Algorithms with Matlab

Eleftherios Gkioulekas
Mathematical Sciences Computing Center
University of Washington

December, 1996

1 Introduction

Numerical analysis is the branch of mathematics whose goal is to figure out how computers can solve problems in a way that's fast, efficient and accurate. *Linear algebra* is a large part of numerical analysis, because many problems eventually reduce to one of the following linear algebra problems:
• We want to solve a system of linear equations.
• We want to solve an eigenvalue problem.
For this reason, researchers think of these problems as "elementary" the way you would think of addition and multiplication elementary enough to use a calculator. Likewise, researchers will use a "calculator" in some sense of the word. Matlab can serve as such a calculator. Beyond Matlab you have the option of using software libraries. Such libraries exist for most commonly used languages like `C++` (e.g. `LAPACK++`) and `FORTRAN` (e.g. LINPACK, EISPACK). For more information about the latter visit the Netlib repository at `http://www.netlib.org/` or send email to `netlib@ornl.gov` with one line that says `send index`.

In this tutorial we are assuming that you have read and understood the *"Programming with Matlab"* tutorial. The purpose of this tutorial is to review the mathematical concepts of linear algebra to give you a feel for the "big picture" and at the same time show you how you can experiment with these concepts using Matlab. As you will see, these problems are not quite "elementary"

2 Basic concepts

A *complex matrix* $\mathbf{A}$ of size $n \times m$ is a mapping

$$\mathbf{A} : \{1, \ldots, n\} \times \{1, \ldots, m\} \mapsto \mathcal{C}$$

where $\mathcal{C}$ is the set of complex numbers. In other words, a matrix will take two integers i and j such that $1 \leq i \leq n$ and $1 \leq j \leq m$ and return back a complex number. That number we denote with a_{ij} (we use the lowercase letter). The set of all $n \times m$ complex matrices is denoted as $\mathcal{C}^{n \times m}$. If we restrict ourselves to real values only, then we are dealing with *real matrices* and the set of these is

denoted as $\mathcal{R}^{n\times m}$. Finally, for the sake of notation, we will denote the set of all integers from 1 to n with the symbol:

$$[n] = \{1, \ldots, n\}$$

In Matlab all data structures are matrices. In particular ordinary numbers are 1×1 matrices, vectors (or "arrays" in general) are $1 \times n$ matrices. The *Programming with Matlab* tutorial has covered the basics of how all these data structures are setup in Matlab.

There exist various operations defined for matrices:

Definition 1 *(a) Let $\mathbf{A}, \mathbf{B} \in \mathcal{C}^{n\times m}$ be two matrices. The sum and difference of these matrices is:*

$$\begin{aligned} \mathbf{C} = \mathbf{A} + \mathbf{B} &\iff c_{ij} = a_{ij} + b_{ij},\ \forall (i,j) \in [n] \times [m] \\ \mathbf{C} = \mathbf{A} - \mathbf{B} &\iff c_{ij} = a_{ij} - b_{ij},\ \forall (i,j) \in [n] \times [m] \end{aligned}$$

(b) Let $\mathbf{A} \in \mathcal{C}^{n\times m}$ and $\mathbf{B} \in \mathcal{C}^{m\times p}$. The product of these matrices is

$$\mathbf{C} = \mathbf{AB} \iff c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj},\ \forall (i,j) \in [n] \times [p]$$

(c) The transpose of a matrix $\mathbf{A} \in \mathcal{C}^{n\times m}$ is:

$$\mathbf{B} = \mathbf{A}^T \iff b_{ij} = a_{ji},\ \forall (i,j) \in [n] \times [m]$$

(d) The hermitian or complex conjugate transpose is:

$$\mathbf{B} = \mathbf{A}^H \iff b_{ij} = a^*_{ji},\ \forall (i,j) \in [n] \times [m]$$

Moreover, there are the following important classes of matrices that we will see mentioned later on:

Definition 2 *A matrix $\mathbf{A} \in \mathcal{C}^{n\times n}$ is:*
(a) symmetric $\iff \mathbf{A} = \mathbf{A}^T$
(b) hermitian $\iff \mathbf{A} = \mathbf{A}^H$
(c) orthogonal $\iff \mathbf{A}\mathbf{A}^T = \mathbf{A}^T\mathbf{A} = \mathbf{I}$
(d) unitary $\iff \mathbf{A}\mathbf{A}^H = \mathbf{A}^H\mathbf{A} = \mathbf{I}$
(e) normal $\iff \mathbf{A}\mathbf{A}^H = \mathbf{A}\mathbf{A}^H$

For real matrices, hermitian means the same as symmetric, unitary means the same as orthogonal, and *both* of these distinct classes are normal.

