual of an approximate eigenvector-eigenvalue pair

If $\mathbf{v}$ and λ are an eigenvector-eigenvalue pair for A, then they are supposed to satisfy the equations: $A\mathbf{v} = \lambda\mathbf{v}$. Thus a scalar residual for approximate $\mathbf{v}$ and λ would be

$$r = \|A\mathbf{v} - \lambda\mathbf{v}\|.$$

The Inverse Power Method

In the application of vibration analysis, the mode (eigenvector) with the lowest frequency (eigenvalue) is the most dangerous for the machine or structure. The Power Method gives us instead the largest eigenvalue, which is the least important frequency. In this section we introduce a method, the *Inverse Power Method* which produces exactly what is needed.

The following facts are at the heart of the Inverse Power Method:

- If λ is an eigenvalue of A then $1/\lambda$ is an eigenvalue for A^{-1}.
- The eigenvectors for A and A^{-1} are the same.

Thus if we apply the Power Method to A^{-1} we will obtain the largest absolute eigenvalue of A^{-1}, which is exactly the reciprocal of the smallest absolute eigenvalue of A. We will also obtain the corresponding eigenvector, which is an eigenvector for both A^{-1} and A. Recall that in the application of vibration mode analysis, the smallest eigenvalue and its eigenvector correspond exactly to the frequency and mode that we are most interested in, i.e. the one that can do the most damage.

Here as always, we do not really want to calculate the inverse of A directly if we can help it. Fortunately, multiplying $\mathbf{x}_i$ by A^{-1} to get $\mathbf{x}_{i+1}$ is equivalent to solving the system $A\mathbf{x}_{i+1} = \mathbf{x}_i$, which can be done

efficiently and accurately. Since iterating this process involves solving a linear system with the same A but many different right hand sides, it is a perfect time to use the LU decomposition to save computations. The following function program does n steps of the Inverse Power Method.

```
function [x e] = myipm(A,n)
    % Performs the inverse power method.
    % Inputs: A -- a square matrix,   n --  number of iterations.
    % Outputs: x -- estimated eigenvector,  e -- estimated smallest eigenvalue.
    [L U P] = lu(A);  % LU decomposition of A with pivoting
    m = size(A,1); % determine the size of A
    x = ones(m,1); % make an initial vector with ones
    for i = 1:n
        px = P*x;  % Apply pivot
        y = L\px;  % solve via LU
        x = U\y;
        % find the maximum entry in absolute value, retaining its sign
        M = max(x);
        m = min(x);
        if abs(M) >= abs(m)
           el = M;
        else
           el = m;
        end
        x = x/el; % divide by the estimated eigenvalue of the inverse of A
    end
    e = 1/el; % reciprocate to get an eigenvalue of A
end
```

Create a 5×5 Hilbert matrix H and use the program to find its smallest eigenvalue and corresponding. Also do `[V, E] = eig(H)` and compare.

Exercises

16.1 For each of the matrices

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -3 \end{bmatrix},$$

perform two iterations of the power method by hand starting with a vector of all ones. State the resulting approximations of the eigenvalue and eigenvector.

16.2 (a) Write a well-commented MATLAB **function** program `mypm.m` that inputs a matrix and a tolerance, applies the power method until the scalar residual is less than the tolerance, and outputs the estimated eigenvalue and eigenvector, the number of steps, and the scalar residual.

(b) Test your program on the matrices A and B in the previous exercise and check that you get the same results as your hand calculations for the first 2 iterations. Turn in your program only.

Lecture 17

The QR Method*

The Power Method and Inverse Power Method each give us only one eigenvalue-eigenvector pair. While both of these methods can be modified to give more eigenvalues and eigenvectors, there is a better method for obtaining all the eigenvalues called the *QR method*. This is the basis of all modern eigenvalue software, including MATLAB, so we summarize it briefly here.

