Notes for Ma221 Lecture 9, Oct 17, 2024

Begin Chapter 4 on eigenvalue problems
Goals:
   Canonical Forms (recall Jordan, why we want Schur instead)
   Variations on eigenproblems (not always just one matrix!)
   Perturbation Theory (can I trust the answer?)
   Algorithms (for a single nonsymmetric matrix)

Recommended reading: Templates for the Solution of Algebraic Eigenvalue Problems
on class webpage.

Recall basic definitions for a square n x n matrix A:

Def: p(lambda) = det(A - lambda*I) is the characteristic polynomial, whose n roots
are the eigenvalues of A

Def: If lambda is an eigenvalue, a nonzero null vector x satisfying (A-lambda*I)*x =0
must exist, i.e. A*x = lambda*x, and is called a right eigenvector. Analogously
a nonzero null vector y^H must exist such that y^H * A = lambda*y^H, and is called
a left eigenvector.

Def: If S is nonsingular, and B = S*A*inv(S), then S is called a similarity
transformation, and A and B are called similar matrices.

Lemma: If A and B are similar, they have the same eigenvalues, and the eigenvectors
are related by multiplying by S:
Proof: A*x = lambda*x iff S*A*inv(S)*S*x = S*lambda*x or B*(S*x) = lambda*(S*x)
ie. iff lambda is also an eigenvalue of B and S*x is a right eigenvector of B
Analogously, y^H * A = lambda*A^H iff y^H * inv(S)*S*A*inv(S) = lambda * y^H * inv(S)
or (y^H * inv(S)) * B = lambda * (y^H * inv(S)), i.e. iff y^H * inv(S) is
a left eigenvector of B.

Our goal will be to take A and transform it to a simpler similar form B, from which
its eigenvalues and eigenvectors are easy to extract. The simplest form, for which
eigenvalues and eigenvectors are obvious, is a diagonal matrix D, since
D*e(i) = D(i,i)*e(i), where e(i) is the i-th column of I.

Lemma: Suppose A*x(i) = lambda(i)*x(i) for i=1 to n, and that the matrix
S = [x(1),...,x(n)] is nonsingular, i.e. x(i) are n linearly independent
eigenvectors.
Then A = S*diag(lambda(1),...,lambda(n))*inv(S). Conversely, if A = S*Lambda*inv(S)
where Lambda is diagonal, then the columns of S are eigenvectors and the Lambda(i,i)
are eigenvalues.
Proof: A = S*Lambda*inv(S) if and only if A*S = S*Lambda if and only if
the i-th columns of both sides are the same, i.e. A*S(:,i) = S(:,i)*Lambda(i,i)

But we can't always make B = S*A*inv(S) diagonal, for two reasons:
  It may be mathematically impossible (recall Jordan form, with multiple eigenvalues)
  It may be numerically unstable (even if the Jordan form is diagonal)

Recall Jordan Form: for any A there exists a similar matrix J = S*A*inv(S) such that
    J = diag(J_1,...,J_k) where each J_i is a Jordan block:

$$J_i = \begin{bmatrix} \lambda & 1 & 0 & \cdots & 0 \\ 0 & \lambda & 1 & 0 & \cdots & 0 \\ & & \cdots & & & \\ 0 & \cdots & & \lambda & 1 \\ 0 & \cdots & & & 0 & \lambda \end{bmatrix}$$

Up to permuting the order of the J_i, the Jordan form is unique.
Different J_i can have the same eigenvalue lambda (eg A = I).

There is only one (right) eigenvector per J_i (namely [0, ... , 0,1,0, ... 0]
with the 1 in the same row as the top row of J_i)).
So a matrix may have n eigenvectors (if there are n 1x1 J_i's; the matrix is called
diagonalizable in this case) or fewer (in which case it is called defective).
The number of times one lambda appears on the diagonal is called its multiplicity.

But we will not compute the Jordan form, for numerical reasons (though algorithms
do exist). Consider the following slightly perturbed 2x2 identity matrices:
what are their eigenvalue and eigenvectors?

```
    (1)   [ 1 0   ]  :  ( 1, [ 1 ] ) and ( 1+e, [ 0 ] )        ... simplest case
          [ 0 1+e ]          [ 0 ]              [ 1 ] )
    (2)   [ 1 e ]    :  ( 1+e, [ 1 ] ) and ( 1-e, [  1 ] )     ... rotated 45 degrees
          [ e 1 ]             [ 1 ]              [ -1 ] )
    (3)   [ 1   e   ]    :  ( 1, [ 1 ] ) and ( 1+e^2, [ 1 ] )  ... nearly parallel
          [ 0 1+e^2 ]           [ 0 ]                [ e ] )
    (4)   [ 1 e ]    :  ( 1, [ 1 ] )                          ... only one
          [ 0 1 ]           [ 0 ]
    (5)   [ 1 0 ]    :  ( 1, anything )  and (1, anything )    ... not unique
          [ 0 1 ]
```

This means that when there are (nearly) multiple eigenvalues, the Jordan Form
is very ill-conditioned (and may be discontinuous), which makes computing
it fraught with numerical peril.

The best we can generally hope for, as in earlier chapters, is backwards stability:
Getting exactly the right eigenvalues and similarity S for a slightly perturbed
input matrix A + E, where || E || = O(macheps)*|| A ||.