In Matlab matrices that are stored in `A` and `B` can be added or multiplied quite simply by saying:

```
C = A + B;
C = A - B;
C = A * B;
```

The *transpose* and the *hermitian* of a matrix can be obtained by

```
B = A.';      % transpose
B = A';       % hermitian
```

The usual rules apply to these operations except for the following:

- Multiplication is not commutative for all matrices. That is there exist matrices such that

$$\mathbf{AB} \neq \mathbf{BA}$$

- Square matrices don't always have a *multiplicative inverse*. A multiplicative inverse is defined by:

$$\mathbf{B} = \mathbf{A}^{-1} \iff \mathbf{AB} = \mathbf{BA} = \mathbf{I}$$

The first exception is not very interesting, but the second one is related to one of the two "Big Questions" in linear algebra: Solving linear systems of equations.

3 Matrix inversion is not easy

Suppose that $\mathbf{A} \in \mathcal{C}^{n \times n}$ is a square matrix and $\mathbf{x}, \mathbf{b} \in \mathcal{C}^{n \times 1}$ are vectors (actually, column matrices). If $\mathbf{A}$ and $\mathbf{b}$ are known, then we want to find an $\mathbf{x}$ such that

$$\mathbf{Ax} = \mathbf{b}$$

If $\mathbf{A}$ has an inverse then $\mathbf{x}$ is unique and given by:

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

otherwise $\mathbf{x}$ will either not exist, or will not be unique.

In order for a matrix to have an inverse, the matrix has to be square. Given that, the existence of matrix inverses is determined by a quantity called *determinant*. The determinant is defined in terms of *permutations* so we must explain what these are first. A permutation $\sigma \in S_n$ of order n is a bijection that maps:

$$\sigma : [n] \mapsto [n]$$

The set of permutations of order n is written as S_n and it has $n!$ elements. In simple terms, take an ordered set of integers say $(1, 2, 3, 4, 5)$ and reorder it to $(4, 1, 3, 2, 5)$. Our concept of how we reordered an ordered set of things is what is being modeled by permutations. There are as many permutations as there are ways in which we can reorder things. One way to represent permutations is by showing reorderings of $(1, 2, \ldots, n)$. For example the permutations of order 3 are:

$$S_3 = \{(1, 2, 3), (2, 3, 1), (3, 1, 2), (1, 3, 2), (3, 2, 1), (2, 1, 3)\}$$

Permutations have an important property called *parity*. If $\sigma \in S_n$ is a permutation then we define the parity to be equal to:

$$s(\sigma) = \text{sign} \prod_{j=1}^{n-1} \prod_{i=j+1}^{n} (\sigma(i) - \sigma(j))$$

When the parity is +1 we talk of *even* permutations. When it is -1 we talk of *odd* permutations. You can verify that the even permutations in S_3 are

$$\{(1,2,3),(2,3,1),(3,1,2)\}$$

and the odd ones are

$$\{(1,3,2),(3,2,1),(2,1,3)\}.$$

Given these definitions, the determinant of a matrix $\mathbf{A} \in \mathcal{C}^{n \times n}$ is defined by:

$$\det(\mathbf{A}) = \sum_{\sigma \in S_n} s(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

Unfortunately, this is not the most useful definition in terms of numerics. There are $n!$ terms to add, and every one of these terms involves n multiplications. This will overwhelm any computer soon enough, not to mention round-off errors. You have probably seen many recursive definitions of the determinant in other courses. Those are not helpful either because they also involve work that increases by $n!$.

Most computations of determinants are based on the following theorem:

Theorem 1 *Let $\mathbf{A}, \mathbf{B} \in \mathcal{C}^{n \times n}$ be two square matrices. Then,*

$$\det(\mathbf{AB}) = \det(\mathbf{A}) \det(\mathbf{B})$$

We write our matrix as a product of matrices whose determinants are easy to compute and apply the theorem. More on this later.

Now the reason why determinants are important is because of this theorem:

Theorem 2 *Let $\mathbf{A} \in \mathcal{C}^{n \times n}$ be a square matrix.*

$$\mathbf{A}^{-1}\text{exists} \Longleftrightarrow \det(\mathbf{A}) \neq 0$$

Notice by the definition that the determinant varies continuously as you vary one of the elements of the matrix. This means that determinants are quite reasonable functions of n^2 variables which has the following implication: Most of the time, the determinant will be quite nonzero and an inverse will exist. Sometimes, the determinant will be very close to zero and an inverse will exist but will be so sensitive to the original matrix that it will be hard to compute. In this case we say we have an *ill-conditioned* matrix. Finally, on very rare matrices the determinant will be *exactly* zero and then the inverse just doesn't exist. In this case we have a *singular* matrix. These last two cases cause trouble from a numerical point of view. To make matters worse, deciding numerically which case is which is not trivial either.

In Matlab computing determinants and inverses may appear innocuously simple. If `A` is your matrix then

```
x = det(A);
```

will compute the determinant and

```
B = inv(A);
```

will return the inverse. However, for ill-conditioned matrices, `inv` will not give you the correct inverse. *Sometimes* Matlab will detect this, but not always.