The QR method uses the fact that any square matrix has a *QR decomposition*. That is, for any A there are matrices Q and R such the $A = QR$ where Q has the property

$$Q^{-1} = Q'$$

and R is upper triangular. A matrix Q with the property that its transpose equals its inverse is called an *orthogonal* matrix, because its column vectors are mutually orthogonal.

The QR method consists of iterating following steps:

- Transform A into a tridiagonal matrix H.
- Decompose H in QR.
- Multiply Q and R together in reverse order to form a new H.

The diagonal of H will converge to the eigenvalues.

The details of what makes this method converge are beyond the scope of the this book. However, we note the following theory behind it for those with more familiarity with linear algebra. First the Hessian matrix H is obtained from A by a series of similarity transformation, thus it has the same eigenvalues as A. Secondly, if we denote by H_0, H_1, H_2, ..., the sequence of matrices produced by the iteration, then

$$H_{i+1} = R_i Q_i = Q_i^{-1} Q_i R_i Q_i = Q_i' H_i Q_i.$$

Thus each H_{i+1} is a related to H_i by an (orthogonal) similarity transformation and so they have the same eigenvalues as A.

There is a built-in QR decomposition in MATLAB which is called with the command: `[Q R] = qr(A)`. Thus the following program implements QR method until it converges:

```
function [E,steps] = myqrmethod(A)
    % Computes all the eigenvalues of a matrix using the QR method.
    % Input: A -- square matrix
    % Outputs: E -- vector of eigenvalues
    %          steps -- the number of iterations it took
    [m n] = size(A);
    if m ~= n
       warning('The input matrix is not square.')
       return
    end
    % Set up initial estimate
    H = hess(A);
    E = diag(H);
    change = 1;
    steps = 0;
    % loop while estimate changes
    while change > 0
        Eold = E;
        % apply QR method
        [Q R] = qr(H);
        H = R*Q;
        E = diag(H);
        % test change
        change = norm(E - Eold);
        steps = steps +1;
    end
end
```

As you can see the main steps of the program are very simple. The really hard calculations are contained in the built-in commands `hess(A)` and `qr(H)`.

Run this program and compare the results with MATLAB's built in command:

```
>> format long
>> format compact
>> A = hilb(5)
>> [Eqr,steps] = myqrmethod(A)
>> Eml = eig(A)
>> diff = norm(Eml - flipud(Eqr))
```

Exercises

17.1 Modify `myqrmethod` to stop after 1000 iterations. Use the modified program on the matrix `A = hilb(n)` with `n` equal to 10, 50, and 200. Use the norm to compare the results to the eigenvalues obtained from MATLAB's built-in program `eig`. Turn in your program and a brief report on the experiment.

Lecture 18

Iterative solution of linear systems*

Newton refinement

Conjugate gradient method

Review of Part II

Methods and Formulas

Basic Matrix Theory:

Identity matrix: $AI = A$, $IA = A$, and $I\mathbf{v} = \mathbf{v}$
Inverse matrix: $AA^{-1} = I$ and $A^{-1}A = I$
Norm of a matrix: $\|A\| \equiv \max_{\|\mathbf{v}\|=1} \|A\mathbf{v}\|$
A matrix may be singular or nonsingular. See Lecture 10.

Solving Process:

Learn the exact Gaussian Elimination algorithm:
Row $j \mapsto$ Row j - (ratio) Row i
Gaussian Elimination this way produces LU decomposition
Row Pivoting (bigger absolute number on top)
Back Substitution

Condition number:

$$\text{cond}(A) \equiv \max \left(\frac{\|\delta \mathbf{x}\|/\|\mathbf{x}\|}{\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta \mathbf{b}\|}{\|\mathbf{b}\|}} \right) = \max \left(\frac{\text{Relative error of output}}{\text{Relative error of inputs}} \right).$$

A big condition number is bad; in engineering it usually results from poor design.