In the last chapter we said that as long as you multiply a matrix by orthogonal
matrices, it is backward stable, i.e.

```
     fl( Q(k)*Q(k-1)*...*Q(1)*A ) = Q(A+E)  where Q is exactly orthogonal, and
                 || E || = O(macheps)*|| A ||
```

If we apply this to computing an orthogonal similarity transformation, we get

```
    fl ( Q(k)*Q(k-1)*...*Q(1) * A * Q(1)^T*...*Q(k-1)^T*Q(k)^T ) = Q*(A+E)*Q^T
```

i.e. the exact orthogonal similarity of the slightly perturbed input A+E.

This means that if we can restrict the similarity transforms S we use in S*A*inv(S)
to be orthogonal, we get backwards stability. So the question is: if we restrict
S to be orthogonal, how close to Jordan form can we get?

Theorem (Schur Canonical Form): Given any square A there is a unitary Q such that
Q^H * A * Q = T is upper triangular. The eigenvalues are the diagonals T(i,i),
which can be made to appear on the diagonal of T in any order.

Note that eigenvectors are easy to compute from the Schur form if you need them:
T*x = T(i,i)*x turns into solving a triangular system:

```
[ T11 T12    T13 ] * [ x1 ] = T(i,i) * [ x1 ] => T11*x1 + T12*x2 + T13*x3 = T(i,i)*x1
[  0  T(i,i) T23 ]   [ x2 ]            [ x2 ]         T(i,i)*x2 + T23*x3 = T(i,i)*x2
[  0  0      T33 ]   [ x3 ]            [ x3 ]                     T33*x3 = T(i,i)*x3
```

If there is only one copy of eigenvalue T(i,i), then the only solution of
T33*x3 = T(i,i)*x3 is x3=0. Then T(i,i)*x2 = T(i,i)*x2 has any solution x2; pick
x2 = 1. Finally we solve (T11-T(i,i)*I)*x1 = -T12, a triangular system for x1.
If T(i,i) is a multiple eigenvalue, then T11 - T(i,i)*I might be singular, so we
might not be able to solve (T11-T(i,i)*I)*x1 = -T12, as expected.

Proof of Theorem: We use induction: Let x be a right eigenvector with ||x||_2 = 1,
and let Q = [x,Q'] be any unitary matrix with x as its first column. Then

```
     Q^H * A * Q = [ x^H  ] * A * [ x, Q' ] = [ x^H * A * x   x^H * A * Q' ]
                   [ Q'^H ]                   [Q'^H * A * x   Q'^H * A * Q']
                 = [ lambda * x^H x     x^H * A * Q'  ] = [ lambda   x^H * A * Q'  ]
                   [ lambda * Q'^H * x  Q'^T * A * Q' ] = [   0      Q'^T * A * Q' ]
```

Now we apply induction to the smaller matrix Q'^H * A * Q' to write it as U^H * T * U where T is upper triangular and U is unitary, so

```
Q^H*A*Q = [ lambda   x^H * A * Q' ] = [ 1  0 ] * [ lambda  x^H*A*Q'*U^H ] * [ 1 0 ]
          [   0      U^H * T * U  ]   [ 0 U^H]   [   0         T        ]   [ 0 U ]
    or    [ 1 0 ] * Q^H * A * Q * [ 1 0   ] = [ lambda  stuff ] as desired
          [ 0 U ]                 [ 0 U^H ]   [    0     T   ]
```

But there is still an obstacle: Real matrices can have complex eigenvalues (unless, say, they are symmetric, the topic of Chap 5). So T may have to be complex even if A is real; we'd prefer to keep arithmetic real if possible, for various reasons (reduce #flops, less memory, make sure complex eigenvalues and eigenvectors come in conjugate pairs despite roundoff).
So instead of a real triangular T, we will settle for a real block triangular T:

```
    T = [ T11 T12 ... T1k ]
        [  0  T22 ... T2k ]
        [      ...        ]
        [  0   0  ... Tkk ]
```

The eigenvalues of T are just the union of the eigenvalues of all the Tii (Q 4.1). We will show that we can reduce any real A to such a block triangular T where each Tii is either 1x1 (so a real eigenvalue) or 2x2 (with two complex conjugate eigenvalues).

Theorem (Real Schur Canonical Form) Given any real square A, there is a a real orthogonal Q such that Q*A*Q^T is block upper triangular with 1x1 and 2x2 blocks.

To prove this we need to generalize the notion of eigenvector to "invariant subspace":

Def: Let V = span{x1,...,xm} = span(X) be a subspace of R^n. It is called an invariant subspace if A*V = span(A*X) is a subset of V.

Ex: V = span{x} = {alpha*x for any scalar alpha} where A*x = lambda*x, then
A*V = {A*(alpha*x) for any alpha} = {alpha*lambda*x for any alpha} is in span(x)=V
(when would A*V not equal V?)

Ex: V = span{x(1),...,x(k)} ={sum_{i=1 to k} alpha(i)*x(i) for any scalars alpha(i)}
where A*x(k) = lambda(k)*x(k), then
A*V = {A*sum_{i=1 to k} alpha(i)*x(i) for any alpha(i) }
= {sum_{i=1 to k} A*alpha(i)*x(i) for any alpha(i) }
= {sum_{i=1 to k} alpha(i)*lambda(i)*x(i) for any alpha(i) }
is a subset of V (when would A*V not equal V?)