There exists a class of square matrices $\mathbf{H}^{(n)}$ called the *Hilbert matrices* that are defined by:

$$h_{ij} = \frac{1}{i+j-1}$$

The matlab command `hilb` will return a Hilbert matrix of any size. That is:

```
A = hilb(10);
```

will return a Hilbert matrix of size 10. These matrices have two properties: Inverting them is very sensitive to floating point error but there is an analytic equation that gives their exact inverse! The inverse is given by

$$\tilde{h}_{ij} = \tilde{h}_{ji} = \frac{r_{ij}}{i+j-1} \ , \forall \ 1 \leq i \leq j \leq n$$

where r_{ij} is defined for $j > i$ by the following recurrences:

$$\begin{cases} r_{i,i} = p_i^2 \\ r_{ij} = -\dfrac{(n-j+1)(n+j-1)}{(j-1)^2} r_{i,j-1}, \ j > i \end{cases} \quad \begin{cases} p_0 = n \\ p_i = \dfrac{(n-i+1)(n-i-1)}{(i-1)^2} p_{i-1}, \ 1 \leq i \leq n \end{cases}$$

This computation is being done by the `invhilb` command. For example, for $n = 10$ you would say:

```
A = invhilb(10);
```

We can use these matrices to show you the limitations of the `inv` command. Try the following program:

```
for i = 1:20
 x = max(max(abs(invhilb(i)-inv(hilb(i)))));
 fprintf(1,'For size i = %d deviation is x = %g \n',i,x);
end
```

This program will progressively increase the matrix size and attempt to invert the matrix. Then it will compare it with the exact known inverse and report the worst value of the element by element difference. The output of the program is:

```
For size i = 1 deviation is x = 0
For size i = 2 deviation is x = 3.55271e-15
For size i = 3 deviation is x = 6.82121e-13
For size i = 4 deviation is x = 5.96629e-10
```

```
For size i = 5 deviation is x = 1.43918e-08
For size i = 6 deviation is x = 0.000447804
For size i = 7 deviation is x = 0.436005
For size i = 8 deviation is x = 35.4408
For size i = 9 deviation is x = 395134
For size i = 10 deviation is x = 3.657e+08
For size i = 11 deviation is x = 3.17671e+11

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 3.659249e-17

For size i = 12 deviation is x = 2.81767e+14
[....etc....]
```

Notice that the deviation managed to increase all the way to `3.657e+08` before Matlab started giving out warnings. Not good! There are two or three sets of errors involved here:
• The errors caused by representing `hilb(n)`
• The errors caused in the matrix inversion process
• The errors, if any, in representing `invhilb(n)`
It turns out that the first of these, which involves representing fractions like 1/3 and 1/5 in floating-point, is the most significant. This error is propagated throughout the matrix inversion and is significantly amplified.

The moral of the story: Sometimes computers don't answer the question you think you are asking them. You think you are asking the computer to find the inverse. When you use the `inv` function, you are asking it to apply an algorithm which *you* believe will give you the inverse.

Exercise 1 *Write a Matlab function that implements the recurrence formula for the inverse Hilbert matrix. Compare your results with the* `invhilb` *command.*

Exercise 2 *Can we trust the* `det` *function in evaluating the determinant for the Hilbert matrix? Look at the errors in computing:* `det(hilb(n))*det(invhilb(n))` *and* `det(hilb(n)*invhilb(n))` *Which is more accurate? Why?*

4 Algorithms related to solving linear systems

After spending all this time talking about inverses, we need to tell you a little secret: we rarely ever compute inverses explicitly, because even if we did, multiplying them with other vectors or with the original matrix itself is susceptible to many floating point errors. In most problems our main goal is to solve the system

$$\mathbf{Ax} = \mathbf{b}$$

and we can do this without computing the inverse. The purpose of our earlier discussion was to alert you that there are difficulties we have to overcome: it's hard to compute determinants, so

Cramer rule is out of the question; matrices can be nasty and matrix inversion can be error-prone. Now we come to the techniques that help us deal with these difficulties.

The techniques are quite involved, and Matlab wants to hide the complexity for the user. Therefore, Matlab uses the backslash operator to do the job

```
x = A\b;
```

This will work provided that the matrix $\mathbf{A}$ is well-behaved. However, in order for Matlab to be flexible, they provide you with further access to particular techniques that you may want to use to do matrix inversions by yourself.