LU factorization:

The LU factorization is a by-product of Gaussian Elimination (if done with the correct algorithm).

$$PA = LU.$$

Solving steps:
Multiply by P: $\mathbf{d} = P\mathbf{b}$
Forwardsolve: $L\mathbf{y} = \mathbf{d}$
Backsolve: $U\mathbf{x} = \mathbf{y}$

Eigenvalues and eigenvectors:

A nonzero vector $\mathbf{v}$ is an eigenvector and a number λ is its eigenvalue if

$$A\mathbf{v} = \lambda\mathbf{v}.$$

Characteristic equation: $\det(A - \lambda I) = 0$
Equation of the eigenvector: $(A - \lambda I)\mathbf{v} = \mathbf{0}$
Residual for an approximate eigenvector-eigenvalue pair: $r = \|A\mathbf{v} - \lambda\mathbf{v}\|$

Complex eigenvalues:

Occur in conjugate pairs: $\lambda_{1,2} = \alpha \pm i\beta$
and eigenvectors must also come in conjugate pairs: $\mathbf{w} = \mathbf{u} \pm i\mathbf{v}$.

Vibrational modes:

Eigenvalues are frequencies squared. Eigenvectors represent modes.

Power Method:

- Repeatedly multiply $\mathbf{x}$ by A and divide by the element with the largest absolute value.
- The element of largest absolute value converges to largest absolute eigenvalue.
- The vector converges to the corresponding eigenvector.
- Convergence assured for a real symmetric matrix, but not for an arbitrary matrix, which may not have real eigenvalues at all.

Inverse Power Method:

- Apply power method to A^{-1}.
- Use solving rather than the inverse.
- If λ is an eigenvalue of A then $1/\lambda$ is an eigenvalue for A^{-1}.
- The eigenvectors for A and A^{-1} are the same.

Symmetric and Positive definite:

- Symmetric: $A = A'$.
- If A is symmetric its eigenvalues are real.
- Positive definite: $A\mathbf{x} \cdot \mathbf{x} > 0$.
- If A is positive definite, then its eigenvalues are positive.

QR method (Not covered in MATH 3600 at Ohio):

- Transform A into H the Hessian form of A.
- Decompose H in QR.
- Multiply Q and R together in reverse order to form a new H.
- Repeat
- The diagonal of H will converge to the eigenvalues of A.

Matlab

Matrix arithmetic:

`A = [ 1  3 -2 5 ;  -1  -1 5 4 ; 0 1 -9  0]` ..... Manually enter a matrix.
`u = [ 1  2  3  4]'`
`A*u`
`B = [3 2 1; 7 6 5; 4 3 2]`
`B*A` ..... multiply B times A.
`2*A` ..... multiply a matrix by a scalar.
`A + A` ..... add matrices.
`A + 3` ..... add 3 to every entry of a matrix.
`B.*B` ..... component-wise multiplication.
`B.^3` ..... component-wise exponentiation.

Special matrices:

`I = eye(3)` ..... identity matrix
`D = ones(5,5)`
`O = zeros(10,10)`
`C = rand(5,5)` ..... random matrix with uniform distribution in $[0, 1]$.
`C = randn(5,5)` ..... random matrix with normal distribution.
`hilb(6)`
`pascal(5)`

General matrix commands:

`size(C)` ..... gives the dimensions $(m \times n)$ of A.
`norm(C)` ..... gives the norm of the matrix.
`det(C)` ..... the determinant of the matrix.
`max(C)` ..... the maximum of each row.
`min(C)` ..... the minimum in each row.
`sum(C)` ..... sums each row.
`mean(C)` ..... the average of each row.
`diag(C)` ..... just the diagonal elements.
`inv(C)` ..... inverse of the matrix.
`C'` ..... transpose of the matrix.

Matrix decompositions:

`[L U P] = lu(C)`
`[Q R] = qr(C)`
`H = hess(C)` ..... transform into a Hessian tri-diagonal matrix, which has the same eigenvalues as A.