Lemma: If V = span{x(1),...,x(m)} = span{X} is an m-dimensional invariant subspace of A (i.e. x(1),...,x(m) are independent) then there is an mxm matrix B such that A*X = X*B. The eigenvalues of B are also eigenvalues of A.
Proof: The existence of B follows from the definition of invariant subspace: A*x(i) in V means there are scalars B(1,i),...,B(m,i) (the i-th column of B) such that A*x(i) = sum_{j=1 to m} x(j)*B(j,i). If B*y = lambda*y, then A*X*y = X*B*y = X*y*lambda, so X*y is an eigenvector of A with eigenvalue lambda.

Lemma: Let V = span{X} be an m-dimensional invariant subspace of A as above, with A*X=X*B. Let X = Q*R, and let [Q,Q'] be square and orthogonal. Then

```
[Q,Q']^T * A * [Q,Q'] = [ A11  A12 ] is block upper triangular
                        [  0   A22 ]
```

with A11 = R * B * inv(R) having the same eigenvalues as B.
Proof:

```
[Q,Q']^T * A * [Q,Q'] = [ Q^T * A * Q   Q^T * A * Q' ] = [ A11 A12 ]
                        [ Q'^T* A * Q   Q'^T* A * Q' ]   [ A21 A22 ]
```

where A * Q = A * X * inv(R) = X * B * inv(R) = Q * R * B * inv(R)
so A11 = Q^T * Q * R * B * inv(R) = R * B * inv(R)
and A21 = Q'^T * Q * R * B * inv(R) = 0

Proof of Real Schur Form: We use induction as before. If A*x = lambda*x where lambda and x are real, we reduce to a smaller problem using the last Lemma.
If lambda and x are complex, it is easy to confirm that the real and imaginary parts of A*x=lambda*x are equivalent to the first and second columns of A*X = X*B, where X = [Re(x),Im(x)] and B = [Re(lambda) Im(lambda)]
[-Im(lambda) Re(lambda)]
and that B's eigenvalues are lambda and conj(lambda).
So by the Lemma we can do an orthogonal similarity on A to get [A11 A12]
[0 A22]
where the eigenvalues of A11 are lambda and conj(lambda), completing the induction.

More general eigenvalue problems:

We briefly review other kinds of eigenvalues that can arise, beyond a single square matrix. In Lecture 1, we pointed out that ODEs can give rise to a range of eigenvalue problems:

(1) In the simplest case, the ODE x'(t) = K*x(t) leads to the eigenvalue problem for K: if K*x(0) = lambda*x(0), then x(t) = exp(lambda*t)*x(0), and similarly if x(0) is expressed as a linear combination of eigenvectors.

(2) When M*x''(t) + K*x(t) = 0, and lambda^2*M*x(0) + K*x(0) = 0, then x(t) = exp(lambda*t)*x(0). This is a "generalized eigenproblem" for the pair (M,K), with eigenvalue lambda^2 and eigenvector x(0).
The usual definition of an eigenvalue becomes det(lambda'*M + K) = 0, where lambda' = lambda^2.

(3) When M*x''(t) + D*x'(t) + K*x(t) = 0, we get the "nonlinear eigenproblem" lambda^2*M*x(0) + lambda*D*x(0) + K*x(0) = 0, which can be reduced to a linear generalized eigenproblem of twice the size.

(4) When x'(t) = A*x(t) + B*u(t), a linear control system, the question of how to choose u(t) to control x(t) turns into a "singular eigenproblem" for the pair of rectangular matrices [B,A] and [0,I].

More generally, all the ideas of this chapter (eigenvalues, eigenvectors, Jordan form,
Schur form, algorithms) extend to these more general eigenvalue problems, see section 4.5 of the textbook for details. We will only discuss the eigenproblem for one square matrix in detail.

Perturbation Theory: How can I trust my answer?

Recall that the best we can hope for is backward stability: right answer (eigenvalues)
for a slightly wrong problem A + E, where ||E|| = O(macheps)*||A||. How much can this change the eigenvalues and vectors?

Last time: showed that if eigenvalues close together, eigenvectors can be very sensitive (or disappear, or be nonunique, as for I). How about the eigenvalues?

To describe perturbations we consider
Def: The epsilon pseudo-spectrum of A is the set of all eigenvalues of all matrices within distance epsilon of A:
Lambda_eps(A) = {lambda: (A+E)x = lambda*x for some nonzero x and some ||E||_2 <=eps}

Ideal case: Lambda_eps(A) = union of disks of radius eps centered at eigenvalues of A
Will show this is true for A = A^H (Chapter 5).
Worst case: Thm (Trefethen & Reichel): Given any simply connected region R in the

complex plane, and point x in R, and any eps>0, there is an A with one eigenvalue at x such that Lambda_eps(A) is as close to filling out R as you like. (picture)
The proof is a simple consequence of the Riemann Mapping Theorem.

Example: Perturb nxn Jordan block at 0 by changing J(n,1) = eps, get eigenvalues on circle of radius eps^(1/n), which is >> eps. This example shows
(1) that eigenvalues are not necessarily differentiable functions of matrix entries (slope of eps^(1/n) is infinite at eps=0), although they are continuous (and differentiable when not multiple)
(2) gives intuition that we should expect a sensitive eigenvalue when it is (close to) multiple, as was the case for eigenvectors.