4.1 LU decomposition

One technique is with the *LU decomposition*. The idea here is that we write $\mathbf{A}$ as the product of a lower triangular matrix $\mathbf{L}$ with 1 in the diagonal, and an upper triangular matrix $\mathbf{U}$. Then we reduce our problem to the following two systems of equations:

$$\mathbf{Ax} = \mathbf{b} \Longleftrightarrow \mathbf{LUx} = \mathbf{b} \Longleftrightarrow \begin{cases} \mathbf{Ly} = \mathbf{b} \\ \mathbf{Ux} = \mathbf{y} \end{cases}$$

These systems can be solved directly without having to find the inverse. As a matter of fact, given the LU decomposition of a matrix you can find the inverse of $\mathbf{A}$ by rewritting the equation

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$$

as n linear systems involving the unknown columns of $\mathbf{A}^{-1}$. Also, since the diagonal elements of $\mathbf{L}$ are 1 and since $\mathbf{U}$ is upper triangular, the determinant of $\mathbf{A}$ can be determined by:

$$\det(\mathbf{A}) = \det(\mathbf{L})\det(\mathbf{U}) = \det(\mathbf{U}) = \prod_{i=1}^{n} u_{ii}$$

The work involved in computing the LU decomposition increases by $O(n^3)$ which is a vast improvement over $O(n!)$. In Matlab, the LU decomposition can be computed with the command:

```
[L,U] = lu(A);
```

Unfortunately, there is a wrinkle with this method as well as with the matrix $\mathbf{L}$ returned by Matlab that has to do with the LU decomposition existance theorem. That theorem states:

Theorem 3 *Let $\mathbf{A} \in \mathcal{C}^{n\times n}$ be a square matrix and $\mathbf{A}_k \in \mathcal{C}^{k\times k}$ be the $k \times k$ submatrix of $\mathbf{A}$ containing the upper left part. Then,*

$$\mathbf{A} \text{ has an LU decomposition} \Longleftrightarrow \forall k \in [n] : \det(\mathbf{A}_k) \neq 0$$

There are many cases where the condition of this theorem will almost fail: one of the submatrix determinants will be close to zero. The workaround is to permute the rows and columns of A in

such a way so that these determinants turn out okey. When you do that you essentially find the LU decomposition of a new matrix **PA** where **P** is a permutation matrix. Permutation matrices have one 1 located in each row and column and when they are applied to another matrix, they permute the other matrix's rows or columns. Since permutations are easy to undo, **P** is yet-another-matrix that's easy to invert: It's inverse is $\mathbf{P}^{-1} = \mathbf{P}^T$. When you run the matlab function `lu` as quoted above, it will not really return an actual lower triangular matrix in `L`. Instead it will do the decomposition $\mathbf{PA} = \mathbf{LU}$ and *return back* $\mathbf{P}^{-1}\mathbf{L}$ *and* $\mathbf{U}$*!*. If you don't like that, you can call `lu` like this:

```
[L,U,P] = lu(A);
```

Now, **L** will be an actual lower triangular matrix and **P** will be a permutation matrix, and the actual decomposition will be

$$\mathbf{A} = \mathbf{P}^T\mathbf{LU}$$

The solution to our system of linear equations then is:

$$\mathbf{x} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{Pb}$$

Now we have a problem: Matlab doesn't have documented facilities for applying $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$ with the well-known back/forward-substitution algorithms. Typically, if you want to solve a system of equations with this method , the backslash operator will do it all in one scoop. However, suppose that you want to solve *many* systems of linear equations in which **A** is the same and only **b** vary. Then, you can have a more efficient program going if do the LU decomposition once and use the `L,U,P` arrays on each different vector `b`. You can write your own function to do that. Then again, if efficiency is such a concern, then perhaps it's time to switch to a programming language.

4.2 QR decomposition

Another technique is the *QR decomposition*. This method will decompose **A** to an *orthogonal* matrix **Q** and a right-triangular matrix **R**. That **Q** is orthogonal means that:

$$\mathbf{Q}^{-1} = \mathbf{Q}^T$$

which makes it easy to invert. Also **R** can be easily inverted being a right-triangular matrix, so by rewriting a linear-systems problem as

$$\mathbf{Rx} = \mathbf{Q}^T\mathbf{b}$$

we can solve it, following the footsteps of the LU technique. You can also compute determinants. It turns out that

$$\det(\mathbf{Q}) = 1$$

so

$$\det(\mathbf{A}) = \det(\mathbf{R}) = \prod_{i=1}^{n} r_{ii}$$

The QR method requires twice as many operations as LU, so LU is more commonly used. However, as you will learn in your numerical methods course, there are many other instances where QR decompositions can come in handy. One application of QR decompositions is in obtaining an orthonormal basis to a vector space. If you think of the columns of $\mathbf{A}$ and the columns of $\mathbf{Q}$ as vectors, then both sets of vectors span the same space. The difference is that the vectors obtained through matrix $\mathbf{Q}$ are orthogonal with one another.

To do a QR decomposition in Matlab call:

```
[Q,R] = qr(A);
```

This will produce two matrices $\mathbf{Q}$ and $\mathbf{R}$ such that $\mathbf{A} = \mathbf{QR}$. Matlab provides a few variations to this call:

```
[Q,R,P] = qr(A);
```

will produce a permutation matrix $\mathbf{P}$, and return in $\mathbf{Q}$ and $\mathbf{R}$ the QR decomposition of $\mathbf{AP}$. The permutation matrix is chosen so that the absolute values of the diagonal elements $|r_{ii}|$ is decreasing as i increases.