Let us find the condition number of a simple (nonmultiple) eigenvalue:
Thm: Let lambda be a simple eigenvalue, with A x = lambda x and y^H A = lambda y^H, and ||x||_2 = ||y||_2 = 1.
If we perturb A to A + E the lambda is perturbed to lambda + dlambda, and
dlambda = (y^H E x) / y^H x + O(||E||^2)
|dlambda|<= ||E|| / |y^H x| + O(||E||^2) = sec(theta) * ||E|| + O(||E||)^2
where theta is the acute angle between x and y. So sec(theta) is the condition number of lambda.
Proof: Subtract A x = lambda x from (A+E)(x+dx) = (lambda + dlambda)(x+dx) to get
A dx + E x + E dx = lambda dx + dlambda x + dlambda dx
Ignore second order terms E dx and dlambda dx, and multiply by y^H to get
y^H A dx + y^H E x = lambda y^H dx + dlambda y^H x
Cancel y^HA dx = lambda y^H dx to get y^H E x = dlambda y^H x as desired.
Note that a Jordan block has x = e(1) and y = e(n), so y^H x = 0 as expected.

An important special case are real symmetric matrices (or more generally normal matrices, where A^H A = A A^H), since these have orthogonal eigenvectors:
Corollary: If A is normal, perturbing A to A+E means | dlambda | <= ||E||+ O(||E||^2)
Proof: A = Q Lambda Q^H is the eigendecomposition, where Q is unitary, so
A*Q = Q*Lambda, and the right eigenvectors are the columns of Q, and
Q^H A = Lambda Q^H, and the left eigenvectors are also the columns of Q, so x = y.

Later, in Chapter 5, for real symmetric matrices A=A^T (or more generally complex Hermitian matrices A = A^H), we will show that if E is also symmetric, then
| dlambda | <= ||E|| no matter how big ||E|| is.

The above theorem is true for small ||E||. It is possible to change it slightly to work for any ||E||:

Thm (Bauer-Fike): Let A have all simple eigenvalues (i.e. be diagonalizable). Call them lambda_i with right and left eigenvectors x_i and y_i, normalized so ||x_i||_2 = ||y_i||_2 = 1. Then for any E the eigenvalues of A+E like in the union of disks D_i in the complex plane, where D_i has center lambda_i and radius n||E||_2 / |y_i^H x_i|.

Note that this is just n times larger than the last theorem. Also note that if two disks D_i and D_j overlap, all the theorem guarantees is that there are two eigenvalues of A+E in the union D_i U D_j (the same idea applies if more disks overlap). (see book for proof).

Algorithms for the Nonsymmetric Eigenproblem

Our ultimate algorithm, the Hessenberg QR algorithm, takes a nonsymmetric A and computes the Schur form A = Q T Q^H, in O(n^3) flops. We will build up to it with simpler algorithms, that will also prove useful as building blocks for the algorithms for sparse matrices, where we only want to compute a few eigenvalues and vectors. The Hessenberg QR algorithm will also be used as a building block for large sparse matrices, because our algorithms for them will approximate them (in a certain sense)

by much smaller dense matrices, to which we will apply Hessenberg QR.

The plan is as follows:
    Power Method: Just repeated multiplication of a vector by A; we'll show
      this makes the vector converge to the eigenvector for the eigenvalue
      of largest absolute value, which we also compute.
    Inverse Iteration: Apply power method to B = (A - sigma*I)^(-1), which has
      the same eigenvectors as A, but now the largest eigenvalue in absolute
      value of B corresponds to the eigenvalue of A closest to sigma (which is
      called the "shift"). By choosing sigma appropriately, this lets us get
      any eigenvalue of A, not just the largest.
    Orthogonal Iteration: This extends the power method from one eigenvector to
      compute a whole invariant subspace.
    QR iteration: we combine Inverse Iteration and Orthogonal Iteration to
      get our ultimate algorithm.

There are a lot of other techniques needed to make QR iteration efficient
(run in O(n^3)) and reliable, as well as to reduce data movement. We will
only discuss some of these.

Power Method: given x(0), we iterate
    i=0
    repeat
       y(i+1) = A*x(i)
       x(i+1) = y(i+1) / ||y(i+1)||_2   ... approximate eigenvector
       lambda'(i+1) = x(i+1)^T * A * x(i+1)   ... approximate eigenvalue
       i = i+1
    until convergence

We first analyze convergence when A = diag(lambda(1),...,lambda(n))
where |lambda(1)| > |lambda(2)| >= |lambda(3)| >= ... and then generalize:

We note that x(i) = A^i * x(0) / || A^i * x(0) ||_2
and that A^i * x(0) = [lambda(1)^i * x(0)_1 , lambda(2)^i * x(0)_2 , ...]
                    = lambda(1)^i [x(0)_1 , (lambda(2)/lambda(1))^i * x(0)_2 , ...]
so x(i) = [x(0)_1 , (lambda(2)/lambda(1))^i * x(0)_2 , ...] / || " ||_2
As i grows, each (lambda(j)/lambda(1))^i converges to 0, and x(i) converges to
[1,0,...,0] as desired, with error O(|lambda(2)/lambda(1)|^i), assuming x(0)_1 neq 0

More generally, suppose A is diagonalizable, with A = S Lambda S^(-1), so
A^i = S Lambda^i S^(-1). Let z = S^(-1) * x(0), so
A^i * x(0) = S * [lambda(1)^i * z_1 ; lambda(2)^i * z_2 ; ...]
           = lambda(1)^i [z_1* S(:,1) + (lambda(2)/lambda(1))^i * z_2 * S(:,2) +...]
As i increases, the vector in [ ] converges to z_1 * S(:,1), i.e. a multiple of the
first column of S, i.e. since A*S = S*Lambda, the eigenvector of A for lambda(1),
as desired.

For this to converge to the desired eigenvector at a reasonable rate, we need
(1) |lambda(2)/lambda(1)| < 1, and the smaller the better. This is not necessarily
true, and for an orthogonal matrix A*A^T = I, all the eigenvalues have
absolute value 1 (since ||x|| = ||A*x|| = ||lambda*x||), so there is no convergence.
(2) z_1 nonzero, and the larger the better. If we pick x(0) at random, it is very
unlikely that z_1 will be very tiny, but there are no guarantees.

To deal with needing |lambda(1)| >> |lambda(2)| to get fast convergence, we use
inverse iteration, i.e. the power method on B = (A - sigma*I)^(-1), where sigma
is called the shift.

Inverse iteration: given x(0), we iterate
    i=0

```
    repeat
       y(i+1) = (A-sigma*I)^(-1)*x(i)
       x(i+1) = y(i+1) / ||y(i+1)||_2        ... approximate eigenvector
       lambda'(i+1) = x(i+1)^T * A * x(i+1)   ... approximate eigenvalue
       i = i+1
    until convergence
```

The eigenvectors of B are the same as those of A, but its eigenvalues are
1/(lambda(i) - sigma). Suppose sigma is closer to lambda(k) than any other eigenvalue
of A. Then the same kind of analysis as above shows that x(i) is gotten by the taking
the following vector divided by its norm:

```
      [ ((lambda(k) - sigma)/(lambda(1) - sigma))^i * z(1)/z(k) ]
      [ ((lambda(k) - sigma)/(lambda(2) - sigma))^i * z(2)/z(k) ]
      [                          ...                            ]
      [                           1                             ]  ... k-th component
      [                          ...                            ]
      [ ((lambda(k) - sigma)/(lambda(n) - sigma))^i * z(n)/z(k) ]
```

So if we can choose sigma much closer to lambda(k) than any other lambda(j), we can
make convergence as fast as we want. Where do we get sigma? Once we start
converging, the algorithm itself computes an improving estimate of lambda(k)
at each iteration; we will see later that this makes convergence very fast, quadratic
or even cubic in some cases.

The next step is to compute more than one vector at a time.
We do this first for the analogue of the power method:

Orthogonal Iteration: given Z(0), an n x p orthogonal matrix, we iterate

```
    i = 0
    repeat
       Y(i+1) = A*Z(i)
       factor Y(i+1) = Z(i+1)*R(i+1)  ... QR decomposition
              ... Z(i+1) spans an approximate invariant subspace
       i=i+1
    until convergence
```

(Note the similarity to the randomized algorithms discussed previously, where Z(0)
was chosen randomly; these techniques can be combined, though we don't discuss
any more details.)
Here is an informal analysis, assuming A = S*Lambda*S^(-1) is diagonalizable and

```
   | lambda(1) | >= | lambda(2) | >= ... >= | lambda(p) | > | lambda(p+1) | >= ...
```

i.e. the first p eigenvalues are larger in absolute value than the others.
Note that

```
    span{Z(i+1)} = span{Y(i+1)} = span{A*Z(i)} = ...
  = span{A^i*Z(0)} by induction
  = span(S*Lambda^i*S^(-1)*Z(0))
```

Now

```
    S * Lambda^i * S^(-1) * Z(0)
  = S * lambda(p)^i *
    diag((lambda(1)/lambda(p))^i ,..., 1 ,
         (lambda(p+1)/lambda(p))^i ,...)*S^(-1)*Z(0)
  = S * lambda(p)^i [ V(i) ]    p x p
                    [ W(i) ]  n-p x p
```

where (lambda(j)/lambda(p))^i goes to infinity if j < p and goes to 0 if j > p.
Thus W(i) goes to zero, and V(i) does not. If V(0) has full rank, so will V(i). Thus

```
   A^i*Z(0) = lambda(p)^i * S * [ V(i) ] approaches lambda(p)^i * S * [ V(i) ]
                                [ W(i) ]                              [  0   ]
```

so it is a linear combination of the first p columns of S, i.e. the first
p eigenvectors, the desired invariant subspace.

Note that the first k < p columns of Z(i) are the same as though we had run the

algorithm starting with the first k columns of Z(0), because the k-th column of
Q and R in the QR decomposition of A=QR only depend on columns 1:k of A.
In other words, Orthogonal Iteration runs p different iterations simultaneously,
with the first k columns of Z(i) converging to the invariant subspace spanned by
the first k eigenvectors of A.