4.3 Cholesky decomposition

When a matrix $\mathbf{A} \in \mathcal{R}^{n \times n}$ is *symmetric* and *positive definite* it has a more efficient triangular decomposition. Symmetric means that

$$a_{ij} = a_{ji}, \forall\, i, j \in [n]$$

and positive definite means that

$$\mathbf{x}^T \mathbf{A} \mathbf{x} > 0 \forall \mathbf{x} \in \mathcal{R}^{n \times 1}$$

If both conditions are true, then $\mathbf{A}$ can be decomposed to

$$\mathbf{A} = \mathbf{R}^T \mathbf{R}$$

where $\mathbf{R}$ is an upper triangular matrix and $\mathbf{R}^T$ is the transpose. In complex matrices, you must use the complex conjugate transpose instead. The Cholesky algorithm will attempt to compute this decomposition. The algorithm can self-detect when it fails, and in practice this ability is used to establish whether symmetric matrices *are* positive definite. In Matlab, you can invoke the Cholesky algorithm with the `chol` command:

```
R = chol(A);
```

If the algorithm fails, Matlab will issue a warning. You can suppress the warning by calling `chol` like this:

```
[R,p] = chol(A)
```

If p is non-zero, then this means that the cholesky algorithm failed and the original matrix is not positive definite. Positive definiteness is an important property. Many theorems in linear algebra are known for positive definite matrices, and many times it is useful to be able to decide numerically whether a symmetric matrix is positive definite. Also, when we *know* that the matrix $\mathbf{A}$ in a system $\mathbf{Ax} = \mathbf{b}$ is positive definite, we can use this method to solve the system; it would be faster than LU.

4.4 SVD decomposition

To conclude, another very useful decomposition is the SVD decomposition. The theorem behind SVD decompositions states:

Theorem 4 *If* $\mathbf{A} \in \mathcal{R}^{m \times n}$ *then there exist two orthogonal matrices* $\mathbf{U} \in \mathcal{R}^{m \times m}$ *and* $\mathbf{V} \in \mathcal{R}^{n \times n}$ *such that*

$$\mathbf{U}^T \mathbf{A} \mathbf{V} = diag(\sigma_1, \sigma_2, \ldots, \sigma_p) \in \mathcal{R}^{m \times n}$$

where $p = \min\{m, n\}$ *and* $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$.

To compute the SVD decomposition of a matrix $\mathbf{A}$ in matlab you use the `svd` function as follows:

```
[U,S,V] = svd(A);
```

where `S` will be a diagonal matrix containing the σ_i. If you only want to look at the σ_i, then do

```
s = svd(A);
```

This call will return an array containing just the σ_i. Given the SVD decomposition, the inverse of a square nonsingular matrix $\mathbf{A}$ can be computed with:

$$\mathbf{A}^{-1} = \mathbf{V} \cdot [diag(1/\sigma_j)] \cdot \mathbf{U}^T$$

and the solution to a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ becomes:

$$\mathbf{x} = \mathbf{V} \cdot [diag(1/\sigma_j)] \cdot \mathbf{U}^T \mathbf{b}$$

The reason why SVD is important is because it can diagnose two pathologies: *ill-conditioning* and *singularity*. That you can tell from the *condition number* which is defined by:

$$\text{cond}(\mathbf{A}) = \frac{\max\{\sigma_i\}}{\min\{\sigma_i\}}$$

When the condition number is too large, i.e. close to $1/\epsilon$ where ϵ is the floating point accuracy, then we say that the matrix is *ill-conditioned*. When this happens, merely computing the equation above for the inverse is very susceptible to floating point errors. One remedy is to increase the floating point accuracy to that needed. However, there is a theorem that suggests a better approach. That theorem says that if you take those $1/\sigma_i$ that are really bad and replace them with zero and then do the equation above as if nothing is wrong, then you can get a much more accurate answer than *both* the SVD solution *and* a direct method; accurate in the sense that the residual:

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

will be much smaller than the numerical result yielded by a direct method or the ordinary SVD method. It takes some discretion to decide which $1/\sigma_i$ to kill, and whether you get the desirable accuracy in the residual. A recommended tolerance is to eliminate those σ_i such that

$$\sigma_i < \epsilon \cdot \max\{m, n\} \cdot \max\{\sigma_i\}$$

where m and n are the size of $\mathbf{A}$, and ϵ the floating point precision. The matrix $\mathbf{A}^{-1}$ obtained by the expression above in which the small σ_i have been eliminated is called the *pseudoinverse* matrix. You can use the `pinv` command to compute the pseudoinverse using the SVD method. One way to invoke this command is by saying:

```
B = pinv(A);
```

in which case the tolerance mentioned above is used. Alternatively, you can specify your own tolerance `tol` by invoking the `pinv` command like this:

```
B = pinv(A,tol);
```