Thus we can let p = n and Z(0) = I, and try to compute n invariant subspaces
at the same time. This will give us Schur Form:

Theorem: Run Orthogonal iteration on A with Z(0) = I. If all the eigenvalues of
A have different absolute values, and if the principal submatrices S(1:k,1:k) of
the matrix of eigenvectors of A all have full rank, then A(i) = Z(i)^T * A * Z(i)
converges to Schur form, i.e. diagonal with eigenvalues on the diagonal

```
Matlab (demo), try
      n=6, D = diag(.5.^[1:n]), S = randn(n,n), A = S*D*inv(S), Z = eye(n);
   and repeat
     Y = A*Z; [Z,R]= qr(Y); Z'*A*Z
```

Proof: By the previous analysis, for each k, the span of the first k columns of Z(i)
converge to the invariant subspace spanned by the first k eigenvectors of A.
Write Z(i) = [Z(i)_1 , Z(i)_2] where Z(i)_1 has k columns so

```
   Z(i)^H * A * Z(i) = [ Z(i)_1^H * A * Z(i)_1  Z(i)_1^H * A * Z(i)_2 ]
                       [ Z(i)_2^H * A * Z(i)_1  Z(i)_2^H * A * Z(i)_2 ]
```

Now A * Z(i)_1 converges to Z(i)_1 * B(i) since Z(i)_1 is converging to an invariant
subspace, so Z(i)_2^H * A * Z(i)_1 converges to Z(i)_2^H * Z(i)_1 * B(i) = 0.
Since this is true for every k, Z(i)^H * A * Z(i) converges to upper triangular form
T(i). Since Z(i) is unitary, this is the Schur form.

The next step is to rewrite Orthogonal Iteration as QR iteration, which
will let us incorporate inverse iteration, and so converge rapidly to
any eigenvalue for which we have a good approximation (which will be supplied
by the algorithm!).

```
QR Iteration: Given A(0) = A we iterate
   i = 0
   repeat
     factor A(i) = Q(i)*R(i)   ... QR decomposition
     A(i+1) = R(i)*Q(i)
     i = i + 1
   until convergence
```

Note that A(i+1) = R(i)*Q(i) = Q(i)^T * Q(i) * R(i) * Q(i) = Q(i)^T * A(i) * Q(i)
so that all the A(i) are orthogonally similar.

Thm: A(i) from QR iteration is identical to Z(i)^T*A*Z(i) from Orthogonal iteration,
starting with Z(0) = I. Thus A(i) converges to Schur Form if all the eigenvalues
have different absolute values.

Proof: We use induction: assume A(i) = Z(i)^T*A*Z(i). Then taking one step of
Orthogonal iteration we write A*Z(i) = Z(i+1)*R(i+1), the QR decomposition.
Then A(i) = Z(i)^T*A*Z(i) = Z(i)^T*Z(i+1)*R(i+1) = orthogonal * upper triangular,
so this must also be the QR decomposition of A(i) (by uniqueness). Then

```
     Z(i+1)^T * A * Z(i+1) = Z(i+1)^T * A * (Z(i)*Z(i)^T) * Z(i+1)
                           = (Z(i+1)^T * A * Z(i) ) * ( Z(i)^T * Z(i+1) )
                           = ( R(i+1) ) * (Z(i)^T * Z(i+1))
                           = R * Q,
                           = A(i+1)
```

where Q*R = Z(i)^T*A*Z(i), i.e. we have taken one step of QR iteration.

Now we show how to incorporate inverse iteration:

QR iteration with a shift: Given A(0) = A, we iterate
```
    i = 0
    repeat
      choose a shift sigma(i) near an eigenvalue of A
      factor A(i) - sigma(i)*I = Q(i)*R(i)   ... QR decomposition
      A(i+1) = R(i)*Q(i) + sigma(i)*I
      i = i+1
    until convergence
```

Lemma: A(i) and A(i+1) are orthogonally similar.
```
Proof:   A(i+1) = R(i)*Q(i) + sigma(i)*I
                = Q(i)^T*Q(i)*R(i)*Q(i) + sigma(i)*I
                = Q(i)^T*(A(i)-sigma(i)*I)*Q(i) + sigma(i)*I
                = Q(i)^T*A(i)*Q(i)
```

If R(i) is nonsingular, we can also write
```
         A(i+1) = R(i)*Q(i) + sigma(i)*I
                = R(i)*Q(i)*R(i)*R(i)^(-1) + sigma(i)*I
                = R(i)*(A(i)-sigma(i)*I)*R(i)^(-1) + sigma(i)*I
                = R(i)*A(i)*R(i)^(-1)
```

If sigma(i) is an exact eigenvalue of A, we claim QR Iteration
converges in one step: If A(i) - sigma(i)*I is singular, then
R(i) is singular, so some diagonal entry of R(i) must be zero.
Suppose that the last diagonal entry R(i)_nn = 0. Then the
whole last row of R(i) is zero, so the last row of R(i)*Q(i) is zero,
so the last row of A(i+1) = R(i)*Q(i) + sigma(i)*I is zero except
for A(i+1)_nn = sigma(i) as desired. This reduces the problem to
one of dimension n-1, namely the first n-1 rows and columns of A(i+1).

If sigma(i) is not an exact eigenvalue, we declare convergence when
A(i+1)_n,1:n-1 is small enough. From earlier analysis we expect
this block to shrink in norm by a factor
```
     | lambda(k) - sigma(i) | / min_{j neq k} | lambda(j) - sigma(i) |
```
where lambda(k) is the eigenvalue closest to sigma(i).

Here is how to see that we are implicitly doing inverse iteration. For simplicity,
we assume the eigenvalue is real. First, since A(i) - sigma(i)*I = Q(i)*R(i), we get
```
        Q(i)^T * (A(i) - sigma(i)*I) = R(i)
```
so if sigma(i) is an exact eigenvalue, the last row of Q(i)^T times A(i) - sigma(i)*I
is zero, and so the last column of Q(i) is a left eigenvector of A(i) for eigenvalue
sigma(i). Now suppose that sigma(i) is just close to an eigenvalue.
Then A(i) - sigma(i)*I = Q(i)*R(i), so
```
      (A(i) - sigma(i)*I)^(-1) = R(i)^(-1)*Q(i)^T
      (A(i) - sigma(i)*I)^(-1)^T = Q(i)*R(i)^(-1)^T
      (A(i) - sigma(i)*I)^(-1)^T * R(i)^T = Q(i)
```
and taking the last column of both sides we get that
```
      (A(i) - sigma(i)*I)^(-1)^T * e_n and the last column of Q(i)
```
are parallel, i.e. the last column of Q(i) is gotten by a step of inverse iteration,
on A(i)^T, starting with e_n (the last column of I).
Thus the last column of Q(i) is closer to an eigenvector of A(i)^T
```
  => the last column of A(i)^T*Q(i) is closer to lambda * the last column of Q(i)
  => the last column of Q(i)^T*A(i)^T*Q(i) is closer to lambda * e_n
  => the last row of Q(i)^T*A(i)*Q(i) is closer to lambda * e_n^T,
     i.e. tiny in the first n-1 entries, and close to lambda on the diagonal
```

So where do we get sigma(i) so that it is a good approximate eigenvalue?
Since we expect A(i)_nn to converge to an eigenvalue, we pick sigma(i) = A(i)_nn.

We show that this in fact yields quadratic convergence, i.e. the error is squared at each step, so the number of correct digits doubles. To see why, suppose
   || A(i)_n,1:n-1 || = eps << 1, so that | A(i)_nn - lambda(k) | = O(eps) for some eigenvalue lambda(k), and that the other eigenvalues are much farther away than eps. Then by the above analysis, || A(i)_n,1:n-1 || will get multiplied by
     | lambda(k) - sigma(i) | / min_{j neq k} | lambda(j) - sigma(i) | = O(eps),
and so on the next iteration || A(i+1)_n,1:n-1 || = O(eps^2)

Matlab (demo), try
```
     format short e,
     n=6, D = diag(.5.^[1:n]), S = randn(n,n), A = S*D*inv(S), Z = eye(n);
```
   and repeat
```
     [Q,R] = qr(A); A = R*Q   ... should be same as Orthogonal Iteration before
```
   and then
```
     s = A(n,n); [Q,R] = qr(A-s*eye(n)); A = R*Q+ s*eye(n),
        ... does it converge quadratically?
     s = A(n,n); [Q,R] = qr(A-s*eye(n)); A = R*Q+ s*eye(n); (s-A(n,n))/s
        ... does it converge quadratically?
```

To see more explicitly why we get quadratic convergence, at least asymptotically, consider the following example in more detail:
```
    A = randn(6,6); A(6,:)=1e-4*A(6,:); A(6,6)=3;
      ... so the bottom row A(6,1:5) is small, so we are close to convergence
    s=A(6,6); [Q,R]=qr(A-s*eye(6))
      ... note that the last row Q(6,1:5) is of the same magnitude as A(6,1:5)
      ... because of the way QR works, and R(6,6) is similarly small
    R*Q
      ... note that R(6,6) multiplies Q(6,1:5), squaring their (small) magnitudes,
      ... this is the source of quadratic convergence in the next line
    A = R*Q + s*eye(6)
```

If A = A^T, convergence is even faster, cubic, for reasons explained in Chapter 5.

Making QR iteration practical:
(1) Each iteration has the cost of QR factorization plus matmul, or O(n^3). Even with just a constant number of iterations per eigenvalue, the cost is O(n^4). We want a total cost of O(n^3).
(2) How do we pick a shift to converge to a complex eigenvalue, when the matrix is real, and we want to use real arithmetic? In other words, how do we compute the real Schur form?
(3) How do we decide when we have converged?
(4) How do we minimize data movement, the way we did for matmul, etc?

Here are the answers, briefly:
(1)We preprocess the matrix by factoring it as A = Q*H*Q^T, where Q is orthogonal and H is upper Hessenberg, i.e. nonzero only on and above the first diagonal.
It turns out that QR iteration on H leaves it upper Hessenberg, and lets us reduce the cost of one QR iteration from O(n^3) to O(n^2), and so the cost of n QR iterations to O(n^3) as desired.
When A is symmetric, so that H is upper Hessenberg and symmetric, it must be tridiagonal; this further reduces the cost of one QR iteration to O(n), and of n iterations to O(n^2). We discuss this further in Chap 5.
(2) Since complex eigenvalues of real matrices come in complex conjugate pairs, we can imagine taking one QR iteration with a shift sigma followed by one QR iteration with shift conj(sigma). It turns out this bring us back to a real matrix, and by reorganizing the computation, merging the two QR iterations, we can avoid all complex arithmetic.
(3) When any subdiagonal entry H(i+1,i) is small enough,
|H(i+1,i)| =O(macheps)*||H||,
then we set it to zero, since this causes a change no larger than what roundoff does

anyway. This splits the matrix into two parts, i.e. it is block upper Hessenberg, and we can deal with each diagonal block separately. If a block is 2x2 with complex eigenvalues, or 1x1, we are done.
(4) No way is known to reduce the number of words moved to Omega(#flops/sqrt(memory_size)), as we could for matmul, LU, and QR, using this algorithm, there is lots of recent research in trying to reduce memory traffic by trying to combine many QR iterations and interleave their operations in such a way as to get the same answer, but move less data (SIAM Linear Algebra Prize 2003, to Byers/Mathias/Braman) There are other algorithms that do move as little data as matmul, using randomization, but they do a lot more arithmetic (and may someday be of more practical importance, see the paper "Minimizing Communication for Eigenproblems and the Singular Value Decomposition" at bebop.cs.berkeley.edu).