It is important to understand that the pseudoinverse matrix is *not* really the same as the inverse matrix. The point is that in an ill-conditioned problem, we are going to get a more accurate answer if we use the pseudoinverse, rather than if we attempt to compute the inverse and use that instead.If, in the worst case scenario, the residual we get by this method is still not good enough then your other option is to increase your floating point precision. To do this, assign the variable `eps` in matlab to your desired precision. For example, to get quad precision set:

```
eps = 1e-32;
```

When the condition number is infinite, that is when some $\sigma_i = 0$ exactly, the matrix has no inverse. Unfortunately, numerics being what it is, if you have an actual singular matrix, you probably won't see exact zeroes because of round-off error. So, it takes some discretion to decide whether you are dealing with a singular matrix or an ill-conditioned matrix as well.

Matlab has a special command for estimating condition numbers:

```
c = cond(A);
```

The `cond` will compute the condition number. There are two other algorithms for condition estimation that can be invoked by the commands `condest` and `rcond`. See the help pages for more information.

The orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ can also give us their share of information. If the n columns of $\mathbf{A}$ are vectors spanning a vector space of interest, then the columns of $\mathbf{U}$ will contain an *orthogonal* set of vectors spanning the same space. Nothing special so far, since you can do this with the QR decomposition too. The bonus is that with SVD you can detect whether the dimensionality of the vector space is the same as the amount of vectors that you use to span the space. If any zeroes (or nearly zeroes) occur in one of the σ_i then this means that the vector space was not in fact n-dimensional. To deal with that case, you just eliminate the columns in $\mathbf{U}$ that correspond to the ill-behaved σ_i. The vectors in $\mathbf{U}$ that remain *will be* an orthogonal basis for the vector space.

An equivalent way of stating the previous paragraph is to say that SVD can be used to determine the *rank* of the matrix $\mathbf{A}$. Rank is the dimensionality of the vector space spanned by the columns of a matrix. [1] We say that a square matrix $\mathbf{A} \in \mathcal{R}^{n \times n}$ is full rank if it has rank n.

[1] it doesn't actually make a difference if you pick the rows of the matrix. A theorem tells you that you will get the same number, always

Many theorems in linear algebra apply only on matrices that are full-rank, so from the numerical perspective we want to have a way to diagnose matrices that are not full rank. SVD provides us with a method to do that.

In addition to rank, we can also determine the *null space* of a matrix $\mathbf{A} \in \mathcal{C}^{n \times m}$. The null space is defined to be

$$\text{null}(\mathbf{A}) = \{\mathbf{x} \in \mathcal{C}^{m \times 1} : \mathbf{A}\mathbf{x} = \mathbf{0}\}$$

that is, the set of vectors $\mathbf{x}$ for which $\mathbf{A}\mathbf{x}$ is zero. For non-singular matrices, the null-space consists only of $\mathbf{x} = \mathbf{0}$. For matrices of $\text{rank}(\mathbf{A}) = r < m$ the last $m - r$ columns of $\mathbf{V}$ give an orthonormal basis for the nullspace. Finally for matrices of $\text{rank}(\mathbf{A}) = r > m$ the null space is $\text{null}(\mathbf{A}) = \{\mathbf{0}\}$

These concepts become useful when you are trying to solve linear systems

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

in which $\mathbf{A} \in \mathcal{C}^{n \times m}$ is no-longer a square matrix. In such cases, if $\mathbf{A}$ actually has a null space, to find the set of $\mathbf{x}$ that satisfy the linear system, we find one particular solution $\mathbf{x}_0$. Then full solution set becomes:

$$\{\mathbf{x} \in \mathcal{R}^{m \times 1} : \mathbf{A}\mathbf{x} = b\} = \{\mathbf{x} + \mathbf{x}_0 : \mathbf{x} \in \text{null}(\mathbf{A})\}$$

In practice, we don't care to know the full solution space. Instead we want to find the unique $\mathbf{x}$ that minimizes the residual

$$r(\mathbf{x}) = |\mathbf{A}\mathbf{x} - \mathbf{b}|$$

The backslash operator will solve this problem. To find the $\mathbf{x}$ that minimizes the residual do:

```
x = A\b;
```

Exercise 3 *Write a matlab function which will accept a set of vectors spanning a linear space in the form of a matrix, and optionally a threshold for the σ_i. Your function should return an orthogonal set of vectors that span the same space, but take care to remove the vectors whose σ_i falls bellow the threshold.*

5 Eigenvalue Problems

A completely different type of problem that shows up in many forms is the *eigenvalue problem*. Let $\mathbf{A} \in \mathcal{C}^{n \times n}$ be a square matrix, $\mathbf{x} \in \mathcal{C}^{n \times 1}$ be a vector and $\lambda \in \mathcal{C}$ be a complex number. We want to find the pairs $(\lambda, \mathbf{x})$ that satisfy the linear equation:

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

The set of λ for which corresponding $\mathbf{x}$ exist are the *eigenvalues* of the matrix. The $\mathbf{x}$ that correspond to each eigenvalue form the corresponding *eigenvector space*. Individual members of that space are called *eigenvectors*. Finding the eigenvalues, and then from these the eigenvectors is the problem at hand.