Here are some more details:
(1)Hessenberg reduction is analogous to QR decomposition: keep multiplying the matrix by Householder transformations P to create zeros. But now you have to multiply on the left and right: P*A*P (since P=P^T) to maintain orthogonal similarity:
(draw 5 x 5 example)
The code is analogous:

```
for i = 1:n-2  ... zero out matrix entries A(i+2:n,i)
  u = House(A(i+1:n,i))
  A(i+1:n,i:n) = A(i+1:n,i:n) - 2*u*(u^T*A(i+1:n,i:n)) ... multiply A = P*A
  A(1:n,i+1:n) = A(1:n,i+1:n) - 2*(A(1:n,i+1:n)*u)*u^T ... multiply A = A*P
```

The cost is (10/3)n^3 + O(n^2) just for A, or (14/3)n^3 + O(n^2) if we multiply out the Householder transformations to get Q. This is a lot more than LU or QR, and is only the first, cheap phase of the algorithm.

When A = A^T, then the resulting Hessenberg matrix H = Q*A*Q^T is also symmetric, and so is in fact a tridiagonal T. This is called tridiagonal reduction, and is the starting point for solving the symmetric eigenvalue problems (Chap 5).

For the SVD, we do something similar, but with different orthogonal matrices on the left and right to make A bidiagonal: QL*A*QR^T = B, i.e. nonzero only on the main diagonal of B and right above it. (draw 5 x 5 example)
This will be the starting point for computing the SVD in chapter 5; once we have the SVD of B = U*Sigma*V^T we get the SVD of A as (QL^T*U)*Sigma*(QR^T*V)^T

Lemma: Hessenberg form is maintained by QR iteration
proof: If A is upper Hessenberg, so is A - sigma*I, and if A - sigma*I = Q*R, it is easy to confirm that Q is also (column i of Q is just a linear combination of columns 1:i-1 of A - sigma*I). Then it is also easy to see that R*Q is also upper Hessenberg.

Finally we explain how to do one step of QR iteration on an upper Hessenberg matrix H = A - sigma*I in just O(n^2) flops, not O(n^3).

Def: an upper Hessenberg matrix H is unreduced if all the subdiagonals H(i+1,i) are nonzero. Note that if H has some H(i+1,i)=0, it is block triangular, and we can solve the eigenproblems for H(1:i,1:i) and H(i+1:n,i+1:n) independently.

Implicit Q Thm: Suppose that Q^T*A*Q is upper Hessenberg and unreduced. Then columns 2 through n of Q are determined uniquely (up to multiplying by +-1) by column 1 of Q.

First we see how to use this to do one step of QR iteration in O(n^2) flops, and then prove it. Recall that Q comes from doing A - sigma*I = Q*R, so the first column is simply proportional to the first column of A - sigma*I, namely
[A(1,1) - sigma; A(2,1); 0, ...]. Let Q(1)^T be a Givens rotation that acts on this column to zero out A(2,1) and form Q(1)^T*A*Q(1); this fills in A(3,1), a so-called "bulge", and makes the matrix no longer Hessenberg. Our algorithm will remove the bulge, multiplying by more Givens rotations Q(i), making

Q(n)^T*Q(n-1)^T ... * Q(1)^T * A * Q(1) * ... *Q(n-1)*Q(n) upper Hessenberg again, at a cost of O(n^2). Since Hessenberg form is uniquely determined by Q(1), this must be the answer. (draw picture)
Since each Q(i) moves the bulge down by one, until it "falls off", this algorithm is called "chasing the bulge".

Proof of the Implicit Q Theorem: let q_i be column i of Q. Then Q^T*A*Q=H implies A*Q = Q*H. Look at column 1 of both sides: A*q_1 = H(1,1)*q(1) + H(2,1)*q(2). This determines H(1,1), H(2,1) and q_2 uniquely, by doing QR on

$$[q_1, A*q_1] = [q_1, q_2] * \begin{bmatrix} 1 & H(1,1) \\ 0 & H(2,1) \end{bmatrix}$$

More generally, suppose we have determined q_2 , ... , q_i and columns 1:i-1 of H. We get the next column by equating the i-th columns of A*Q=Q*H to get

$$A*q_i = \sum_{j=1 \text{ to } i+1} q_j*H(j,i)$$

So $q_j^T*A*q_i = H(j,i)$ for j=1 to i, and then

$$A*q_i - \sum_{j=1 \text{ to } i} q_j*H(j,i) = q_{i+1} * H(i+1,i)$$

gives us q_i+1 and H(i+i,i).
This is the basis of the LAPACK code xGEES (for Schur form) or xGEEV (for eigenvectors), which is used by eig() in Matlab.