One approach to solving this problem is to find the eigenvalues first, and then from them, find the eigenvectors. To find the eigenvalues we use this theorem:

Theorem 5 *Let* $\mathbf{A} \in \mathcal{C}^{n \times n}$ *and* $\lambda \in \mathcal{C}$.

$$\lambda \text{is an eigenvalue of } \mathbf{A} \Longleftrightarrow \det(\lambda \mathbf{I} - \mathbf{A}) = 0$$

It turns out that if you expand that determinant, you obtain a polynomial

$$p(\lambda) = \det(\lambda \mathbf{I} - \mathbf{A}) = \sum_{k=1}^{n} a_k x^k$$

which is called the *characteristic polynomial*, so to find the eigenvalues we need find the roots of that polynomial. For matrices up to size $n = 4$ there are direct methods that will give you the solution. So for such matrices, the eigenvalue problem is trivial. For larger polynomials there are no direct methods, and in many cases there is no simpler way to solve polynomial equations except by reducing them to eigenvalue problems! In fact, it is quite common to solve even 3rd order and 4th order polynomials this way.

In Matlab we usually solve eigenvalue problems with the `eig` function. The call:

```
e = eig(A);
```

will return a vector containing the eigenvalues of `A`. The call:

```
[V,D] = eig(A);
```

produces a diagonal matrix `D` of eigenvalues and a full matrix `V` whose columns are the corresponding eigenvectors.

The grand strategy of all eigenvalue algorithms is based on the fact that the matrix $\mathbf{Z}^{-1}\mathbf{A}\mathbf{Z}$ has the same eigenvalues as $\mathbf{A}$ for all non-singular $\mathbf{Z}$. What we want to do then is to apply the appropriate sequence of $\mathbf{Z}$ matrices that will diagonalize our matrix $\mathbf{A}$. Then, the diagonal elements of the diagonal matrix will also be it's eigenvalues. Of course, this is easier said than done. In what follows, we will sketch out how this is done, and introduce you to all the related nifty Matlab functions. You can use these functions to operate on a lower level, and experiment with the eigenvalue algorithms more directly.

5.1 Hessenberg Forms

The first thing we want to do to our matrix $\mathbf{A}$ is bring it to the so-called *Hessenberg Form*. The Hessenberg form of a matrix is zero bellow the first lower subdiagonal and non-zero everywhere else. The existence theorem for Hessenberg forms says that:

Theorem 6 *If* $\mathbf{A} \in \mathcal{C}^{n \times n}$ *then there exists a unitary* $\mathbf{P} \in \mathcal{C}^{n \times n}$ *such that*

$$\mathbf{P}^H \mathbf{A} \mathbf{P} = \mathbf{H}$$

where $\mathbf{H}$ *is such that*

$$h_{ij} = 0, \forall (i,j) : i > j + 1$$

The theoretical emphasis here is that we can bring our $\mathbf{A}$ to this funky form by applying a sequence of *unitary* $\mathbf{Z}$ matrices. In your numerical analysis course you will learn that this guarantees the numerical stability of these repeatitive transformations.

In Matlab, you can get this far by applying the `hess` function:

```
H = hess(A);
```

will return the matrix $\mathbf{H}$. If you also want to look at $\mathbf{P}$, call:

```
[P,H] = hess(A);
```

5.2 Schur Decompositions

The next step is to converge to the *Schur decomposition*. Here are the relevant existence theorems:

Theorem 7 *If $\mathbf{A} \in \mathcal{C}^{n\times n}$ then there exists a unitary $\mathbf{Q} \in \mathcal{C}^{n\times n}$ such that*

$$\mathbf{Q}^H\mathbf{A}\mathbf{Q} = \mathbf{T} = \mathbf{D} + \mathbf{N}$$

where $\mathbf{D} = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$ and $\mathbf{N} \in \mathcal{C}^{n\times n}$ is strictly upper triangular. Furthermore, $\mathbf{Q}$ can be chosen so that the eigenvalues λ_i appear in any order along the diagonal.

This decomposition is called the *Complex Schur Form*. There is another one for real matrices:

Theorem 8 *If $\mathbf{A} \in \mathcal{R}^{n\times n}$ then there exists an orthogonal $\mathbf{Q} \in \mathcal{R}^{n\times n}$ such that*

$$\mathbf{Q}^T\mathbf{A}\mathbf{Q} = \begin{pmatrix} \mathbf{T}_{11} & \mathbf{T}_{12} & \cdots & \mathbf{T}_{1m} \\ 0 & \mathbf{T}_{22} & \cdots & \mathbf{T}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{T}_{mm} \end{pmatrix}$$

where each $\mathbf{T}_{ii}$ is either a 1×1 matrix which is an eigenvalue of $\mathbf{A}$ or a 2×2 matrix whose eigenvalues are also eigenvalues of $\mathbf{A}$.

This one is called the *Real Schur Form*.

Matlab provides with a function called `schur`. The call

```
[Q,T] = schur(A)
```

will return the real schur form if the matrix is real, and the complex schur form if the matrix is complex. If you don't want the real schur form, then the function `rsf2csf` will convert the inputted real schur form to complex:

```
[Q,T] = rsf2csf(Q,T);
```

How do we get to Schur from Hessenberg form? The simplest way to do it is with the so called QR algorithm:

1. First of all, we get the Hessenberg form of $\mathbf{A}$:

$$\mathbf{H}_0 = \mathbf{P}_0^T \mathbf{A} \mathbf{P}_0$$

2. Then we apply the following recurrence:

$$\begin{cases} \mathbf{H}_k = \mathbf{Q}_k \mathbf{R}_k & \text{QR decomposition} \\ \mathbf{H}_{k+1} = \mathbf{R}_k \mathbf{Q}_k \end{cases}$$

It can be proven that these repetive iterations are actually equivalent to applying unitary matrices for $\mathbf{Z}$ and that they will take the Hessenberg form all the way to *Real Schur form*! In your numerical analysis course you will learn more details about the various algorithms that take you from Hessenberg to Schur. With Matlab you can actually experiment with many of these algorithms and see how they perform.

5.3 Balancing

Before applying the QR algorithm it is a good idea to *balance* the matrix. By balancing we rescale the matrix by transforming it to $\mathbf{D}^{-1}\mathbf{A}\mathbf{D}$ where $\mathbf{D}$ is a *diagonal* matrix so that it it has approximately equal row and column norms. In matlab, you can balance a matrix with the `balance` function:

```
B = balance(A);
```

While experimenting with variations of the QR algorithm, you can see what happens as you include or not include balancing.

5.4 Miscellaneous

With symmetric matrices, the following miracles happen: the eigenvalues are real, the Hessenberg form is a tridiagonal matrix, and the QR algorithm doesn't merely take us to a real Schur decomposition, but it completely diagonalizes our matrix! An even more exciting development is that for square symmetric matrices the Schur decomposition and the SVD decomposition *are the same thing!* From this fact, plus a few theoretical results, one can obtain an algorithm for computing the SVD by reducing it to a symmetric eigenvalue problem. The details of this are described in *"Matrix Computations"* by Gene Golub.

Another application of eigenvalue problems is solving polynomial equations. As you may know, direct methods for finding polynomial roots, exist only for polynomials up to 4^{th} degree. Let

$$P(x) = \sum_{k=1}^{n-1} a_k x^k + x^n$$

be your polynomial. Form the matrix:

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & \cdots & 0 & -a_0 \\ 1 & 0 & \cdots & 0 & -a_1 \\ 0 & 1 & \cdots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -a_{n-1} \end{pmatrix}$$

The eigenvalues of $\mathbf{C}$ are the roots of $P(x)$. $\mathbf{C}$ is called the *companion matrix* of the polynomial. In matlab, if an array `P` holds the coefficients of a polynomial with `p(1)` being the highest order coefficient, the companion matrix can be retrieved by the `compan` function.

```
A = compan(p);
```

Of course Matlab, in it's never-ending efforts to make this all easier for the user, provides you with a function called `roots` that you can use to do all this automatically. So, if you describe your polynomial with an array `p` such that

```
y = p(1)*x^d + p(2)*x^(d-1) + ... + p(d)*x + p(d+1)
```

then the roots can be obtained by simply saying:

```
r = roots(p);
```

You will obtain an array of roots in `r`. To verify your roots, you can use the `polyval` function to evaluate the polynomial. Just call:

```
y = polyval(p,x);
```

where `p` is the array with the polynomial and `x` is a scalar.

Finally, Matlab has commands that allow the user to solve more generalized eigenvalue problems. One such problem is the following. Given two square matrices $\mathbf{A}, \mathbf{B} \in \mathcal{C}^{n \times n}$ find $\lambda \in \mathcal{C}$ and $\mathbf{x} \in \mathcal{C}^{n \times 1}$ such that

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{B}\mathbf{x}.$$

To solve this problem call:

```
[V,L] = eig(A,B);  % eigenvectors -> V, eigenvalues -> L
l = eig(A,B);      % eigenvalues  -> l
```

In the first version the eigenvectors are returned as columns of `V` and the eigenvalues as diagonals of `L`. In the second version an array of the eigenvalues is returned.